**MCA03**

# KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

**Housefed Complex, Dispur, Guwahati - 781 006**

## Master of Computer Applications

## COMPUTER PROGRAMMING USING C

**CONTENTS**

## Subject Expert

Prof. Anjana Kakati Mahanta, Deptt. of Computer Science, Gauhati University
Prof. Jatindra Kr. Deka, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
Prof. Diganta Goswami, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati

## Course Coordinator

Tapashi Kashyap Das, Assistant Professor, Computer Science, KKHSOU
Arabinda Saikia, Assistant Professor, Computer Science, KKHSOU

## SLM Preparation Team

| Units | Contributor |
|---|---|
| 1 , 3 | **Arabinda Saikia,** KKHSOU |
| 2, 5, 6, 7 | **Tapashi Kashyap Das**, KKHSOU |
| 4 , 8 | **Binod Deka**, System Analyst, KKHSOU |

***July 2011***



*Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.*

The university acknowledges with thanks the financial support provided by the **Distance Education Council, New Delhi**, for the preparation of this study material.

**Housefed Complex, Dispur, Guwahati- 781006; Web: www.kkhsou.net**

## COURSE INTRODUCTION

This is a course on ***Computer Programming using C.*** C language is a very popular and powerful programming language for creating computer programs. It is because, for most of the system software developments, efficiency of time and space become crucial and can be very effectively achieved by C language. C language is suitable for many applications as it has an excellent support of high-level and low-level functionality. Although several new high level languages have already been developed, C language has not lost its importance and popularity.

With this course the learners will be able to write codes in C languages. They will be able to develop programs using various features of the language.

The course consists of eight units.

The ***first unit*** introduces some basics of programming. It gives us the concept of pseudo code, algorithm, flow chart and the fundamental elements of programming.

The ***second unit*** concentrates on operators and expressions. Different types of operators like arithmetic, logical, relational, bitwise etc. are discussed in this unit. Concepts like precedence and associativity of operators are also covered in this unit.

The ***third unit*** deals with the decision and control structures. It includes Input/output functions like *scanf(), printf(), gets(), puts()*, different kinds of conditional statements and loop structures etc.

The ***fourth unit*** concentrates on storage classes. Two important concepts - *macros* and p*reprocessor* directive are also introduced in this unit.

The ***fiifth unit*** is on functions. With this unit, learners will be acquainted with function declaration, definition, function call, formal and actual parameter and the concept of recursion.

The ***sixth unit*** gives us the concept of arrays and pointers. It includes array declaration, accessing array elements, concept of strings, pointer variables, passing pointer to a function, and the most important dynamic memory allocation.

The ***seventh unit*** concentrates on structure and union. It includes structure declaration, definition, array of structures, pointer to structure, defining and declaring union and enumerated data types.

The ***eighth unit*** focuses on file handling. With this unit learners will be acquainted with different operations associted with files.

Each unit of this course includes some along-side boxes to help you know some of the difficult, unseen terms. Some “EXERCISES” have been included to help you apply your own thoughts. You may find some boxes marked with: “LET US KNOW”. These boxes will provide you with some additional interesting and relevant information. Again, you will get “CHECK YOUR PROGRESS” questions. These have been designed to make you self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with “ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

# MASTER OF COMPUTER APPLICATIONS

## Computer Programming using C

## DETAILED SYLLABUS

**Unit 1: Introduction to Programming** (Marks:12)

Basic Definition of Pseudo Code, Algorithm, Flowchart, Program, Elementary Data Types: Integer, Character, Floating Point and String Variables; Constants and Identifiers; Variable Declarations, Syntax and Semantics, Reserved Word, Initialization of Variable during Declarations, Symbolic Constants.

**Unit 2: Operators and Expressions** (Marks:12)

Expression in C; Different Types of Operators: Arithmetic, Relational and Logical, Assignment, Conditional, Increment and Decrement, Bitwise, Comma and Sizeof; Precedence and Associatively of Operators; Type Casting.

**Unit 3: Decision and Control Structures** (Marks:12)

Various Input /Output Functions: scanf, getch, getchar, printf, putchar; Conditional Statement- if, if- else, nested if-else switch; Other Statement: Break, Continue, Goto; Concept of Loops: While, Do-While, For, Nested Loop.

**Unit 4: Storage Class** (Marks:12)

Automatic, External, Static, Register, Scope and Lifetime of Variables, Macro, Preprocessor Directive.

**Unit 5: Functions** (Marks: 14)

Function: Function Declaration, Function Definition, Function Call, Function Parameters, Formal and Actual Parameter, Parameter Passing Methods, Recursive Function.

**Unit 6: Arrays and Pointers** (Marks: 14)

Arrays, 1-Dimensional Array, 2-Dimensional Array and its Declaration; String; Pointers: Declaration, Passing Pointer to a Function, Pointer and 1-Dimensional Arrays, Dynamic Memory Allocation.

**Unit 7: Structures and Union** (Marks: 12)

Structure Declarations, Definitions, Array of Structures, Pointers to Structures; Union: Definition, Declaration, Use; Enumerated Data Types; Defining Your Own Types (typedef)

**Unit 8: File Handling** (Marks: 12)

Opening, closing, reading and writing of files. Seeking forward and backward. Examples of file handling programs.

# UNIT-1 INTRODUCTION TO PROGRAMMING

## UNIT STRUCTURE



## 1.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- define a computer program
- know types of programming language and programming techniques

- describe the basic concepts of programming
- identify the structure of a C program
- describe the primary elements like identifiers, tokens, constants etc.
- know the basic data types used in C language
- declare and initialize C variables

## 1.2 INTRODUCTION

Computer programming is defined as telling a computer what to do through a special set of instructions which are then interpreted by the computer to perform some task(s). These instructions can be specified in one or more programming languages including (but not limited to) **Java**, **PHP**, C, and C++. A computer goes through a series of steps whose purpose is to achieve something - a series of steps that are instructed to it in great detail by computer programs. Essentialy, computer programming is the process by which these programs are designed and implemented. There are many advantages to learning the subject of computer programming such as gaining new skills, being able to tell the computer what to do, and becoming better acquainted with computers. So whether you are a computer hobbyist, a student, an IT professional, or are just curious about the subject, learning how to program a computer will be highly beneficial.

This unit is an introductory unit of C programming. This unit includes the types of programming language along with the programming techniques. Basic programming concepts including pseudo code, algorithm and flowcharts are briefly described here. In addition, you will come across of terms that are used in C like identifiers, keywords and constants etc. The elementary data types used in C and the concept of variables are also included as an important part of the unit.

## 1.3 DEFINING A PROGRAM

A computer program or simply a program is a set of instructions or codes which directs the computer to do some kind of specific functions. The codes or instructions are written using a programming language like FORTRAN, C, C++ etc. For a given problem a particu-

lar program or set of instructions can be design to solve it. The method of writing the instructions is called the programming. The person who writes such instruction is called a programmer. An another way of defining a program is : *a computer program is nothing but a combination of algorithm and data structure combined into a single unit which is designed to solve a given problem.*

To get the computer solution of a problem first the problem should be transformed into a computer program. Every program takes input data and manipulates it according to the instructions written in the program and finally produces the output which represents the computer solution to that particular problem.
Some characteristics of a good program are : *accurate, efficiency, reliability, portability, robustness* etc.

***Accurate*** means - the problem that has to be solved by computer must be predefined clearly with specification of requirements. Based on the requirements the programmer designs the program so that it must be accurate to perform the specified task.

***Efficiency*** means - a program should be such that it utilizes the resources of a computer system (e.g.. memory, CPU time etc.) in an efficient manner.

***Reliability*** means - a program should be work its intended function accurately even if there are temporary or permanent changes in the computer system. Program having such ability are known as reliable.

***Portability*** means - a program prepared on a certain type of computer system should run on different types of computer system. If a program can be transferred from one system to another system or one platform to another platform with ease and still it works properly then it is called portable.

***Robustness*** means - a program is expected to continue with its functionalities even at the unexpected errors. It means a program is

called robust if it provides meaningful results for all inputs (correct or incorrect). For correct data it will provide desired results and for incorrect data it will give appropriate message with no run time errors.

## 1.3.1 Program Development Cycles

The process of developing a program comprises of many steps. Before going to coding a program in a particular language, the programmer has to determine the problem that need to be solved. The program development cycle which is consists of six different steps is described in the following where each of the steps has its own significance.

**Problem definition :** This is the very important phase where the problem is analysed precisely and completely. The problem is defined with respect to the needs of the user it means that all the possible views of the users are collected and defined clearly. The ultimate final quality of the program mainly depends on this stage of program development. If this step is erroneous then it will result in poor quality of final product. The major components of program like input, processing logic, output etc. are clearly defined in this stage.

**Analysis :** This phase involves the identifying the appropriate algorithm and data structure based on the problem definition done on the above stage. The need of data structure depends on the nature of input, processing and desired output. Proper names for the identifiers can be given at this step of program development.
You will know the meaning of the terms *algorithm, identifier* etc. in the next section of this unit.

**Design Phase :** After selecting the appropriate solution, algorithm is developed to depict the basic logic of the selected solution. An algorithm depicts the solution in logical steps which is nothing but a sequence of instructions. An algorithm can be represented by flowcharts and pseudocodes. These tools make the program logic clear and helped in coding the problem.

**Testing the Algorithm for accuracy :** Before converting the algo-

rithms into actual code it should be checked for accuracy. The main purpose of checking the algorithm is to identify the major logical errors that may occur at any steps in the algorithm. Logical errors are often difficult to detect and correct at later stage so it will be helpful if the logical errors can be eliminated in this step.

**Coding :** The actual coding of the problem is done at this step by using a suitable programming language.

**Testing and Debugging :** At this step of program development, it is expected to test the complete program. The initial program codes may have errors. Generally, a program known as compiler check the code for syntax errors as well as finally produce the executable format of the whole code. Thus, the results obtained are compared with the expected results. Depending upon the complexity of the program, several round of testing may be required.

**Implementation and Documentation :** In this step of program development the program that is tested locally is taken to the user's place and installed there. This step will be absent if the program is small and simple. Moreover, the ***manuals or instruction booklet*** of the program is also prepared. The manual helps the user in understanding the program. The documentation of the program contains system flowchart to explain the complete functionality of the program at a glance.

**Maintenance :** During the operation of the program the users may come across many points where they need some changes in the program. Such types of change of a working program is belongs to the maintenance part of program development. Moreover, proper backup and restore methods also falls into the maintenance part.

The following figure shows the program development cycle :

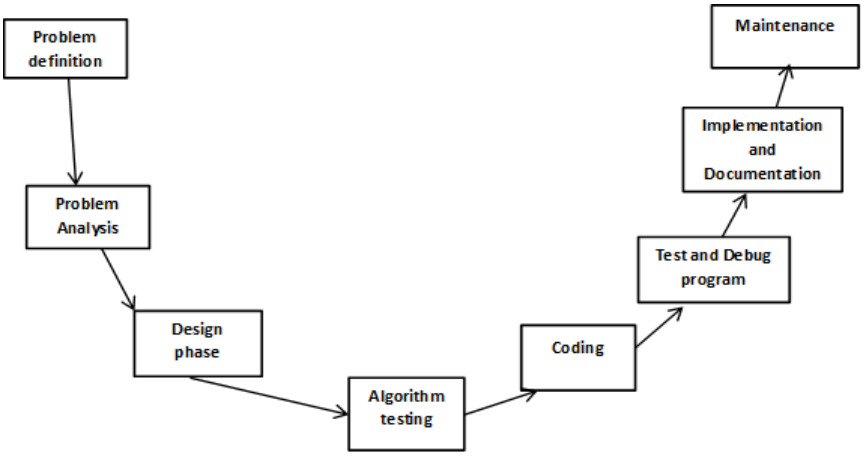



Fig. 1.1 Program development cycles

## CHECK YOUR PROGRESS

1. State True or False

a) Computer program is a combination of data structure and algorithm.

b) An accurate program means it is platform independent.

c) In program development cycle appropriate algorithm and data structure is identified in coding phase.

d) Logical errors of a program is identified during algorithm testing phase.

e) The semantic and syntax errors in a program is checked in testing & debugging phase.

## 1.4 TYPES OF PROGRAMMING LANGUAGE

The programming language is the medium through which the problems to be solved by computer can be represented in the form of programs or set of instructions. The computer executes the set of instructions written in a particular language to produce the desired result. A programming language consists of a set of characters, symbols, and usage rules that allow the user to communicate with computers. In a simple form it can be defined as - *any notation for the description of algorithm and data structure may be termed as a pro-*

*gramming language*.

The computer understand only the binary language i.e. the languages of 0 and 1, which is also called machine language. In the initial years of computer programming the computer programs were written using machine language which were too difficult to remember all the instructions in the form of 0 and 1s. But with due course of time the other languages were evolved along with the generation of computers. There are different levels of programming language as follows :

- Machine language
- Assembly language
- High level language
- 4GL language

The different levels of programming language is shown in the figure below :

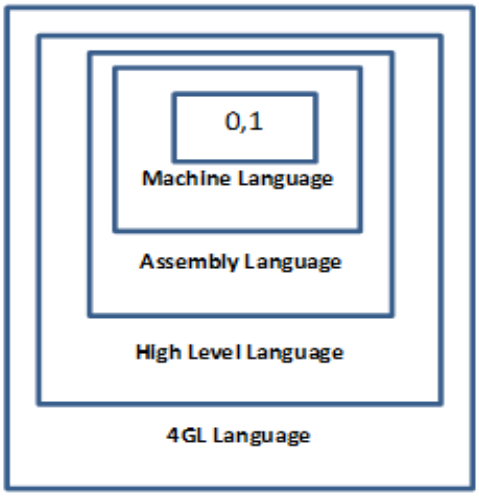



Fig. Levels of Programming Language

i) **Machine language :** The machine language programs were written using the digit 0 and 1. The instruction in machine language consists of two parts. The first part is an operation which tells the computer what operation is to be performed. The second part of the instruction is operand, which tells the computer where to find or store the data after applying the operation.

**Advantage :**

*Translation free* **:** Machine language is directly understand by the computer so it does not need any translation. Otherwise, if a program is developed in high level language then it has to be translated into the machine language so that the computer can understand the instructions.

*High speed :* The application developed in the machine language are extremely fast because it does not require any conversion.

**Disadvantage :**

*Machine dependent :* Program developed in one computer system may not run on an another computer system.

*Complex language :* This language was very difficult to understand and programming techniques were tedious.

*Error prone :* Writing machine language programs, a programmer has to remember all the opcodes and memory locations, so machine language is bound to be error prone.

**ii) Assembly language :** Assembly language is not a single language, but a group of language. An assembly language provides a mnemonic instruction, usually three letters long, corresponding to each machine instruction. The letters are usually abbreviated indicating what the instruction does, like ADD for addition, MUL for multiplication etc. The assembly language allows the programmer to interact directly with the hardware. Each line of assembly language program consists of four columns called fields. The general format of an assembly language instruction is :

| | **[Label]** | **<Opcode>** | **<Operands>** | **[; Comment]** |
|---|---|---|---|---|
| **e.g.** | **BEGIN** | **ADD** | **A, B** | **; Add B to A** |

Assembler is a software tool that is used to convert the assembly language program into a machine language program.

**Advantage :**

*Easy to understand and use :* Assembly language uses mnemonics instead of using numerical opcodes and memory locations used in machine language, so it is easy to understand and use.
*Less error prone :* Assembly language is less error prone and it provides better facility to locate errors and correct them.
*Faster :* It is much faster and use less memory resources than the high level language.

**Disadvantage :**

- Assembly language programs are machine dependent.
- Assembly language is complex to learn.
- A program written in assembly language is less efficient than machine language because every assembly instruction has to be converted into machine language.

**iii) High level language :** COBOL, FORTRAN, PASCAL and C are the examples of high level languages. High level languages are similar to English language. Programs written using these languages may be machine independent. A single instruction of a high level language can substitute many instructions of machine language and assembly language. Using the high level language complex software can be design.

**Advantages :**

- *Readability :* Since high level languages are similar to English language so they are easy to learn and understand.
- Programs written in high level languages are easy to modify and maintain.
- High level language programs are machine independent.
- Programs written in high level language are easy to error checking. Due to the concept of abstraction used in such type of language the programmers don't have to mind on the hardware level representation of programs.

**Disadvantages :**

Programs written in high level language has to be compile first for

execution of the program. This step of compilation increases the execution time of an application. Moreover, the programs occupies more memory space during the execution time.

Another main draw back of HLL programs is - its poor control on the hardwares.

**Fourth generation language :** 4G languages has the special characteristics that they are closer to human languages so they are easy to learn. 4GL languages are specially designed for some specific applications with limited set of functions, so they are easy to learn. Oracle, VB, SQL, VC++ etc are the example of 4GL languages. Most of the database accessed languages are 4GL language.

### 1.4.1 Language Translators

The language translators or also called language processors are the programs which converts the high level language programs into the machine language programs for execution. We know that the computer understand only the machine language, so for execution of a program written in HLL must be converted into the machine language codes. The examples of language translators are - compiler, interpreter, assembler etc. Let us see them very briefly.

**Compiler :** A compiler is a program that can translates an higher level language program to its machine form. This language processor translates the complete source program as a whole into machine code before execution. Here the source program means - the program that is written by the programmer and the translated program is called a object program or object code. Examples: C and C++ compilers. If there are errors in the source code, the compiler identifies the errors at the end of compilation. Such errors must be removed to enable the compiler to compile the source code successfully. The object program generated after compilation can be executed a number of times without translating it again.

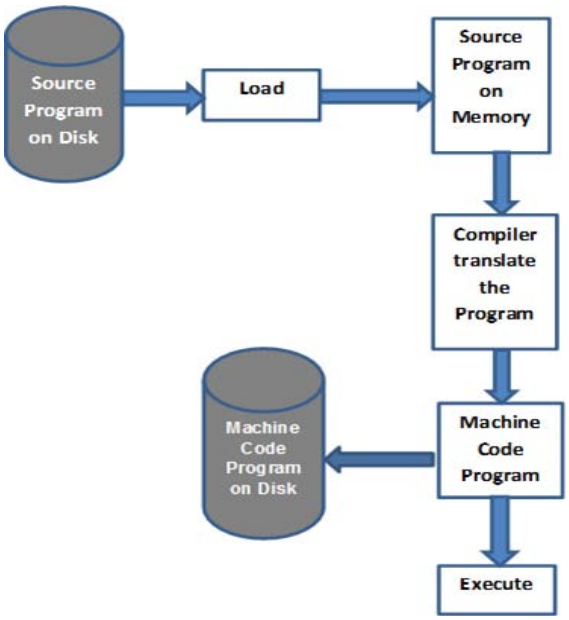



Fig. 1.3 A compiler translate a program

**Interpreter** : It is a language processor whose working principle is different from compilers. It translates each statement of source program into machine code and executes it immediately before translating the next statement. If there is an error in the statement the interpreter terminates its translating process at that statement and displays the error message. The GWBASIC is an example of interpreter.

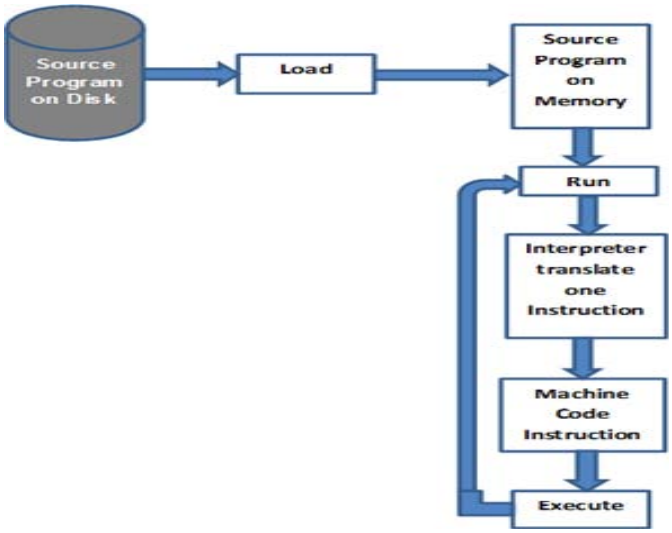



Fig.1.4 An Interpreter translate a program

**Assembler :** We have already known that - *assembly language* is the symbolic representation of a computer's binary encoding - **machine language** . Assembly language is more readable than machine language because it uses symbols instead of bits.

A program called an **assembler** that translates assembly language into binary instructions or machine language.An assembler reads a single assembly language *source file* and produces an *object file* containing machine instructions.

**Linker** : A linker is a program that attached or linked several modules of a program. Generally, a program written for a particular application may be consists of thousands or more lines of codes. In such a case, the large program can be broken into a number of smaller program which is called a *module*. Each module will perform its specific task and compiled them separately. Finally each module have to be linked together to construct the complete application. A linker is a program that links a number of modules (object codes, which are generated after the compilation step) and library functions to form a single executable program.

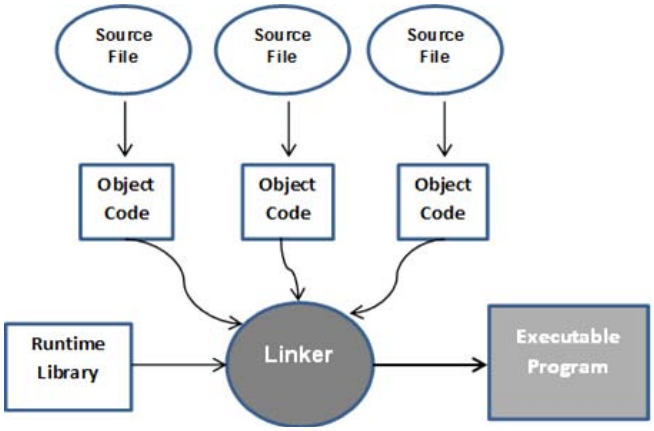



Fig. 1.5 Working of a Linker

**Loaders** : A loader is a program and that is also a part of the operating system and whose main function is to bring an executable program residing on the disk into main memory and start running it. Generally, a loader performs the function of a linker program also.

The loader performs the following tasks :

Allocation : It allocates memory space for programs
Linking : It combines two or more modules and supplies the information needed to allow reference between them.
Relocation : It prepares a program to execute properly from its storage area.
Loading : It loads the program into memory.

There are two types of loaders depending on the way the loading is performed : *absolute loader and relocating loaders*.
The absolute loader load the executable code into memory locations specified in the object module. On the other hand, relocating loader loads the object code into memory locations which are decided at load time. The relocating loader can load the object code at any location in memory.

**CHECK YOUR PROGRESS**

2. State True or False
a) Assembly language instructions are consisting with 0 & 1.
b) High level languages are machine independent.
c) Query languages are fourth generation language.
d) Interpreters generate a complete executable code.
e) Linkers are responsible for moving a program from disk into main memory.
3. What is a compiler? What do you meant by the compilation process?
4. What is a source program and an object program ?

## 1.5 BASIC TOOLS OF PROGRAMMING

So far, we have came to know that programming is a technique of solving problems using program codes. The program codes are designed by following specific rules and using some programming tools. The general tools used for designing programs are : pseudo

code, algorithms, flowcharts etc. In this section, we will discuss briefly about those programming tools.

## 1.5.1 Pseudo Code

**Pseudo code** derived from '*pseudo*' which means imitation and '*code*' means instruction (pronounced as soo-doh-kohd) is a generic way of describing an algorithm without using any specific programming language related notations. Pseudocode is an artificial and informal language that helps programmers to develop algorithms. It is a "text-based" detail (algorithmic) design tool. Pseudo code generally consists of short English phrases to express a specific task. But, interestingly, there are no specific rules to write pseudo code. Then why do we need pseudo code? Implementing a problem with proper symbol and syntax is a complicated job. To solve the complicated job, first we need the complete explanation of the job, and then we extract information from the explanation, and roughly design the solution, which will be relevant to a programming construct. This rough work will help in designing the proper solution with proper symbol and syntax.

The general guidelines for developing pseudocodes are :

- Statements should be written in English and should be programming language independent.
- Steps of the pseudocodes must be understandable.
- It should be concise.
- Each instruction should be written in a separate list and each statement in pseudocode should express just one action for the computer.
- The keywords like READ, PRINT etc should be write in capital letter.
- Each set of instruction should written from top to bottom, with only one entry and one exit.

**Advantage :**

Its language independent nature helps the programmer to express the design in plain natural language.

Based on the logic of a problem it can be designed without concerning the syntax or any rule.

It can be easily translated into any programming language.
It is compact in nature and can be easily modify.

**Limitations :**
It is unable to provide the visual presentation of the program logic.
It has not any standard format or syntax of writing.
It cannot be compiled or executed.

The following is an example of pseudocode :

```
1. If student's grade is greater than or equal to 60
                Print "passed"
        else
                Print "failed"
```

## 1.5.2 Algorithm

Algorithm is a sequential, finite, non-complex, step-by-step solution to a problem written in English like statement. It can also be defined as - *an algorithm consists of a set of explicit and unambiguous finite steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in a finite time.*

The writing of an algorithm follows the algorithmic notations. The algorithmic notations are the symbolic representation of the steps of the algorithms.

Problem: Design an algorithm to add two integers and display the result.

Solution: First of all you need two integers. There are two ways to get the two integers. First, you can supply the two integers from your side; secondly, you can ask the user to supply the integers. After getting the integers you store it into two containers. You need another container to store the sum of the two integers. You must initialize the third container as zero before you store the summation. Now, perform the addition task and store the result into the third container.

Now, display the content of the third container. This is the solution to your problem.

Step 1: Ask the user to supply the first integer.
Step 2: Store the first integer into container A.
Step 3: Ask the user to supply the second integer.
Step 4: Store the second integer into container B
Step 5: Initialize container C as zero.
Step 6: Add the content of container A and B and store the result into container C.
Step 7: Display the content of the container C.
Step 8: End.

The above example can be expressed in a different way :

Step 1 : Read a, b, c
[a, b, c are integers]
Step 2 : c ← 0
Step 3 : c ← a+b
Step 4 : Print c
Step 5 : Stop

Here, in the above expression, we have input the numbers in a single line instead of three different line. We have used here a comment line within the **[ ]** brackets. Again we have assign the value 0 to **c**. We have add the values **a** & **b** and store it in **c**. Finally we display the value of **c**.

**Always remember :**

i) Usually word *Read, Accept or Input* can be used to represent input operation to give values of variables to the computer.
ii) Similarly for the output operation the words Print, Write or Display can be used.
iii) The symbol ← is used in case of assignment (i.e. = ) operation. It means the value obtained by evaluating the right side expression is stored to the left side variable.
iv) The comments are non-executable statements written within pair

of square brackets and they are generally used to explain a particular step.

v) The steps are written sequentially and if two statements are written in a single line then they have to separate by semicolon '**;**'.

For example :

```
sum←0; a←25
sum←sum+a
Print sum
```

vi) In case of branching or conditional steps - ***If-Then*** or ***If-Then:Else*** is used. The conditional steps are consisting with the relational operators like - <, >, <=, >= etc.

For example :

```
If condition Then :
        Step 1
        Step 2
        ....
        Step n
        [ End of If ]
```

```
If condition Then
        Step i
Else
        Step j
 [ End of If ]
```

vii) The iterative or repetitive steps can be write between **Repeat For** and **[ End of For ]** as shown below :

```
Repeat For I=1,2,3,....N
        Step 1
        ....
        ....
[ End of For I ]
```

**or**

```
I←N
Repeat While I>0
        Step 1
        ....
        I←I-1
[ End of While ]
```

For example,

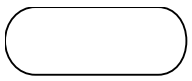
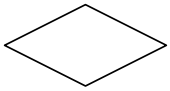
```
Fact←1
Repeat For I=1,2,3,...N
        Fact←Fact*I
[ End of For I ]
Print Fact
```

The following algorithm find the biggest of given three numbers :

Step 1 : Read a, b, c
Step 2 : big ← a
Step 3 : If b>big Then
big ← b
Step 4 : If c>big Then
big ← c
Step 5 : Print big
Step 6 : Stop

## 1.5.3 Flow Chart

A ***flowchart*** is a pictorial representation of an algorithm that uses boxes of different shapes to denote different types of instructions. A flowchart acts as a road map for a programmer and guides the programmer as to how to go from the starting point to the final point while writing a computer program. A flow chart will also helps the programmer at the time of analyzing, debugging and testing the programs. The main disadvantage of flowchart is that it is very time consuming and laborious to draw with proper symbol, especially for large complex program. The following symbols are generally used to draw a flowcharts :

| Symbol | InstructionType | Symbol | Instruction Type |
|---|---|---|---|
|  | Start/Stop |  | Decision |
|  | Input/Output |  | Connector |
|  | Process |  | Flow of Data |

The general rules for drawing a flowchart is given below :

a) The flowchart should contain one Start and one Stop terminator.

b) The symbols of the flowcharts are always labeled with simple codes.

c) The codes used in flow chart should not give more than one meaning.

d) The control flows are always represented by arrows.

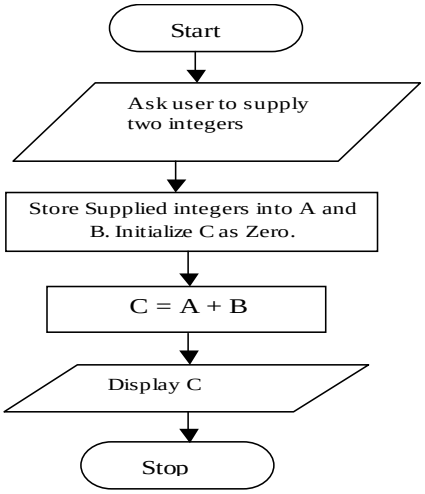
e) The control flow lines i.e. arrows should not cross each other.

f) The arrows moves from either vertically (top to bottom vice versa) or horizontally (left to right vice versa).

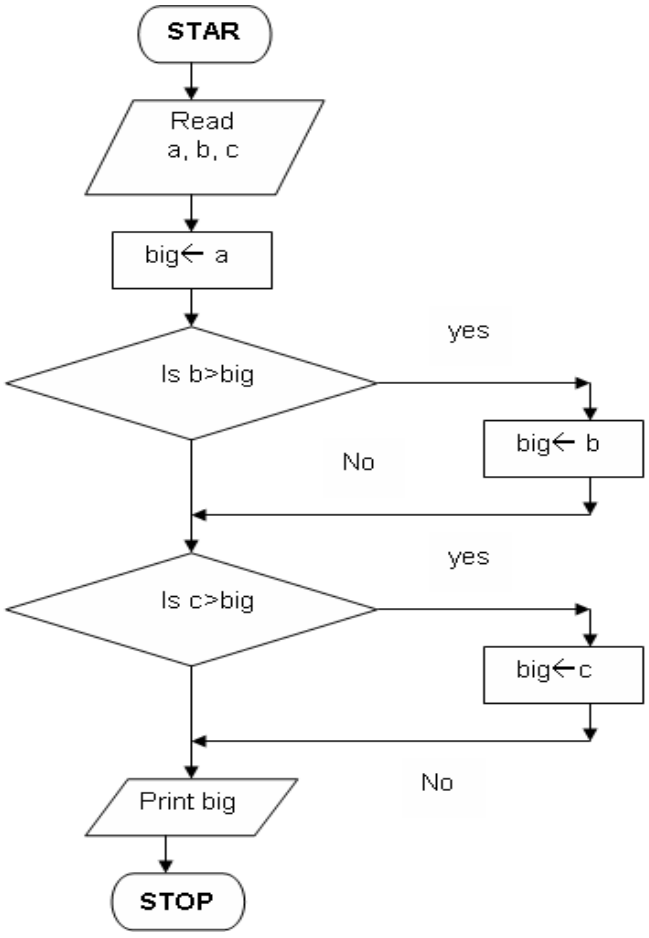
g) Only one flow line comes out from all the boxes or symbols except the decision box.

h) Two lines can flow from the decision b ox if single condition is checked. It means a single condition results in one of the two values TRUE or FALSE.

If we try to draw the flowchart of the algorithm we have written for finding the addition of two integers and display the result, then it will look as shown below :




The following flowchart is for the program for finding the greatest number from given three numbers.




## CHECK YOUR PROGRESS

5. In an examination 50 students obtain marks in their 10 subjects. Draw a flowchart for calculating the average percentage marks of 50 students. The flowchart should show the counting of the number of students that appeared in the examination and the calculation should stop when the number of counts reaches number 50.

6. Write an algorithm to generate a Fibonacci sequence.

## 1.6 PROGRAMMING TECHNIQUES

The programming techniques refers to the design and writing of programs to solve a problem or task. During the problem definition the programming techniques can be clearly understand. The following are the techniques used in programming :

- Top-down approach
- Bottom-up approach
- Unstructured technique
- Structured technique
- Modular programming

**Top down approach :** In this approach, the given problem is divided into two or more sub problems, each of which resembles the original problem. The solution of each sub problem is taken out independently. Finally, the solution of all sub problems is combined to obtain the solution of the main problem. The following figure shows the meaning of top down approach.

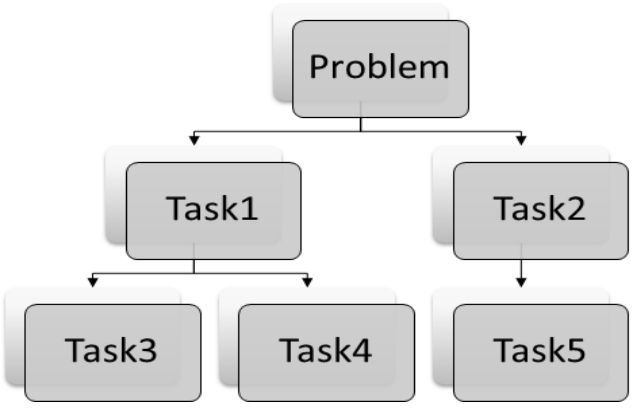



Fig. 1.6 Top-down approach

**Bottom-up approach :** This technique is just reverse of the top down programming. In this programming technique, the solutions of the independent sub-problems are designed first. Then these solutions are combined or composed in a main module in order to design the final solution of the problem. The following figure shows the bottom-up approach :

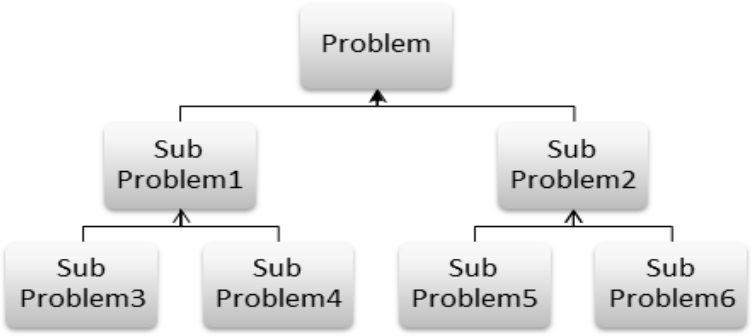



Fig. 1.7 Bottom-up approach

**Unstructured Technique :** This designing technique is refers to writing small programs using series of statements and having only the main program. All actions such as providing input, processing and output are done within one program only.The program design as well as the logic of the program is simple. To branch from one point to another point within the program is achieved by ***goto*** statement. There are a lots of difficulties seen in this type of technique. Testing and redesigning of the programs take more time.

**Structured Technique :** In structured technique, a program is broken down into small independent task that are small enough to be understood easily, without having to understand the whole program at once. Each task has its own functionality and perform specific part of the actual processing. Each task is again decomposed into subtask if necessary. The programs designed using the structured technique are well organized. In such type of technique, program design is simple and testing and debugging is easy because of the well defined control structures. The followings are some of the reasons for preferring structured programming :

a. It is easier to write a structured program - Complex programming problems or program are broken into a number of smaller, simpler tasks. Every task can be assigned to a different programmer and/or function.

b. It's easier to debug a structured program - If the program has a bug , a structured design makes it easier to isolate the problem to a specific section of code.

c. Reusability - Repeated tasks or routines can be accomplished using functions. This can overcome the redundancy of code writing, for same tasks or routines, it can be reused, no need to rewrite the code, or at least only need a little modification.

The three basic building blocks for writing structured programming are given below :

a) Sequence structure

b) Loop or Iterations

c) Binary decision structure

**Modular Programming :**

The technique of breaking down a large problem into a number of smaller program units known as ***module,*** is called a modular programming approach. Each module is designed to perform a specific function. There is one entry and one exit point for each module. Each individual modules can be easily tested and debugged. In case of modular programming, program maintenance is easy as the module showing errors can be easily detected and corrected. A large problem can be easily monitored and controlled using this technique.

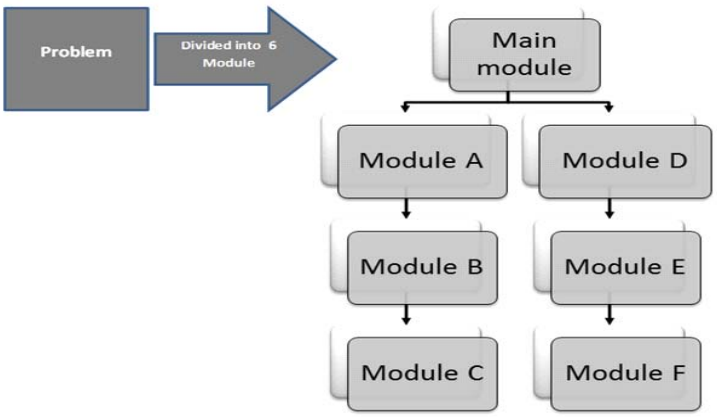



Fig. 1.8 Modular technique

## 1.7 THE C LANGUAGE

The programming language C is a general purpose computer

language. It is a structured, high-level and machine independent language. It was originally created for the specific purpose of writing Operating System software.

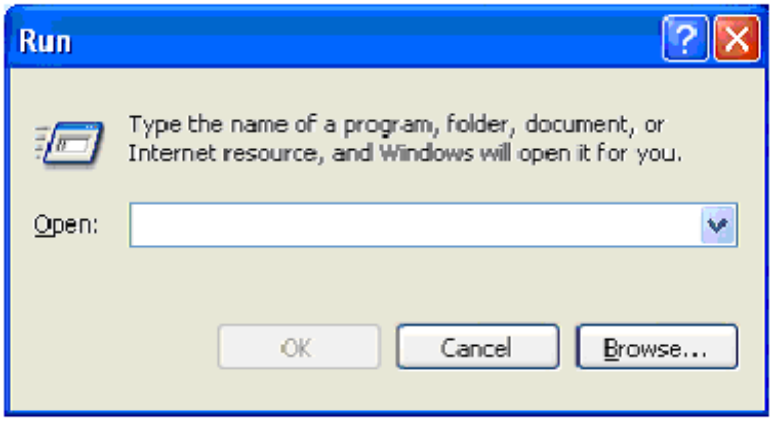
The development in C has seen many evolutionary processes. Like any other programming language, the original version of the C language has undergone a number of revisions. A lot of new features have been added to make it more useful and powerful. C was evolved from ALGOL, BCPL and B by ***Dennis Ritchie*** at the *Bell Laboratories* in 1972. C uses many concepts from these languages and added the concepts of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. For many years , C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among the computer professionals. Today C is running under a variety of operating systems and hardware platforms. During 1970s, C had evolved into what is now known as "traditional C". The language became more popular after the publication of the book 'The C Programming Language' by **Brain Kerningham and Dennis Ritchie** in 1978. The book was so popular that the language came to be known as "K&R **C**" among the programming community.

To assure that the C language remains standard, in 1983, *American National Standards Institute* (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989 which is now known as ***ANSI C***. It was then approved by the *International Standards Organization* (ISO) in 1990.

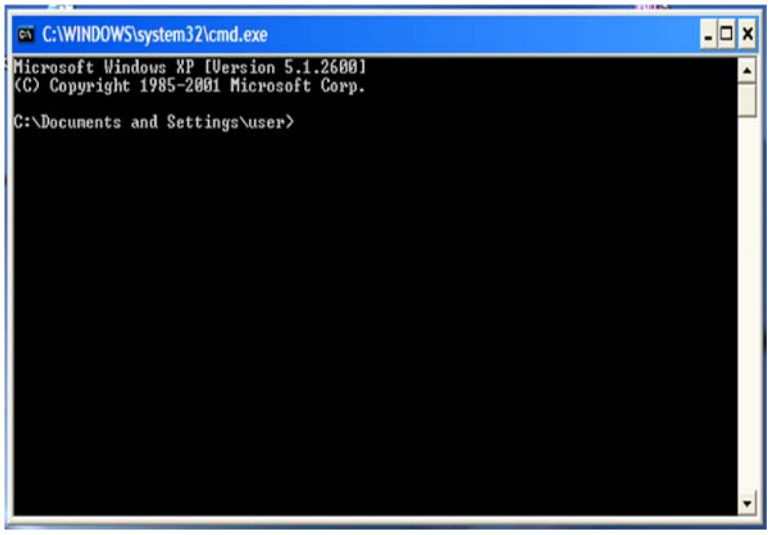
## 1.8 GETTING STARTED WITH C

The best way to get started with is to actually look at the program. So, let us first create a file "first.c". To create first.c file, we have to follow the following steps:–:

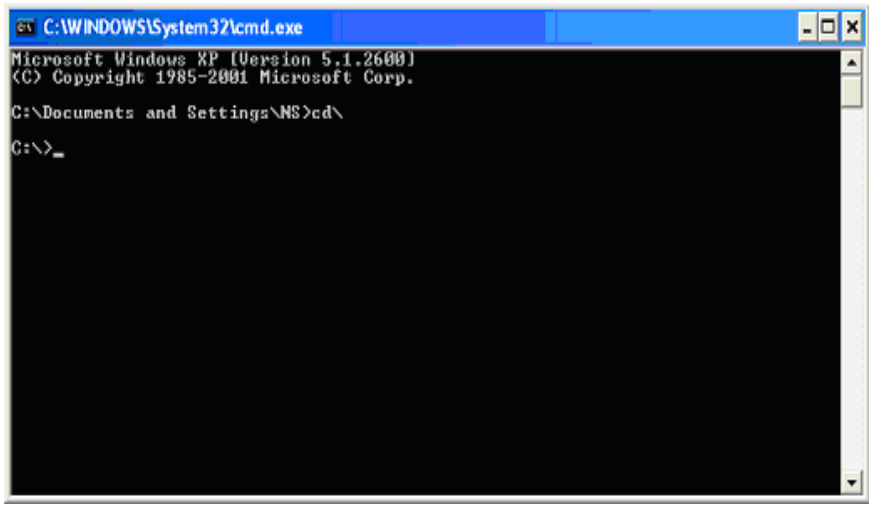
Step 1: Open Turbo C++ Editor from the icon on your desktop. If it is not in the desktop then go to Start and click on Run command from the Pop up Menu. The following command window will appear.



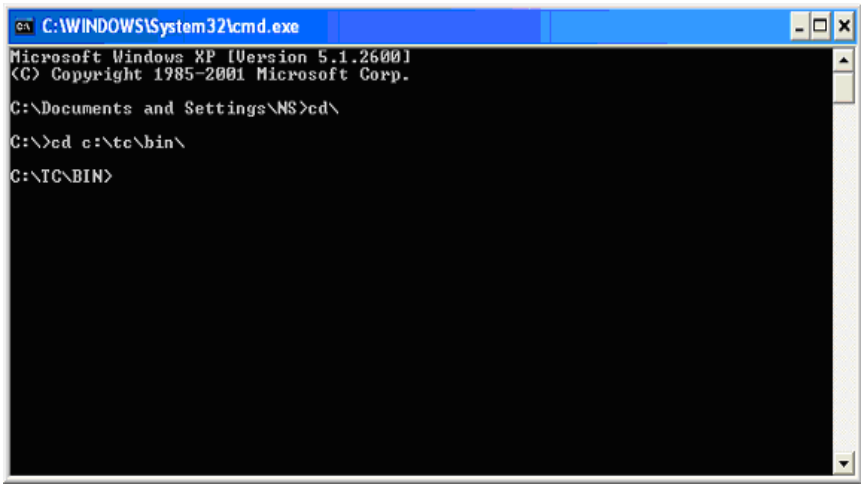

In the open field type cmd and press enter key. The command prompt will appear as an active window.



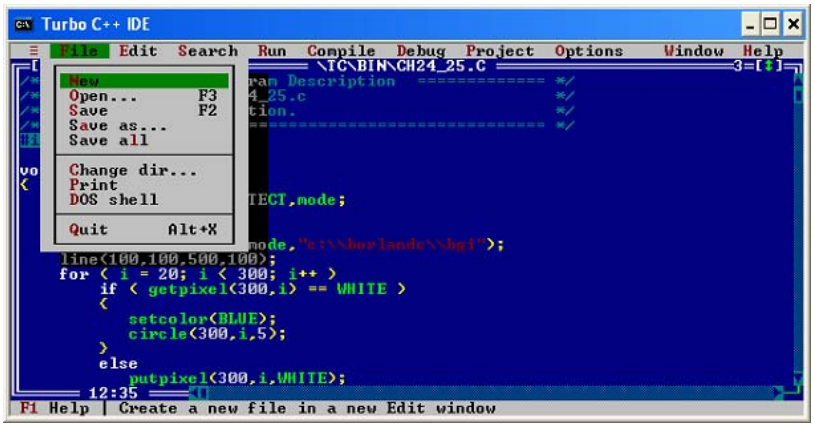

Now type cd\ . Now file pointer is in root Directory that is in C:\



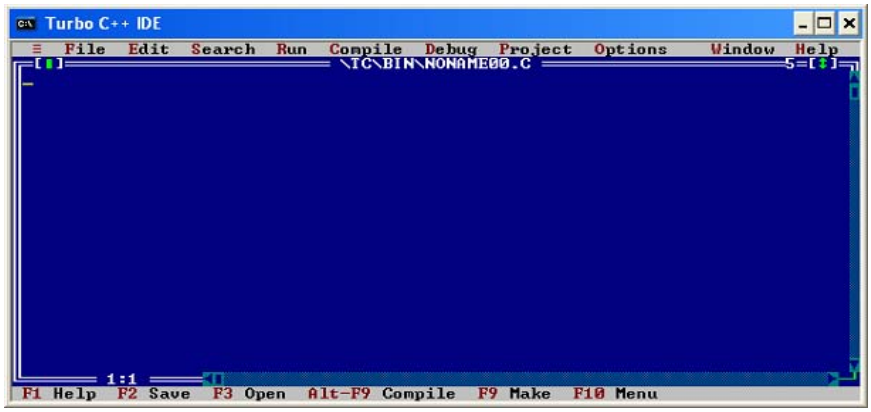

Now type the path for Turbo C++ Editor that is cd c:\tc\bin and press enter. The following window will appear.



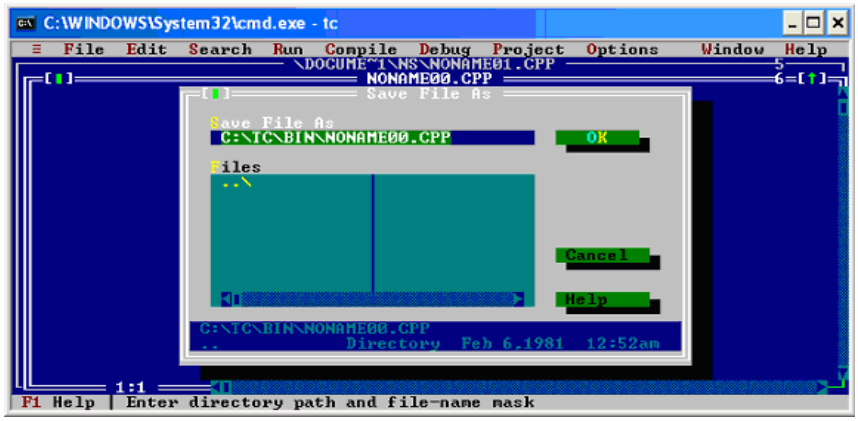

Now Type **TC** and press **Enter**. The Turbo C++ Editor will open. Go to **File** Menu and



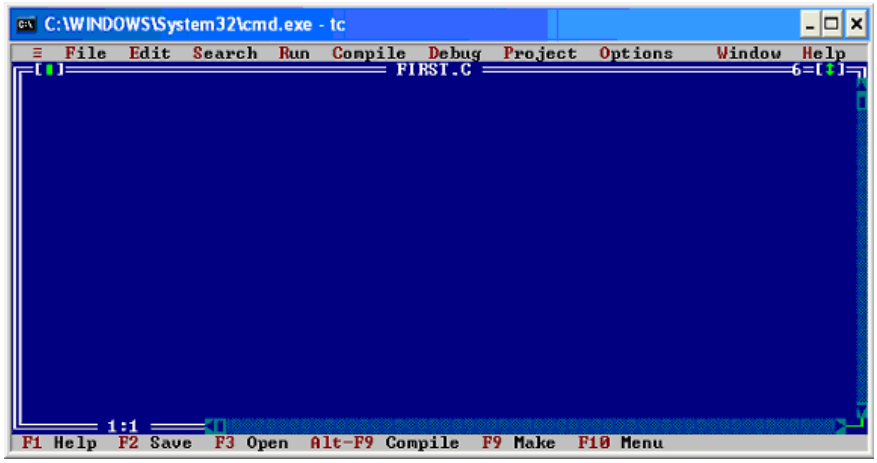
Step 2: After clicking the New submenu the following window will appear.




Step 3: Now click on save submenu of file Menu.




Step 4: Type the file name that is c:\tc\bin\first.c and press **enter**. The following window will appear.




At the end of this step, you successfully create a C source file name

first.c and save the blank file. Now type the following code in your first.c program.

```
#include<stdio.h>
   int main()
    {
          printf("This is output from my first program!\n");
          return 0;
    }
```

Let us dissect our first program

- This C program starts with **#include <stdio.h>**. This line **in cludes** the "standard I/O library" into your program. The stand ard I/O library lets you read input from the keyboard (called "standard in"), write output to the screen (called "standard out"), process text files stored on the disk, and so on. It is an extremely useful library. C has a large number of standard libraries like stdio, including string, time and math libraries. A **library** is simply a package of code that someone else has written to make your life easier (we'll discuss libraries a bit later).

- The line **int main()** declares the main function. Every C program must have a function named **main** somewhere in the code. We will learn more about functions shortly. At run time, program execution starts at the first line of the main function.

- In C, the **{** and **}** symbols mark the beginning and end of a block of code. In this case, the block of code making up the main function contains two lines.

- The **printf** statement in C allows you to send output to standard out (for us, the screen). The portion in quotes is called the **format string** and describes how the data is to be formatted when printed. The format string can contain string literals such as "This is output from my first program!," symbols for carriage returns (\n), and operators as placeholders for variables (see below).

- The **return 0;** line causes the function to return an error code of 0 (no error) to the shell that started execution. More on this capability a bit later.

Now to compile this source code press **Alt+F9** key or click compiler submenu of *compiler* menu. If there is error in the source program then the compiler will point out the error with line number. On successful compilation, the compiler will give a message that the source program is error free. Press **Ctrl+F9** (or click submenu run of *run* menu) to execute the compiled source code. This will give the required output. Now to see the output generated, press **Alt+F5** key.

**CHECK YOUR PROGRESS**

7. What is structured programming ?

8. What is a header file? What is a library function ?

## 1.9 FUNDAMENTALS OF C LANGUAGE

In this section, we will concentrate on the basic elements of C language. The basic elements of C language can be classified as follows :

- Identifiers
- Keywords
- Constants
- C Character set

### 1.9.1 Identifiers

Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consist of letters and digits in any order except that the first character must be a letter.To construct an identifier you must obey the following points :

- Only alphabet, digit and underscores are permitted
- An identifier can't start with a digit.
- Identifiers are case sensitive, i.e. uppercase and lower case letters are distinct.
- Maximum length of an identifier is 32 characters.

The following names are valid identifiers :

| | | | |
|---|---|---|---|
| **x** | **y12** | **sum_1** | **temperature** |
| **names** | **area** | **tax_rate** | **table** |

The following names are not valid identifiers for the reasons stated:

| | |
|---|---|
| **x"** | illegal characters ( " ). |
| **order-no** | illegal character ( - ) |
| **total sum** | illegal character ( blank space) |

## 1.9.2 Reserved Word

Reserved words are the essential part of a language definition. The meaning of these words has already been explained to the C compiler. So you can't use these reserved words as a variable names. Since these reserved words have some special meaning in C, therefore these words are often known as "keyword". In the following shows all the reserved word available in C.

| | | | |
|---|---|---|---|
| *auto* | *double* | *if* | *static* |
| *break* | *else* | *int* | *struct* |
| *case* | *enum* | *long* | *switch* |
| *char* | *extern* | *near* | *typedef* |
| *const* | *far* | *register* | *union* |
| *continue* | *float* | *return* | *unsigned* |

| | | | |
|---|---|---|---|
| *default* | *for* | *short* | *void* |
| *do* | *goto* | *signed* | *++ while* |

## 1.9.3 Constants

A ***constant*** is a container to store value. But you can't change the value of that container (constant) during the execution of the program. Thus, the value of a constant remains constant through the complete program.

There are two broad categories of constant in C, ***literal constant*** and ***symbolic constant***. A literal constant is just a value. For example, 10 is a literal constant. It does not have a name, just a literal value. Depending of the type of data, literal constant is of different type. They include *integer*, *character* and *floating point* constant. Integer constant can again be subdivided into *decimal* (base-10), *octal* (base-8), and *hexadecimal* (base-16) integer constant. One important variation of character constant is *string constant*. Table 1.4 explains the different types of literal constant. Remember that a character constant is always enclosed with single quotation mark, whereas a string constant is always enclosed with a double quotation mark. Another point to remember is that an octal integer constant always starts with 0 and a hexadecimal constant with 0x.

| **EXAMPLE** | **TYPES** |
|---|---|
| 153 | Decimal integer constant |
| 015 | Octal integer constant |
| 0xA1 | Hexadecimal integer constant |
| 153.371 | Floating point constant |
| 'a' | Character constant |
| '1' | Character constant |
| "a" | String constant |
| "153" | String Constant |

## 1.9.4 C Character Set

C does not use, nor does it require the use of, every character found on a modern computer keyboard. The only characters required by the C Programming Language are as follows :

| | |
|---|---|
| Alphabets | A - Z<br>a - z |
| Digits | 0 - 9 |
| Special Symbol | # & \| ! ? _ ~ ^ { } [ ] ( ) < ><br>space . , : ; ‘ $ “ + - / * = % |

## 1.10 BASIC DATA TYPES IN C

Now, we have to start looking into the details of the C language. To process with data first of all you must know the type of the data. Data type of C has 3 distinct categories. Figure 1.1 explains the different categories of C data type.

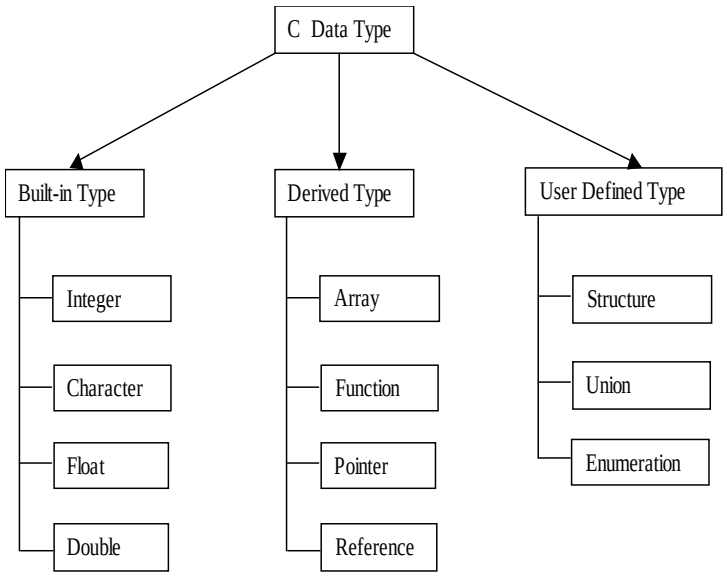



Fig 1.1: Classification of C data type

The first category of data type is the ***built-in*** data type, which are also known as elementary or basic type. Sometime these are called the “primitive” type. These basic data type have several type modifiers, which alter the meaning of the base data type to yield a new type. Table 1.1 lists all combinations of the basic data types and modifiers along with their size and ranges:

| Type | Size (Bytes) | Range |
|---|---|---|
| char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| signed char | 1 | -128 to 127 |
| int | 2 | -32768 to 32767 |
| unsigned int | 2 | 0 to 65535 |
| signed int | 2 | -32768 to 32767 |
| short int | 2 | -32768 to 32767 |
| unsigned short int | 2 | 0 to 65535 |
| signed sort int | 4 | -32768 to 32767 |
| long int | 4 | -2147483648 to 2147483647 |
| unsigned long int | 4 | 0 to 4294967295 |
| signed long int | 4 | -2147483648 to 2147483647 |
| float | 4 | 3.4E-38 to 3.4E+38 |
| double | 8 | 1.7E-308 to 1.7E-308 |
| long float | 10 | 3.4E-4932 to 1.1E+4932 |

Table 1.1: Size and range of basic data type and its modifier

Moreover, besides these basic data types another special data type is ***void***. The void type specifies the return type of a function, when it is not returning any value. Sometime void is used to indicate an empty parameter to a function. After all, this data type holds the literal meaning of void. At this point don’t worry about the other two categories. All these will be discussed in the subsequent unit.

The “%” symbol along with a (special) character in known as ***format specifier*** or ***conversion specifier***. It indicates the data type to be

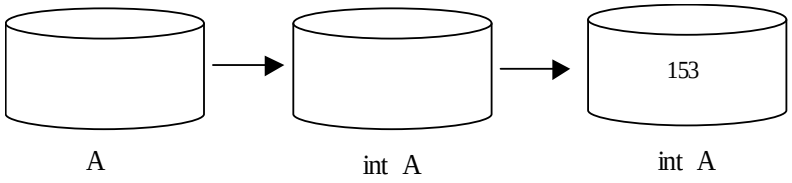
printed or scanned and how that data type is converted to the character that appears on the screen. Format specifiers for usual variable types are shown in Table.1.2.

| Format Specifier | Usual VariableType | Display as |
|---|---|---|
| %c | char | single character |
| %d | int | signed integer |
| %f %lf | float or double | signed decimal |
| %e %le | float or double | exponential format |
| %o | int | unsigned octal value |
| %u | int | unsigned integer |
| %x | int | unsigned hex value |
| %ld | int | long decimal integer |
| %s | array of char | sequence of characters |

Table 1.2: Format specifier for usual variable type

## 1.11 C - VARIABLES AND THEIR DECLARATIONS

A variable is an identifier to store value. You can resemble a variable with a container which takes different values at different time during the execution of the program. Thus, the value of the variable may change within the program. A variable name can be chosen by the programmer in a meaningful way that reflects what it represents in the program. The naming convention of variable follows the rule of constructing identifiers. Suppose you want to store value 153 to a variable. What you will do is – first create the name of the variable, suppose A. Since 153 is integer so declare the variable A as integer, and then assign 153 to that variable.




Now, in C programming language this can be done using the follow-

ing statement :

```
int A ;
A = 153 ;
```

The first statement says that A is a container, where we can store only integer type variable. This means that we can't store value into A other than integer. Therefore this type of statement is known as declaration statement (A declares that A can store only integer type of variable). Thus the general form of declaration of a variable is

**data_type variable1, variable2, . . . . . , variableN;**

By declaring a variable you tell 3 things to the compiler :

- What the variable name is.
- What type of data the variable will hold.
- and the scope of the variable.

Up to this point container A is empty. The second statement says that the value 153 is stored in A. This means variable A is initialized with 153. Therefore this type of statement is known as variable initialization. A variable must store a value after it has been declared (but before it is used in an expression). You can store values to a variable in two ways

: 
- By using assignment statement.
- and by using a read statement.

The first method is used in the above example. In second approach you can make a call of C standard input function (that is *scanf, getch, getc, gets* etc.) to store value to a variable. For example, the above initialization statement can be written as

**scanf("%d",&A);**

This statement will take an integer type input from standard input device (that is keyword) and store it to A.

The above two statement (program segment) can be written in a single statement.

**int A = 153;**

This type of statement is known as initialization of variable during declaration. As a shorthand, you can declare variables that have the same type in a single line of declaration by separating the variable names with commas. For example, you can declare the variable j and k in a single line as :

**int j, k;**

which is the same as the declaration of j and k as :

**int j;**
**int k;**

It is always a good practice to group together declarations of the same data type for an easy reference. For example :

**int j, k;**
**float x,y,z;**

A few examples of variable declarations are shown below :

| **Variable declaration** | **Remarks** |
|---|---|
| int i = 0, j = 1; | *i* and *j* are declared as integer variables. The variables i and j are initialized with value as 0 and 1 respectively. |
| float basic_pay; | *basic_pay* is a floating point variable with a real value or values containing decimal point. |
| Char a; | *a* is a character variable that stores a single character. |
| double theta; | *theta* is a double precision variable that stores a double precision floating point number. |

## 1.12 SYMBOLIC CONSTANTS IN C

A *symbolic constant* is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a

character constant is a string constant. Thus, a symbolic constant allows a name to appear in place of a numeric constant, character constant or a string. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence.

Symbolic constants are usually defined at the beginning of a program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc. that the symbolic constants represent.

Symbolic constants are defined using *#define* as given below:

**#define<symbolic constant name> <value>**

Suppose that you are writing a program which performs a variety of geometrical calculations. For example, using the value $\pi$ (3.14) for the calculations. To calculate the circumference and area of a circle with a known radius, you could write

```
circum = 3.14 * (2*radius);
area = 3.14 * (radius) * (radius);
```

If, however, you define a symbolic constant with the name PI and assign it the value 3.14, you would write the name PI and the value 3.14 as shown below

```
#define PI 3.14
circum = PI * (2*radius);
area = PI * (radius) * (radius);
```

Some valid examples of symbolic constant definitions are :

```
#define TAXRATE 0.55
#define TRUE 1
#define FALSE 0
```

**CHECK YOUR PROGRESS**

9. What is an identifiers ?
10. What is a keyword in C ?
11. What are the types of constants ?
12. What is the built in data types in C ?
13. What is a variable ?

## 1.13 LET US SUM UP

A computer program is a set of instructions that directs the computer to perform some specific task.
Characteristics of a good program are - accurate, efficiency, reliability, portability, robustness etc.
The program development cycle consists of the phases : *problem definition, analysis, design phase, testing the algorithm for accuracy, coding, testing and debugging, implementation and documentation* and the last one is *program maintenance*.

The programming languages can be classified as: machine language, assembly language, high level language and 4GL language.

The compiler, interpreter and assembler are the language translator programs.

A linker is a program that attached or links many modules of a program.

The loader brings a program residing on disk into the main memory of computer and run it.

*Pseudocode* is a program-planning tool that allows programmers to plan program logic by writing program instructions in an ordinary natural language, such as English.

The term *algorithm* refers to the logic of a program. It is a step-by-step description of how to arrive at a solution to a given problem.

Pictorial representation to depict clearly the flow of control to arrive at the solution of a problem is called flowchart.

The programming techniques used for designing a program are: *top-down technique, bottom-up technique, unstructured technique, structured technique and modular programming technique*.

C is a general purpose, high-level programming language developed by **Kerningham** and **Ritchie** at AT & T Bell Labs.

C program logic is a combination of statements. Statements are always found between { } braces called the body of a function. Each statement performs a set of operations. Simple statements are terminated by semicolon ' ;'.
Every C program is required to have a special function called *main*. This function is the entry point of the program.

Identifiers are the name given to the various program elements - variables, functions, arrays etc.

C character set includes uppercase and lowercase alphabets, digits and several special characters. Altogether there are 93 valid characters allowed in C.

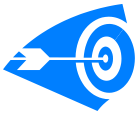
There are 32 keywords (reserved words) in C. They cannot be used as variable names.

A C variable is an entity whose value may vary during program execution. C makes it compulsory to declare the type of any variable name that a programmer wishes to use in a program before using it. The basic data type that can be used for such declaration are *int, float, double* and *char.*

Symbolic constants are generally defined at the beginning of a program.

## 1.14 FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.



## 1.15 ANSWERS TO CHECK YOUR PROGRESS

1. a) T, b) F, c) F, d) T, e) T

2. a) F, b) T, c) T, d) F, e) F

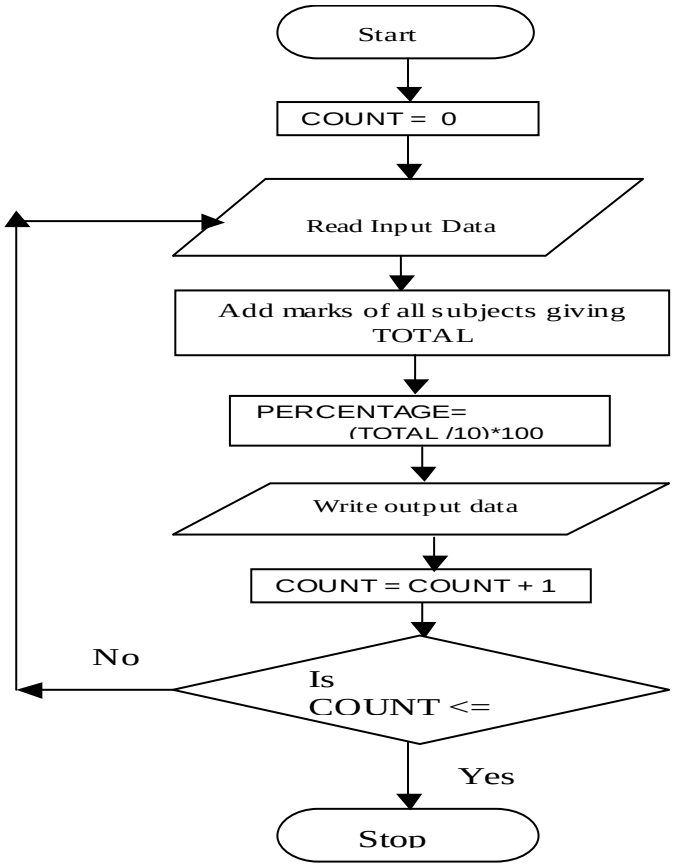
3. A **compiler** is a computer program (or set of programs) that translates text written in a computer language (the *source language*) into another computer language (the *target language*).

A program that is written in a high level language must be translated into machine language before it can be executed. This is known as compilation process.

4. The language in which a programmer writes programs is called source language. It may be a high level language or an assembly language. A program written in a source language is called a source program.

The language in which the computer works is called object language or machine language. When a source program is converted into machine language by an assembler or compiler it is known as an object program. In other words, a machine language program ready for execution is called an object program.

5.




6. In a Fibonacci sequence, the previous two numbers are added to generate the next Fibonacci number.

f1 = 1 lst number

f2 = 2nd number

f3 = f1 + f2 = 1+2 = 3

f4 = f2 + f3 = 2+3 = 5

f5 = f3 + f4 = 3+5 = 8, and so on.

| f1 | f2 | f3 | f4 | f5 | f6 | f7 ..... |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 8 | 13 | 21 ..... |

To get next fibonacci number, we have to do sum of previous two numbers in the series.

Algorithm :

1. Assign sum =0, A=0, B=1, i=1
2. Get the number of terms upto which you want to generate the

Fibonacci number, i.e. n.

3. Add A and B to get the next Fibonacci number
4. Assign the value of B to A i.e. A=B.
5. Assign the value of sum to B i.e. B = sum
6. Write the value of sum to get next Fibonacci number in the series.
7. Increment i with 1 i.e. i = i+1 and repeat step 3,4,5,6 with the last value of i = n (n is the number of terms up to which we want to generate Fibonacci number series.)
8. Stop

7. In structured programming a program is broken down into small independent task and each task has its own functionality and perform specific part of the actual processing. These task are developed independently without the help of the other. When these task are completed, they are combined together to solve a whole program.

8. In computer programming, particularly in the C and C++ programming languages, a header file or include file is a file, usually in the form of source code, that is automatically included in another source file by the compiler. Typically, header files are included via compiler directives at the beginning (or *head*) of the other source file.

The **C standard library** (also known as **libc**) is a now-standardized collection of header files and library routines used to implement common operations, such as input/output and string handling, in the C programming language. Unlike other languages such as COBOL, Fortran, and PL/I, C does not include built-in keywords for these tasks, so nearly all C programs rely on the standard library to function.

9. Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consist of letters and digits in any order except that the first character must be a letter.

10. Keywords are also called the reserved words in C. They have specific meaning to compiler. These words should not be used for naming any other variables.

11. There are two broad categories of constant in C, ***literal constant*** and ***symbolic constant.***

12. The built in data types are *integer, character, float, double*.

13. A variable is an identifier that is used to represent a single data item i.e. a numerical quantity or a character constant. A given variable can be assigned different data items at various place within thin a program.

## 1.16 MODEL QUESTIONS

1. What is a program ? Explain the characteristics of a good program.
2. Describe the various stages of program development.
3. Write down the characteristics of high level language ?
4. Write the difference:
   a) Machine language and Assembly language
   b) High level language and Machine language
   c) Compiler and Interpreter
   d) Linker and Loader
   e) Compiler and Assembler
   f) Pseudocode and Algorithm
5. What do you understand by compilation and execution of a program.
6. What is an algorithm ? Why is it necessary to write an algorithm before program coding ?
7. How flowchart is more effective than algorithm ? Explain with algrithm.
8. What do you mean by programming technique ? Compare modular and structured programming techniques giving example.
9. Give the merits and demerits of modular and structured programming technique.
10. What are the major differences between compilation and

interpretation ? Which process does take more time on repeated processing and why ?

11. Write down a few characteristics of 'C' language.
12. Write an algorithm to find the area of a triangle.
13. Write an algorithm to find the sum of a set of number.
14. Write an algorithm to test whether the given number is a prime number.
15. Write an algorithm to find the factorial of a given number.
16. What is the difference between a keyword and an identifier ?
17. List the rules of naming an identifier in C ?
18. Name and describe the four basic data types in C ?
19. What is a variable ? How can variables be characterized ?
20. Draw a flowchart to print the sum of numbers between 1 and the entered number. For example if you enter 5, then it will find the sum of 1+2+3+4+5 = 15
21. Draw a flowchart to input three numbers and print the largest number.
22. Write an algorithm that reads a year and determine whether it is a leap year or not.
23. Write an algorithm to sort an array in the descending order.

# UNIT- 2 OPERATORS AND EXPRESSIONS

## UNIT STRUCTURE



## 2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define operators and operands
- define and use different types of operators like arithmetic, logical, relational, assignment, conditional, bitwise and special operators
- learn about the order of precedence among operators and the direction in which each associates.
- use expession in programming
- perform type conversion to get the correct result in expression

## 2.2 INTRODUCTION

In our previous unit we have learnt to use variables, constants, data types in programming. With the help of operators, these variables, constants and other elements can be combined to form expressions.

In this unit we will discuss various operators supported by C

language such as Arithmetic operators, Relational operators, Logical operators, Assignment operators, Increments, Decrement operators, Conditional operators, Bitwise operators, Special operators etc. Besides, we will learn to carry out type conversion.

## 2.3 OPERATORS

***Operators*** are special symbols which instruct the compiler to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. The data items that operators act upon are called ***operands***. Operators are used with operands to build expressions. Some operators require two operands, while other act upon only one operand.

C includes a large number of operators which fall into different categories. These are:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increments and Decrement Operators
- Conditional Operators
- Bitwise Operators
- Special Operators

### 2.3.1 Arithmetic Operators

There are five main arithmetic operators in C language. They are '+' for additions, '-' for subtraction, '*' for multiplication, '/' for division and '%' for remainder after integer division. This '%' operator is also known as *modulus* operator. The operators +, -, * and / all work the same way as they do in other languages. These operators can operate on any built-in data types allowed in C. Some uses of arithmetic operators are :

```
x + y
x - y
- x + y
a * b + c
- a * b
```

Here, ***a, b, c, x*** and ***y*** are *operands*. The modulus (%) operator produces the remainder of an integer division. For example :

```
5 / 2 = 2      (Division)
```

where as `5 % 2 = 1` (Modulo Division or modulus)

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be *integer quantities*, *floating-point quantities* or *characters* (character constants represent integer values, as determined by the computer's character set). Modulus cannot be used with floating-point numbers. It requires that both operands be integers and the second operand be nonzero.

Division of one integer quantity by another is referred to as *interger division*. The result of this operation will be a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a flaoting-point quotient.

Let us consider the following expression.

x = 100+ 2/4;

What will be the actual value of x ?

Is it 100 + 0.5 =100.5

or 102 / 4 = 25.5

Since / has precedence over +, the expression will be evaluated as

100 + 0.5 = 100.5

To avoid ambiguity, there are defined presedence rules for operators in C language. The concept of precedence will be discussed later in this unit.

**Integer Arithmetic**

When an arithmetic operation is performed on two whole numbers or integers then such an operation is called ***integer arithmetic***. It always gives an integer as the result. Let, x = 5 and y = 2 be two integer numbers. Then the integer arithmetic leads to the following results:

| | |
|---|---|
| x + y = 5 + 2 = 7 | (Addition) |
| x – y = 5 - 2 = 3 | (Subtraction) |
| x * y = 5 * 2 = 10 | (Multiplication) |
| x / y = 5 / 2 = 2 | (Division) |
| x % y = 5 % 2 = 1 | (Modulus) |

In integer division the fractional part is truncated. *Division* gives the quotient, whereas *modulus* gives the remainder of division. Following program is an example to illustrate the above operations.

**Program1**: Summation, subtraction, multiplication, division and modulo division of two integer numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int n1, n2, sum, sub, mul, div, mod;
     clrscr( );
    scanf (“%d %d”, &n1, &n2);          //inputs the operands
     sum = n1+n2;
    printf(“\n The sum is = %d”, sum);    //display the output
    sub = n1-n2;
    printf(“\n The difference is = %d”, sub);
    mul = n1*n2;
    printf(“\n The product is = %d”, mul);
    div = n1/n2;
    printf(“\n The division is = %d”, div);
     mod = n1%n2;
    printf(“\n The modulus is = %d”, mod);
     getch( );
}
```

If we enter 5 for n1 and 2 for n2, then the output of the above program will be :

```
The sum is = 7
The difference is =  3
The product is =  10
The division is = 2
The modulus is = 1
```

**Floating point arithmetic**

When an arithmetic operation is performed on two real numbers or fractional numbers, such an operation is called ***floating point arithmetic***. The floating point results can be truncated according to the properties requirement. The modulus operator is not applicable for fractional numbers. Let us consider two operands x and y with floating point values 15.0 and 2.0 respectively. Then,

```
x + y = 15.0 +2.0 = 17.0
x – y = 15.0 - 2.0 = 13.0
x * y = 15.0 * 2.0 = 30.0
x / y = 15.0 / 2.0 = 7.5
```

**Mixed mode arithmetic**

When one of the operands is real and the other is an integer and if the arithmetic operation is carried out on these two operands, then it is called as mixed mode arithmetic. If any one operand is of real type then the result will always be real, thus 15 / 10.0 = 1.5

## 2.3.2 Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. Two variables of same type may have a relationship between them. They can be equal or one can be greater than the other or less than the other. We can check this by using ***relational operators***. While checking, the outcome may be *true* or *false*. *"True"* is represented as 1 and *"False"* is represented as 0. It is also by convention that any non-zero value is considered as 1(true) and zero value is considered as 0 (false). For example, we may compare the age of two persons, marks of students, salary of persons, or the price of two items, and so on.

There are four relational operators in C. They are:

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

All these operators fall within the same precedence group, which is lower than the arithmetic and unary operators. The associativity of these operators is left to right. There are two *equality operators* associated with the relational operators. They are:

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

For checking equality the double equal sign is used, which is different from other programming languages. The statement ***a == b*** checks whether ***a*** is equal to ***b*** or not. If they are equal, the output will be *true*; otherwise , it will be *false*. The statement ***a = b*** assigns the value of ***b*** to ***a***. For example, if ***b = 10***, then ***a*** is also assigns the value of ***10***.
>, >=, <, <= have precedence over == and !=. Again, the arithmetic operators +, -, *, / have precedence over relational and logical operators. Therefore, in the following statement :

a - 4 > 6

a - 4 will be evaluated first and only then the relation will be checked. So, there is no need to enclose a - 4 within parenthesis.
A simple relational expression contains only one relational operator and takes the following form:

exp1 ***relational operator*** exp2

where *exp1* and *exp2* are expressions, which may be simple constants, variables or combination of them. Some examples of relational expressions and their evaluated values are listed below:

| | |
|---|---|
| 4.5 <= 12 | TRUE |
| -5 > 0 | FALSE |
| 10 < 8 + 5 | TRUE |
| 5 == 2 | FALSE |
| 6! = 2 | TRUE |

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. Relational expressions are used in decision making statements of C language such as ***if***, ***while*** and ***for*** statements to decide the course of action of a running program. We shall learn about ***if, while*** and ***for*** statement very soon in our next unit.

## 2.3.3 Logical Operators

Logical operators compare or evaluate logical and relational expressions. C language has the following logical operators:

**&&** denoting **Logical AND**
**||** denoting **Logical OR**
**!** denoting **Logical NOT**

The logical AND and Logical OR operators are used when we want to test more than one condition and make decisions.

**Logical AND (&&)**

The logical AND operator is used for evaluating two conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator are true, then the whole compound expression is true. For example:

**a > b && x = = 8**

The expression to the left is a > b and that on the right is x == 8. The whole expression is true only if both expressions are true i.e., if **a** is greater than **b** and **x** is equal to **8**.

**Logical OR (||)**

The logical OR is used to combine two expressions and the condition evaluates to true if any one of the two expressions is true. For example:

**a < m || a < n**

The expression evaluates to true if any one of the expressions **a<m** and **a<n** is true or if both of them are true. It evaluates to true if **a** is less than either **m** or **n** and when **a** is less than both **m** and **n.**

**Logical NOT (!)**

The logical NOT operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words, it just reverses the value of the expression.

For example, ! (x >= y)

This NOT expression evaluates to true only if the value of **x** is neither greater than nor equal to **y** .

## 2.3.4 Assignment Operators

In C language, there are several different assignment operators. The most commonly used assignment operator is =. The assignment operator(=) evaluates the expression on the right of the operator and substitutes it to the value or variable on the left of the operand. For example:

x = a + b ;

In the above statement, the value of a + b is evaluated and substituted to the variable x. Let us consider the statement x = x + 1; This will have the effect of incrementing the value of x by 1. This can also be written as x + = 1;

The commonly used *shorthand assignment operators* are as follows:

| | | |
|---|---|---|
| a = a + 1 | is same as | a += 1 |
| a = a – 1 | is same as | a -= 1 |
| a = a * (n+1) | is same as | a *= (n+1) |
| a = a / (n+1) | is same as | a /= (n+1) |

a = a % b is same as a %= b

The assignment operators =, + =, - =, * =, / =, % = , have the same precedence than the arithmetic operators. Therefore, the arithmetic operations will be carried out first before they are used to assign the values. C language allows multiple assignments in the following form:

**identifier1 = identifier2 = identifier3 = .....= expression**

For example, a = b = c = 50;

The assignment operator = and the equality operator == are distinctly different. The assignment operator is to assign a value to an identifier, whereas the equality operator is used to determine if two expressions have the same value.

If the two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left. The entire assignment expression will then be of this same data type. Under some circumstances, this automatic type conversion can result in an alteration of the data being assigned. For example:

- A floating-point value may be truncated if assigned to an inte ger identifier. For example,

```
int n;
n = 5.5;
```

The above expression will cause the integer 5 to be assigned to i.

- A double-precision value may be rounded if assigned to an integer identifier.
- An integer quantity may be altered if assigned to a shorter integer identifier or to a character identifier.

**Example**: Let us consider i , j are two integer variables and p, q are floating-point variables whose values are

```
i = 8;
j = 5;
p = 4.5;
q = -2.75;
```

Several assignments that make use of these variables are shown below:

| Expression | Shorthand expression | Final value |
|---|---|---|
| i = i + 2; | i += 2; | 10 |
| j = j * ( i - 2 ); | j * = ( i - 2 ) ; | 30 |
| p = p / 2 ; | p /= 2; | 2.25 |
| i = i % ( j - 2 ); | i %= ( j - 2); | 2 |
| p = p - q ; | p - = q; | 7.25 |

**Program 2:** Calculate the sum and average of five numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
      float a,b,c,d,e,sum,avg;
       clrscr( );
      printf("Enter the five numbers:\n ");
      scanf("%f%f%f%f%f ", &a,&b,&c,&d,&e);
       sum=a+b+c+d+e;
      avg=sum/5.0;
       printf("\n\nSum is = %f ",sum);
      printf("\nAverage is = %f ",avg);
       getch( );
}
```

If we enter 4,10,12, 3 and 6, then the output of the above program will be :

```
Enter the five numbers:
4       10      12      3       6

Sum is = 35.00
Average is = 7.00
```

## EXERCISE

Q. What is the output of the following program?

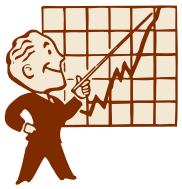
```
#include<stdio.h>
void main()
  {
        int i=15, j=4, m, n;
         m = i > 9;
         n = j > 2 &&  j  != 2;
         printf("m=%d n=%d", m,n);
  }
```

## LET US KNOW

**Unary Operators**

The operator that acts upon a single operand to produce a new value is called ***unary operator***. Unary operators usually precede their single operands, though some unary operators are written after their operands. Unary minus operation is distinctly different from the arithmetic operator which denotes subtraction (-). The subtraction operator requires two separate operands. All unary operators are of equal precedence and have right-to-left associativity. Following are some examples of the use of unary minus operation:

```
-145        // unary minus is followed by an integer constant
-0.5        //unary minus is followed by an floting-point constant
- a                //unary minus is followed by a variable 'a'
-5 *(a + b) //unary minus is followed by an arithmetic expresion
```



## CHECK YOUR PROGRESS

1. Choose the correct option:

(i) The shorthand expression for x = x + 10 is:

a) x += 10; b) +x = 10; c) x =+ 10; d) x = 10+;

(ii) What is the value of *sum* for the expression

sum = 5 + 3 * 4 - 1 % 3;

a) 31 b) 8 c) 7 d) 16

(iii) The expression i=30*10+27 evaluates to

a) 327 b) -327 c) 810 d) 0

2. State whether the following expressions are true or false.

(i) The modulus operators % can be used only with integers.

(ii) The modulo division operator produces the remainder of an integer division

(iii) 10 % 3 yields a result of 3.

(iv) Unary operator requires more than one operands.

(v) If both the expressions to the left and to the right of the && operator is true, then the whole compound expression is true.

## 2.3.5 Increment and Decrement Operators

C language contains two increment and decrement operators which are present in postfix and prefix forms. Both forms are used to increment or decrement the appropriate variables. The increment and decrement operators are one of the unary operators which are very useful in C language.

***Pointer:***
*A pointer is a variable which holds the address of another variable*

- **Increment operator ++**

The increment operator (++) adds 1 to the value of an operand or if the operand is a pointer then increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. We can put ++ before or after the operand. These operators are used in a program as follows:

```
+ + i ;    (prefix form)
or  i + +;   (postfix form)
```

The statement + + i; is equivalent to i = i + 1; or i + = 1;

In the above statements, *i* is an integer type variable. ++ i and i ++ means the same thing when they form statements independently. But they behave differently when they are used in expressions on right-hand side of an assignment statement. If ++ appears before the operand (***prefix*** form), the operand is incremented first and then used in the expression. If we put ++ after the operand (***postfix*** form), the value of the operand is used in the expression before the operand is incremented. Let us consider the following statements:

```
i = 5;
j = ++i;     // pre increment
printf("%d%d", i, j);
```

In this case, the value of j and i would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. Thus, 1 is added to i and the value of i becomes 6. Then this incremented value of i is assigned to j and the value of j also becomes 6. If we rewrite the above statements as

```
i = 5;
j = i++;    // post increment
```

then the value of j would be 5 and i would be 6. This is because a postfix operator first assigns the value to the variable on the left and then increments the operands. Thus, 5 is first assigned to j and then

i is incremented by 1.

- **Decrement operator - -**

The decrement operator (- -) subtracts 1 from the value of a operand or if the operand is a pointer, it decreases the operand by the size of the object to whcih it points. The operand reveives the result of the decrement operation.

Like increment operator, decrement operator - - can be put before or after the operand. If it appears before the operand, the operand is decremented and the decremented value is used in the expression. But if - - appears after the operand then the current value of the operand is first used in the expression and then the operand is decremented.

- - i ; (prefix form)

or i - - ; (postfix form)

The statement - - i; is equivalent to i = i -1; or i - = 1;

**Rules for + + and - - Operators**

Increment and decrement operators are unary operators and they require single variable as their operand.

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

## 2.3.6 Conditional Operator

The conditional operator is also termed as *ternary operator* and is denoted by (? :) . The syntax for the conditional operator is as follows:

**expression1 ? expression2 : expression3**

When evaluating a conditional expression, **expression1** is evaluated first. If **expression1** is true (i.e., if its value is nonzero), then **expression2** is evaluated and this becomes the value of the expres-

sion. However, if **expression1** is false (i.e., if its value is zero) , **expression3** is evaluated and this becomes the value of the conditional expression. Only one of the expressions is evaluated.
For example :

```
a = 10;
b = 15;
x = (a > b) ? a : b ;
```

Here **x** will be assigned to the value of **b**. The condition follows that the expression is false; therefore **b** is assigned to **x**.

**Program4:** Program to illustrate the use of conditional operator.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int age;
     clrscr( );
    printf("Enter your age in years: ");
    scanf("%d",&age);
   (age>=18)? printf("\nYou can vote\n") : printf("You can't vote");
     getch( );
}
```

**Output :**
```
Enter your age in years:
26
You should vote
```

If we run the program again and enter *age=15,* then the output will be:

```
Enter your age in years:
15
You cannot vote
```

**Program5:** Program for finding the larger value of two given values using conditional operator

```
#include<stdio.h>
void main()
{    int i,j, large;
   printf ("Enter 2 integers : "); //ask the user to input 2 numbers
   scanf("%d %d",&i, &j);
   large = i > j ? i : j;          //evaluation using conditional operator
   printf("The largest of two numbers is %d \n", large);
}
```

**Output :** Enter 2 integers : 14 25
The largest of two numbers is 25

**Program6:** Conversion of centigrate to fahrenheit and vice-versa

```
#include<iostream.h>
void main()
{
    float c, f;
    printf("Enter temperature in celcius: );
    scanf("%f", &c);
    f=1.8 * c+32;
   printf("\nEquivalent Fehrenheit will be: %f",f);
    printf("\nEnter temperature in Fehrenheit: );
    scanf("%f", &f);
    c=(f-32)/1.8;
   printf("\nEquivalent Celsius will be: %f",c);
}
```

## 2.3.7 Bitwise Operators

The bitwise operatos are used for testing, complementing or shifting bits to the right or left. A bitwise operator operates on each bit of data. Bitwise operators may not be applied to a float or double. The bitwise operators with their meaning are listed below:

| | |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise Exclusive |
| << | Shift left |
| >> | Shift right |

**Bitwise Logical Operators :**

The logical bitwise operators are similar to the Boolean or Logical operators, except that they operate on every bit in the operand(s). For instance, the bitwise AND operator (&) compares each bit of the left operand to the corresponding bit in the right hand operand. If both bits are 1, a 1 is placed at that bit position in the result. Otherwise, a 0 is placed at that bit position.

**Bitwise AND (&) Operator :**

The bitwise AND operator performs logical operations on a bit-by-bit level using the following truth table:

| Bit x of operator1 | Bit x of of operator2 | Bit x of result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Truth Table for the bitwise AND (&) operator

Let us consider the following program segment for understanding AND(&) operation.

```
void main( )
{    unsigned int a = 60;          // a= 60 = 0011 1100
     unsigned int b = 13;          // b= 13 = 0000 1101
     unsigned int c = 0;
      c = a & b;                    //c= 12 = 0000 1100
      printf("%d",c);
}
```

The output will be 12

**Bitwise OR (|) operator :**

The bitwise OR operator performs logical operations on a bit-by-bit level using the following truth table:

| Bit x of operator1 | Bit x of operator2 | Bit x of result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Truth Table for the bitwise OR (|) operator

The bitwise OR operator (|) places a 1 in the corresponding value's bit position if either operand has a bit **set** (i.e.,1) at the position. Bitwise OR(|) operation can be understood with the following example:

```
void main( )
{   unsigned int a = 60;          // 60 = 0011 1100
    unsigned int b = 13;          // 13 = 0000 1101
    unsigned int c = 0;
    c = a | b;                    // 61 = 0011 1101
 }
```

**Bitwise exclusive OR (^) :**

The bitwise exclusive OR(XOR) operator performs logical operations on a bit-by-bit level using the following truth table:

| Bit x of operator1 | Bit x of operator2 | Bit x of result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Truth Table for the exclusive OR(^)

The bitwise exclusive OR(^) operator sets a bit in the resulting value's bit position if either operand (but not both) has a bit **set** (i.e.,1)at the position. Bitwise exclusive OR(^) operation can be understood with the following example:

```
void main( )
{   unsigned int a = 60;     // 60 = 0011 1100
    unsigned int b = 13;     // 13 = 0000 1101
    unsigned int c = 0;
    c = a ^ b;               // 49 = 0011 0001
 }
```

**Bitwise Complement (~)**

The bitwise complement operator (~) performs logical operations on a bit-by-bit level using the following truth table:

| bit x of op2 | result |
|---|---|
| 0 | 0 |
| 0 | 1 |

Truth table for the ~, Bitwise Complement

The bitwise complement operator (~) reverses each bit in the operand.

**Bitwise Shift Operators :**

C provides two bitwise shift operators, bitwise left shift (**<<**) and bitwise right shift (**>>**), for shifting bits left or right by an integral number of positions in integral data. Both of these operators are binary, and the left operand is the integral data whose bits are to be shifted, and the right operand, called the shift count, specifies the number of positions by which bits need shifting. The shift count must be nonnegative and less than the number of bits required to represent data of the type of the left operand.

**Left-Shift (<<) operator**

The left shift operator shift bits to the left. As bits are shifted toward high-order positions, **0** bits enter the low-order positions. Bits shifted out through the high-order position are lost. For example, let us consider the following declaration:

unsigned int Z = 5;

and Z in binary is 00000000 00000101 when 16 bits are used to store integer values.

Now if we apply left-shift, then

Z << 1 is 00000000 00001010 or 10 decimal

and Z << 15 is 10000000 00000000 or 32768 decimal.

Left-Shift is useful when we want to MULTIPLY an integer (not floating point numbers) by a power of 2. The operator, takes 2 operands like this:

a << b

This expression returns the value of a multiplied by 2 to the power of b.

For example, let us consider 4 << 2. In binary, 4 is 100. Adding 2 zeros to the end gives 10000, which is 16, i.e., $4*2^2 = 4*4 = 16$.

Similarly, 4 << 3 can be evaluated by adding 3 zeros to get 100000, which is $4*2^3 = 4*8 = 32$.

Shifting once to the left multiplies the number by 2.Multiple shifts of 1 to the left results in multiplying the number by 2 over and over again. In other words, multiplying by a power of 2. Some examples are:

$5 << 3 = 5*2^3 = 5*8 = 40$
$8 << 4 = 8*2^4 = 8*16 = 128$
$1 << 2 = 1*2^2 = 1*4 = 4$

**Right-Shift (>>) operator**

The right shift operator shifts bits to the right. As bits are shifted towards low-order positions, **0** bits enter the high-order positions, if the data is unsigned. If the data is signed and the sign bit is **0**, then **0** bits also enter the high- order positions. However, if the sign bit is **1**, the bits entering high-order positions are implementation-dependent. On some machines **1**s, and on others **0**s, are shifted in. The former type of operation is known as the arithmetic right shift, and the latter type the logical right shift. For example,

unsigned int Z = 40960;

and Z in binary 16-bit format is 10100000 00000000

Now, if we apply right-shift, then

Z >> 1 is 01010000 00000000 or 20480 decimal

and Z >> 15 is 00000000 00000001 or 1 decimal

In the second example, the **1** originally in the fourteenth bit position has dropped off. Another right shift will drop off the **1** in the first bit position, and **Z** will become zero. Bitwise Right-Shift does the opposite, and takes away bits on the right.

## 2.3.8 Other Operators

There are some other useful operators supported by C language. These are: *comma* operator, *sizeof* operator, *member selection operator* (. and -> ), *pointer operators* (* and &) etc. Here we will discuss *comma* and *sizeof* operators. The other two will be covered in later unit.

**The Comma Operator**

The ***comma*** operator can be used to link related expressions together. The comma allows for the use of multiple expressions to be used where normally only one would be allowed.

The comma operator forces all operations that appear to the left to be fully completed before proceeding to the right of comma. This helps eliminate side effects of the expression evaluation.

```
num1 = num2 + 1, num2 = 2;
```

The comma ensures that **num2** will not be changed to a 2 before **num2** has been added to 1 and the result placed into **num1**. Some examples of comma operator are:

In *for* loops:

```
for (n=1, m=15, n <=m; n++,m++)
```

In *while* loops:

```
while (c=getchar(), c != '15')
```

Exchanging values :

```
temp = x, x = y, y = temp;
```

The concept of loop will be discussed very soon in the next unit.

**Program7:** Swap (interchange) two numbers using a temporary variable.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,temp;
    clrscr();
    printf("\nEnter the two integer numbers:");
    scanf("%d%d",&a,&b);
    printf("\nEntered numbers are.:");
    printf("%d%8d",a,b);
    temp=a,a=b,b=temp;        // comma operator is used
    printf("\n\nSwapped numbers are:%d%8d",a,b);
    getch();
}
```

**Output :** (Suppose we have entered 2 and 4)

```
Enter the two integer numbers.: 2    4
Entered numbers are: 2       4
Swapped numbers are: 4       2
```

**The Sizeof Operator**

The **sizeof** operator returns the physical size, in bytes of the data item for which it is applied. It can be used with any type of data item except bit fields.

When **sizeof** is used on a character field the result returned

is 1 (if a character is stored in one byte). When used on an integer the result returned is the size in bytes of that integer.
For example:

```
`       s = sizeof (sum);
        t = sizeof (long int);
```

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

For example: s = sizeof (sum);
t = sizeof (long int);

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

**Program8:** Program that employs different kinds of operators like arithmetic, increment, conditional and sizeof operators.

```
#include<stdio.h>
#include<conio.h>
void main( )
{     int a, b, c, d,s;
      clrscr( );
      a = 20;
      b = 10;
      c = ++a-b;
      printf (“a = %d, b = %d, c = %d\n”, a,b,c);
      d=b++ + a;
      printf (“a = %d, b = %d, d = %d\n, a,b,d);
      printf (“a / b = %d\n, a / b);
      printf (“a % b = %d\n, a % b);
      printf (“a *= b = %d\n, a *= b);
      printf (“%d\n, (c < d) ? 1 : 0 );
      printf (“%d\n, (c > d) ? 1 : 0 );
      s=sizeof(a);
      printf(“\nSize is: %d bytes”,s);
}
```

**Output :** a=21 b=10 c=11
a=21 b=11 d=32
a/b=1

a%b=10
a*=b=231
1
0
2 bytes

The increment operator ++ works when used in an expression. In the statement c = ++a – b; new value a = 16 is used thus giving value 6 to C. That is **a** is incremented by 1 before using in expression However in the statement d = b++ + a; the old value b = 10 is used in the expression. Here **b** is incremented after it is used in the expres sion.

## EXERCISE

Q. Write a program that reads a floating-point number and then display the right-most digit of the integral part of the number.

## CHECK YOUR PROGRESS

3. Find the output of the following program segment?

(a)
```
void main(){     int x = 50;
                 printf("%d\n",5+ x++);
                 printf("%d\n",5+ ++x);   }
```

(b)
```
void main(){     int x, y;
                 x = 50;
                 y =100;
                 printf("%d\n",x+ y++);
                 printf("%d\n",++y -3);    }
```

(c)
```
void main(){     int s1,s2;
                 char c='A';
                 float f;
                 s1=sizeof(c);
                 s2=sizeof(f);
                 printf("ASCII value of 'A' is %d",c);
printf("\nSize of s1 and s2 in bytes:%d%8d",s1,s2);}
```

4. Find the output of the following C program:

```
void main( )
{    int a,b,c;
     a=b=c=0;
    printf("Initial value of a,b,c :%d%d%d\n",a,b,c);
     a=++b + ++c;
     printf("\na=++b + ++c=%d%d%d\n",a,b,c);
     a= b++ + c++;
     printf("\na=b++  + c++= %d%d%d\n",a,b,c);
     a=++b + c++;
     printf("\na=++b + c++= %d%d%d\n",a,b,c);
     a = b- - + c - -;
     printf("\na=b-- +c --= %d%d%d\n",a,b,c);
}
```

5. Choose the correct option:

(i) If i=6, and j=++i, the the value of j and i will be

(a) i=6,j=6 (b) i=6, j=7 (c)i=7,j=6 (d)i=7,j=7

(ii) If the following variables are set to the values as shown below, then what will be the expression following it?

```
answer=2;
marks=10;
!(("answer<5")&& (marks>2))
```

(a) 1 (b) 0 (c) -1 (d) 2

6. Write a C program to find the area of a triangle when base and height are given.

7. What will be output of the following code?

```
#include<stdio.h>
void main()
{
        int n;
        n = 20;
        printf("\nValue of n : %d", sizeof(n));
        printf("\nSizeof n: %d", sizeof(n));
}
```

## 2.4 PRECEDENCE AND ASSOCIATIVITY

There are two important characteristics of operators which determine how operands group with operators. These are ***precedence*** and ***associativity***.

The operators have an order of precedence among themselves. This order of precedence dictates in what order the operators are evaluated when several operators are together in a statement or expression. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses. Also, with each operator is an associativity factor that tells in what order the operands associated with the operator are to be evaluated. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.

For example, ***operator precedence*** is why the expression 6 + 4 * 3 is calculated as 6 + (4 * 3), giving 18, and not as (6 + 4) *3, giving 30. We say that the multiplication operator (*) has higher precedence than the addition operator (+), so the multiplication must be performed first. ***Operator associativity*** is why the expression 8 - 4 - 2 is calculated as (8 - 4) - 2, giving 2, and and not as 8 - (4 - 2), giving 6. We say that the subtraction operator (-) is left associative, so the left subtraction must be performed first. When we cannot decide by operator precedence alone in which order to calculate an expression, we must use associativity.

The following table lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied. R indicates *Right* and L indicates *Left*.

| **Operator category** | **Operators** | **Associativity** |
|---|---|---|
| Unary Operator | -- ++ ! sizeof(type) | R to L |
| Arithmetic multiply, Divide and remainder | * / % | L to R |
| Arithmetic add,subtract | + - | L to R |
| Relational Operators | < <= > >= | L to R |
| Equality Operators | == != | L to R |
| Logical AND | && | L to R |
| Logical OR | \|\| | L to R |
| Conditional operator | ? : | R to L |
| Assignment operator | = += -+ *= /+ %= | R to L |

Precedence and Associativity of operators

In the following statements, the value 10 is assigned to both ***a*** and ***b*** because of the right-to-left associativity of the = operator. The value of ***c*** is assigned to ***b*** first, and then the value of ***b*** is assigned to ***a***.

```
b = 9;
c = 10;
a = b = c;
```

In the expression

```
a + b * c / d
```

the ***** and **/** operations are performed before **+** because of precedence. ***b*** is multiplied by ***c*** before it is divided by ***d*** because of associativity.

## 2.5 EXPRESSIONS

C Expressions are based on algebra expressions - they are very similar to what we learn in Algebra, but they are not exactly the same. An expression is a combination of variables, constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Here are some examples of expressions:

```
15       // a constant
i        // a variable
i+15     // a variable plus a constant
(m + n) * (x + y)
```

The following program illustrates the effect of presence of parenthesis in expressions.

**Program9:**

```
#include<stdio.h>
#include<conio.h>
void main( )
   float a, b, c x, y, z;
   a = 9;
   b = 12;
   c = 3;
   x = a – b / 3 + c * 2 – 1;
   y = a – b / (3 + c) * (2 – 1);
   z = a – ( b / (3 + c) * 2) – 1;
   printf (“x = %fn”,x);
   printf (“y = %fn”,y);
```

```
    printf ("z = %fn",z);
}
```

**Output**

x = 10.00
y = 7.00
z = 4.00

**Rules for evaluation of expression**

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

## 2.6 TYPE CONVERSION

The **type conversion** or **typecasting** refers to changing an entity of one data type into another i.e., values of one type can be converted to a value of another type. For example, the integer 355 can be converted to the string "355". Again, 355 + 5 will give 400, "355" + "5" will give "3555".

The language C allows programmer to perform typecasting by placing the type name in parentheses and placing this in front of the value. The form of the cast data type is :

**(type) expression**

For example,

```
(float)25       // Gives the float 25.0
(int)5.8        // Gives the int 5
(string)5.8     // Gives the string "5.8"
(float)"4.3"    // Gives the float 4.3
```

Let us consider the case where we want to divide two integers a/b, where the result must be an integer. However, we may want to force the output to be a float type in order to keep the fraction part of the

division. The typecast operator is used in such a case. It will do the conversion without any loss of fractional part of data.

**Program10:**

```
#include<stdio.h>
void main()
{
        int a,b;
        a=3,b=2;
        printf("\n%f",(float)a/b);
}
```

The output of the above program will be 1.500000. This is because data type cast (float) is used to force the type of the result to be of the type float.

From the above it is clear that the usage of typecasting is to make a variable of one type act like another type for one single operation. So by using this ability of typecasting it is possible to create ASCII characters by typecasting integer to its character equivalent. Typecasting is also used in arithmetic operation to get correct result. This is very much needed in case of division when integer gets divided and the remainder is omitted. In order to get correct precision value, one can make use of typecast as shown in example above. Another use of the typecasting is shown in the example below:

For instance:

```
void  main()
{
    int a = 5000, b = 7000 ; long int c = a * b ;
 }
```

Here, two integers are multiplied and the result is truncated and stored in variable c of type long int. But this would not fetch correct result for all. To get a more desired output the code is written as

```
long int c = (long int) a * b;
```

Though typecast has so many uses one must take care about its usage since using typecast in wrong places may cause loss of data like, for instance, truncating a *float* when typecasting to an *int*.

Some conversions are done automatically. For example, in the expression "Hello" + 15 the integer 15 will be automatically converted to the string "15", before the two strings are concatenated, giving the result "Hello15".

## CHECK YOUR PROGRESS

8. State whether the following expressions are true or false.

(i) Conditional operator (? :) has right to left associativity.

(ii) Logical OR operator has right to left associativity

(iii) C permits mixing of constants and variables of different types in an expression.

(iv) Precedence dictates in what order the operators are evalu ated when several operators are together in a statement or expression.

(v) A typecast is used to force a value to be of a particular variable type.

## 2.7 LET US SUM UP

***Operators*** form expressions by joining individual constants, variables,array elements etc. C language includes a large number of operators which fall into different categories. In this unit we have seen how arithmetic operators, assignment operators, unary operators, relational and logical operators, the conditional operators are used to form expressions.The data items on which operators act upon are called ***operands***. Some operators require two operands while others require only one operand. A list of operators with their meaning are given below:

**Arithmetic Operator**

| Operator | Description |
|---|---|
| * | multiplication |
| / | division |
| % | modulo division |
| + | addition |
| - | subtraction |

**Logical Operator**

| Operator | Description |
|---|---|
| ! | NOT |
| && | AND |
| \|\| | OR |

**Relational Operator**

| Operator | Description |
|---|---|
| < | less than |
| > | greater than |
| >= | greater than or equal |
| == | equal to |
| != | not equal |

**Bitwise Operators**

| Operator | Description |
|---|---|
| ~ | One's complement |
| << | Left shift |
| >> | Right shift |
| & | Bitwise AND |
| ^ | Bitwise XOR |

**Assignment Operator**

The Assignment Operator(=) evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression. For example: sum = n1+n2 ;
value of *n1* and *n2* are added and the result is assigned to the variable *sum*.

**Increment and Decrement**

There are two formsof increment and decrement operators. These are:

| Operator | Description | Example |
|---|---|---|
| ++ | increment | a++ (post increment) |
| | | ++a (pre increment) |
| -- | decrement | a-- (post decrement) |
| | | --a (pree decrement) |

**The Conditional Operator**

It works on three values. The conditional operator is used to replace *if-else* logic in some situations. It is a two-symbol operator **?:** with the format:

result = condition ? expression1 :expression2;

**Comma Operator**

We can use the comma operator(,) available in c language, to build a compound expression by putting several expressions inside a set of parentheses. The expressions are evaluated from left to right and the final value is evaluated last.

**sizeof Operator**

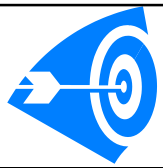
The **sizeof** operator returns the physical size in bytes of the data item for which it is applied. It can be used with any type of data item except bit fields. The general form is: s = sizeof (item );

***Expressions*** in C are syntactically valid combinations of operators amd operands that compute to a value determined by the priority and associativity of the operators.

Converting an expression of a given type into another type is known as ***type-casting*** or ***type conversion***. Type conversions depend on the specified operator and the type of the operand or operators.

## 2.8 FURTHER READINGS

1. Venugopal, K L and Prasad S R : *Mastering C,* Tata McGraw-Hill publication.
2. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
3. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.



## 2.9 ANSWERS TO CHECK YOUR PROGRESS

1.(i) (a) x = x +10 (ii)(d) 16 (iii) (a) 327

2. (i)True (ii) True (iii) False (iv) False (v) True

3. (a) 55 (b)150
99 57
(c) ASCII value of 'A' is 65
Size of s1 and s2 in bytes: 1 4

4. Initial value of a,b,c : 0 0 0
a= ++b + ++c = 2 1 1
a= b++ + c++ = 2 2 2
a= ++b + c++ = 5 3 3
a= b-- + c -- = 6 2 2

5. (i)(d)i=7,j=7 (ii) (b) 0

6. //area of a triangle given the base and height

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float b, h;
     float area;
    printf("Enter the base: ");
    scanf("%f", &b);
    printf("\nEnter the height:");
    scanf("%f", &h);
```

```
        area= b * h / 2;
        printf("\nArea of the triangle is %f", area);
    }
```

7. Value of n is: 20
   Size of n is: 2

8. (i) True (ii)False (iii)True (iv)True (v) True

## 2.10 MODEL QUESTIONS

1.What is an operator ? What are the types of operators that are included in C.
2. What is an operand ? What is the relationship between operator and operand?
3.Describe the three logical operators included in C?
4. Write a C program to compute the surface area and volume of a cube if the side of the cube is taken as input.
5. What is unary operator ? How many operands are associated with a unary operator?
6. What is meant by operator precedence?
7. What is meant by associativity? What is the asslociativity of the arithmetic operators?
8. What will be the output of the following program:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
      printf("The size of char is %d",sizeof(char));
      printf("\nThe size of int is %d",sizeof(int));
      printf("\nThe size of short is %d",sizeof(short));
      printf("\nThe size of float is %d",sizeof(float));
      printf("\nThe size of long is %d",sizeof(long));
      printf("\nThe size of char is %d",sizeof(char));
      printf("The size of double is %d",sizeof(double));
   getch();
}
```

9. List the relational operator and their meaning.
10 . Write programs for computing the volume of a sphere, a cone

and a cylinder. Assume, the dimensions are integers. Use type casting whereever necessary.

11. If a,b,c,d and e are declared using the statement

int a, b, c, d;

What value is assigned to the variablea in the following statement

a = b > c ? c > d ? 12 : d > e ? 13 : 14 : 15 in each of the following cases:

(i) b = 5; c = 15; d = e = 8;

(ii) b = 15; c = 10; d = e = 8;

(iii) b = 15; c = 10; d = e = 20;

(iv) b = c = 9; d = 20; e = 19;

*****

# UNIT-3 DECISION AND CONTROL STRUCTURES

## UNIT STRUCTURE



## 3.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- learn about Input/Output functions
- describe Conditional Statements
- describe Iterative Statements
- define *break, continue* and *goto* statements

## 3.2 INTRODUCTION

We have seen that the C language is accompanied by a collection of library functions, which includes a number of input/output functions. Moreover the instructions were executed in the same order in which they appeared within a program. Each instructions was executed once and only once. But the C language provides the facilities to carry out logical test at some particular point within the program and

repeated execution of a group of instructions. In this unit we will discuss about various input/output library functions and various control statements available in C language.

## 3.3 INPUT/OUTPUT FUNCTIONS

C language simply has no provision for receiving data from any of the input devices (say keyboard, floppy) or for sending data to the output devices (say VDU, floppy etc.). It means that 'C' compiler does not provide any I/O (Input/Output) statements. In programming practice, many programs may require data which needs to be read into the variable names.Moreover, sometimes data stored in variable names need to be shown externally. That is why, in C, there are numerous library functions available for I/O, through which data can be read from or written to files or standard I/O devices. These library functions can be classified into three broad categories:

a. *Console I/O functions* - functions to receive input from keyboard and write output to VDU.
b. *Disk I/O functions* - functions to perform I/O operations on a floppy disk or hard disk.
c. *Port I/O functions* - functions to perform I/O operations on various ports.

We will now deal with the Console I/O functions.Console I/O functions can be further classified as shown in the Table 1.

**Console Input/Output functions**

| Formatted functions | | | Unformatted functions | | |
|---|---|---|---|---|---|
| ***Type*** | ***Input*** | ***Output*** | ***Type*** | ***Input*** | ***Output*** |
| char | scanf() | printf() | char | getch() | putch() |
| | | | | getchar() | putchar() |
| | | | | getche( ) | |
| int | scanf() | printf() | int | - | - |
| float | scanf() | printf() | float | - | - |
| string | scanf() | printf() | string | gets() | puts() |

The basic difference between formatted and unformatted I/O functions is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per

our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points etc., can be controlled using formatted functions.

The library implements a simple model of text input and output. A text consists of a sequence of lines, each ending with a newline character. If the system doesn't operate that way, the library does whatever is necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.

**Unformatted Console I/O Functions :** There are several standard library function available under this category. These functions deals with a single character or with a string of characters. Let us first look at the functions that can handle one character at a time.

**a) Single Character Input - the *getchar* Function :**

Single character can be entered in to the computer using the C library function getchar. This function reads one character from the keyboard after the new-line character is received (when press Enter key). The function does not require any arguments, though a pair of empty parentheses must follow the word *getchar*.

In general terms, getchar function is written as

*character variable = getchar();*

Where *character variable* refers to some previously declared character variable.

e.g.

```
char ch;
ch = getchar ( ) ;
```

***Program 1 : Demonstration of getchar() function***

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

{
 char key;

 printf("\n Type your favourite keyboard character:");
 key=getchar();
 printf("Your favourite character is %c!\n",key);
}
```

RUN:

```
Type your favourite keyboard character: 1
Your favourite character is 1!
```

**b) Single Character Output - the *putchar* Function :**

Single character can be displayed using the C library function putchar. *putchar()*, the opposite of getchar(), is used to put exactly one character on the screen. Putchar requires as an argument the character to put on the screen.
In general the putchar function is written as

*putchar(character variable);*

where *character variable* refers to some previously declared character variable

e.g.

```
char ch;
ch = getchar ( ) ;  /* input a character from kbd*/
putchar (ch) ;      /* display it on the screen */
```

***Program 2 : Demonstration of putchar() function***

```
#include <stdio.h>
void main()
{
    char x = 'A'
    putchar(x);
    putchar('B');
}
```

RUN:

```
AB
```

**c) String Input and String Output Function :**

**gets( ) -** The gets( ) function receives a string from the keyboard. The scanf( ) function has some limitations while receiving a string of characters because the moment a blank character is typed, scanf( ) assumes that the end of the data is being entered. So it is possible to enter only one word string using scanf( ). To enter multiple words in to the string, the gets( ) function can be used. Spaces and tabs are perfectly accepted as part of the string. It is terminated when the

enter key is hit.

In general terms, gets function is written as

**gets(variable name);**

Where *variable name* will be a previously declared variable.

**puts( ) -** The *puts( )* function works exactly opposite to *gets( )* function. It outputs a string to the screen. Puts( ) can output a single string at a time.

In general terms, gets function is written as

**puts(variable name);**

Where *variable name* will be a previously declared variable.

***Program 3 : Demonstration of gets() and puts() function***

```
#include<stdio.h>
void main( )
    {
        char name[40];
        puts("Enter your name");
        gets(name);
        puts("Your name is");
        puts(name);
    }
```

**Formatted Console I/O Functions :** In order to write a user interactive program in C language, we would need input and output functions that are also called routines. The two functions used for this purpose are : **printf()** and **scanf()**

**scanf( ) -** scanf( ) allows us to enter data from the keyboard that will be formatted in a certain way. The general form of scanf( ) statement is as follows:

**scanf(control string, arg1, arg2.....argn);**

Where control string refers to a string containing certain required formatting information, and arg1, arg2,....argn are arguments that represent the individual data items.

Note that we are sending the addresses of variables (addresses are obtained by using & - 'address of' operator) to scanf( ) function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s). Do not include these escape sequences in the format string.

**scanf("%d %f %c",&c,&a,&ch);**

**printf() -** The output function printf() translates internal values to character.

**printf(control string, arg1, arg2, .....argn )**

The format string can contain:

- Characters that are simply printed as they are.
- Conversion specification that begins with a % sign.
- Escape sequences that begins with a \ sign.

Printf() converts, formats, and prints its arguments on the standard output under the control of the format. It returns the number of characters printed. The format string contains two types of objects - ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf(). Each conversion specification begins with % and ends with a conversion character. Between the % and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted arguments will be printed in a field at least as wide as the specified minimum. If necessary, it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period (.) separates the field width from the precision.
- A number, i.e., the precision that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.

If the character after the % is not a conversion specification, the behavior is undefined.

| **Data type** | | **Conversion character** |
|---|---|---|
| Integer | short signed | %d or %I |
| | short unsigned | %u |
| | long signed | %ld |
| | long unsigned | %lu |
| | unsigned hexadecimal | %x |
| | unsigned octal | %o |
| Real | float | %f |
| | double | %lf |
| Characters | signed char | %c |
| | unsigned char | %c |
| string | | %s |

***Program 4 : Demonstration of scanf() and printf() function***

```
#include <stdio.h>
void main()
        {
              int i,j;
              printf("Please type in 2 numbers: ");
              scanf("%d %d",&i,&j);
             printf("you typed %d and %d\n",i,j);
        }
```

## CHECK YOUR PROGRESS

1. A C program contains the following statements

```
#include<stdio.h>
char a, b, c;
```

a) Write appropriate getchar statements that will allow values for a, b, c to be entered into the computer.

b) Write appropriate putchar statement that will allow the current value of a, b, c to be written out of the computer.

2. A C program contains the following statements

```
#include<stdio.h>
int i, j, k;
```

Write a printf function for each of the following groups of variable or expressions. Assume all variable represent decimal integers.

a) i, j and k

b) (i+j), (i-k)

c) sqrt (i+j), abs (i-k)

3. Write appropriate scanf function for the above problem to enter automatic values for i, j and k assuming

a) the value for i, j and k will be decimal integers.

b) the value for i will be decimal integer, j an octal integer and k a hexadecimal integer.

## 3.4 CONDITIONAL STATEMENT

When programming, you will ask the computer to check various kinds of situations and to act accordingly. The computer performs various comparisons of various kinds of statements. These statements come either from you or from the computer itself, while it is processing internal assignments.

In this section we will discuss about some of the conditional statemet used in C language i.e. *if* Statement, *if else* Statement, *Nested if-else* Statement, *switch* Statement, Conditional Operator Statement etc.

### 3.4.1 *if* Statement

This is the most popular decision making statement. First of all it

checks the *test condition* and then, depending on the result of the test condition it transfers the control to a particular statement or a block of statements.

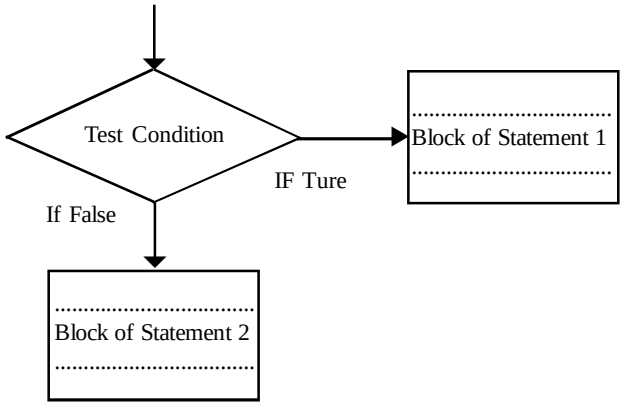



Fig 3.1: Control transfer in if statements

Depending on the complexity of conditions to be tested if statement may be implemented in 4 different ways.The syntax of simple **if** statement is –

```
If(test condition)
{
    - - - - - - - - - - - - - -
    block of statement
    - - - - - - - - - - - - - -
}
```

Example 1: Write a C program to check whether the entered number is positive or negative?

Solution:

```
#include<stdio.h>

void main()
    {
        int a ;
        printf ( " Enter the number " ) ;
        scanf ( " %d " , &a ) ;
        if( a == 0 )
        printf ( "The Number is Zero ") ;
        if ( a > 0 )
        printf (" The Number is Positive " ) ;
        if ( a < 0 )
        printf ( "The number is negative " );
    }
```

Output:

```
Enter the number  3
The Number is Positive
```

## 3.4.2 *if - else* Statement

This is a bi-directional condition control statement. This type of statement is used to test the condition and take one of two possible actions. If the test condition is evaluated and found to be true then *block of statement 1* will executed, otherwise *block of statement 2* will executed. Syntax is –

```
If (test condition)
   {
     block of statement 1
   }

 else
    {
      block of statement 2
    }
```

Example 3.2 Write a C program to print the largest of two given numbers?

```
#include<stdio.h>

void main()
  {
     int a , b ;
     printf ( " Enter the First number " ) ;
      scanf ( " %d " , &a ) ;
      printf ( " Enter the Second number " ) ;
      scanf ( " %d " , &b ) ;

     if(a>b)
          printf ( " Largest Number is = %d" , a ) ;
                /* This statement will executed if the conditional
                statement is true i.e.  if the value of a is greater than
                the value of b */


     else
          printf ( " Largest Number is=%d" , b ) ;
                /* This statement will executed if the conditional
                statement is false i.e  if the value of a is less than the
                value of b */
  }
```

Output :

```
Enter the First number 12
Enter the Second number 20
Largest  Number is = 20
```

## 3.4.3 Nested if - else Statement

In certain cases we may use one if else structure within another if else. This is known as nested if else structure. The Syntax is as follows.

```
if(test condition 1)
  {
   if(test condition 2)
    {
        block of statement 1
    }
   else
   {
      block of statement 2
   }
 }
else
 {
    block of statement 3
 }
```

Example 3.3: Example 3.1 can be rewritten as following ways-

```
#include<stdio.h>
void main()
{
        int a ;
        printf ( " Enter the number: " ) ;
        scanf ( " %d " , &a ) ;
          if( a == 0 )
            printf ( "The number is Zero ") ;
          else
            if ( a > 0 )
              printf (" The number is Positive " ) ;
            else
           printf ( "The number is Negative " );
}
```

Output : Enter the number : 5

The number is Positive

### 3.4.4 *switch* Statement

The main disadvantage of if …else statement is that they are complex to understand,read and debug. If you have a complex set of choices to make, the switch statement is the more powerful alternative. “Pick the matching value and act” is the working strategy of switch statement, rather checking the conditions of nested if…else ladder. The syntax is –

```
switch ( expression )
        {
            case label1 :
                    statement(s) ;
                    break ;
            case label2 :
                    statement(s) ;
                    break ;
            -   - - - - - - - - - - - - -
            -   - - - - - - - - - - - - -
            default:
             statement(s);
                    break ;
        }
```

The switch statement check the value of expression against the list of case labels and when a match is found, the block of statements associated with that case is executed. The break statement at the end of each block signals the end of a particular **case** and causes immediate exit from the switch statement. If the expression does not match any of the case labels then it will execute the default case. Let us consider an example –

Example 3.4: Write a program to convert number to words. For example if you enter 5 the output will be “Five”.

```
#include<stdio.h>
void main()
 {
    int x;
   printf(“Enter a number less than 10 : ” );
    scanf(“%d”,&x) ;
```

```
        switch(x)
        {
        case 1: printf(“ One ”) ;
        break;
        case 2: printf(“ Two ”) ;
        break ;
        case 3: printf(“ Three ”) ;
        break ;
        case 4: printf(“ Four ”) ;
        break ;
        case 5: printf(“ Five ”) ;
        break ;
        case 6: printf(“ Six ”) ;
        break ;
        case 7: printf(“ Seven ”) ;
        break ;
        case 8: printf(“ Eight ”) ;
        break ;
        case 9: printf(“ Nine ”) ;
        break ;
        case 10: printf(“ Ten ”) ;
        break ;
        default : printf( “ Out of range ” ) ;
      }
}
```

Output: Enter a number less than 10 : 5
Five

## 3.4.5 Conditional Operator Statement

C has a special type of operator which has the two way decision making capability. This operator is known as ternary conditional operator. The syntax is -

Conditional expression ? statement_1 : statement_2

The conditional expression is evaluated first. If the result is true, statement_1 is evaluated and is returned as the value of the conditional expression, otherwise statement_2 is evaluated and its value is returned. Let us again consider example 3.2

```
        void main()
              {
                 int a , b ;
                 printf ( “ Enter the First number ” ) ;
```

```
        scanf ( " %d " , &a ) ;
        printf ( " Enter the Second number " ) ;
        scanf ( " %d " , &b ) ;
        if(a>b)
          printf ( " Largest Number is = %d " , a ) ;
        else
          printf ( " Largest Number is=%d" , b ) ;
    }
```

Can be written as

```
void main()
  {
    int a , b ;
    printf ( " Enter the First number " ) ;
    scanf ( " %d " , &a ) ;
    printf ( " Enter the Second number " ) ;
    scanf ( " %d " , &b ) ;
    printf ( " Largest Number is = %d " , ( a > b ) ? a : b  ) ;
  }
```

## CHECK YOUR PROGRESS

4. It is possible to have nested ..... statement in C.
5. A switch statement is used to -
   a) switch between function in a program
   b) switch from one variable to another variable
   c) to choose from multiple possibilities which may arise due to different values of a single variable
   d) to use switching variable
6. Study the following C program

```
#include<stdio.h>
void main()
  {
     int a=7, b=5;
     switch( a/ a%b)
        {
           Case1 : a=a-b;
           Case2 : a=a+b;
           Case3 : a=a*b;
           Case4 : a=a/b;
           default : a=a;
        }
  }
```

On the execution of the above program, what will be the value of the variable a ?

## 3.5 ITERATIVE STATEMENT

There are three iterative statements available in C, which is also known as Loop control statement. They are-

*for* statement
*while* statement
*do…while* statement

These statements are known as iterative because the block of statement will be executed until the stated condition become false.

### 3.5.1 *for* Statement

This is the most popular iterative statement. The syntax is –

```
for (
        initialization statement;
        conditional statement ;
        increment/ decrement statement
    )

        {
        .....
        Block of statement
        .....
        }
```

The *for* structure consist of three different statement separated by semicolon, the first statement is the initialization statement, which is executed once before the control entered into the block of statement. The initialization is done using assignment statement. The second statement is the conditional statement. The condition is a relational expression. If the condition is true, the block of statement will executed, otherwise the loop is terminated. The third statement is increment or decrement statement.

The initialization statement initializes the loop variable. Let us take an example -

Example 3.6: Write a C program to print the natural number up to a

given limit?

```
#include<stdio.h>
void main()
  {
      int a, c;
      printf ( " Enter the Limit : " ) ;
      scanf ( "%d", &a ) ;
      for ( c = 0 ; c < a ; c++ )
      printf ( " %d", c ) ;
  }
```

OUTPUT :

```
Enter the Limit : 10
0123456789
```

In the above example the first loop variable c is initialized as zero, then check the condition, is the value of c is less then the value of a, that is 10. For the first execution the condition results true. The control will execute the printf statement which will print the value 0. Next the value of loop variable c is incremented by 1. Thus the current value of loop variable c is now 1. Again check the condition whether 1<10. Since the condition is true therefore the printf statement will executed and print 1. Again increment the loop variable by 1 and repeat the same procedure until the resultant of the condition is false.

## 3.5.2 *while* Statement

The general form of while statement is-

```
while ( test condition )
   {
        Statement(s);
   }
```

The program will repeatedly execute the *statement(s)* inside the while loop until the condition becomes false. If the condition is initially false, the statement will not be executed. Let us consider an example

Example 3.7: Write a C program to print the natural numbers less

than 10 using while loop.

```
#include<stdio.h>
void main()
  {
    int c ;                    // declaration statement
    c=0 ;                      // initialization statement
    while ( c<10)              // loop statement
       {
            printf ( "%d" , c ) ;      /* block of statement within
            c++;                       while loop */
       }
   }
```

Output :

0 1 2 3 4 5 6 7 8 9

In the above example the first one is a declaration statement, which declares that c is an integer type variable. The second one is an initialization statement which initializes the loop variable c to zero. Next one is the while loop. The loop will be executed until the value of the loop variable c is less than or equal to 9. As soon as the value of c is 10 the outcome of the conditional statement of while loop will false and the loop will be terminated. The execution of while loop for each execution can be tabulated as below –

| Execution Number | Value of conditional statement | Result | printf("%d",-c) | value of c (c++) |
|---|---|---|---|---|
| 1 | 0<10 | True | 0 | |
| 2 | 1<10 | True | 1 | |
| 3 | 2<10 | True | 2 | |
| 4 | 3<10 | True | 3 | |
| 5 | 4<10 | True | 4 | |
| 6 | 5<10 | True | 5 | |
| 7 | 6<10 | True | 6 | |
| 8 | 7<10 | True | 7 | |
| 9 | 8<10 | True | 8 | |
| 10 | 9<10 | True | 9 | |
| 11 | 10<10 | False | will not execute | will not execute |

Fig - Execution of while loop for example 3.5

## 3.5.3 *do…while* STATEMENT

Sometime a *while* loop might not serve your purpose. In such situation you might want to reverse the semantics from "run while this is true" to the subtly different "do this, while this condition remains true". In other words take the action, and then, after the action is completed, check the condition. Such a loop will always run at least once. To ensure that the action is taken before the condition is tested, use do…while loop. The syntax is-

```
do
   {

        Statement(s);

   } while ( condition ) ;
```

Mind the syntax with the above two. The do…while statement always terminated with a semicolon, whether others two are not. It first executes the statement(s) and then checks whether the condition is true or false. It will repeatedly execute the statement(s) until thecondition become false. Let us take an example –

Example 3.8: Write a C program to print all the even number less than or equal to10 using do…while loop.

```
#include<stdio.h>

void main()
   {
      int c=2;
        do
         {
            printf( "%d  " , c) ;
            c=c+2;
         } while( c < = 10 );
    }
```

Output:

```
2  4  6  8  10
```

## 3.6 *break* STATEMENT

The break statement can only appear in a switch body or a loop body. It causes the execution of the current enclosing switch or loop body to terminate.
The general format of the break statement is :

**break;**

The *break* is a key word in programming language C and a semicolon must be inserted after the word *break*.
break statement with switch - case structure

```
switch ( expression )
        {
            case label1 :
                    statement(s) ;
                    break ;
            case label2 :
                    statement(s) ;
                    break ;
            -   - - - - - - - - - - - - -
            -   - - - - - - - - - - - - -
            default:
             statement(s);
                    break ;
        }
```

The following program shows the use of *break* statement inside *for* loop :

```
#include <stdio.h>

void main()
  {
     int i;

    for (i = 1; i < 10; i++)
      {
        printf ("%d\n", i);

        if (i == 4)
        break;
      }
} // Loop exits after printing 1 through 4
```

Output
1
2
3
4

## 3.7 *continue* STATEMENT

The continue statement forces the next iteration of the loop to take place, skipping any statement(s) following the continue satement in the body of the loop. The syntax of the continue statement is :

*continue;*

The use of continue statement in loops is illustrated in the following figure. In *while* and *do - while* loops, continue causes the control to go directly to the test condition and then to continue the iteration process. In the case of *for* loop, the increment section of the loop is executed before the test test-condition is evaluated.

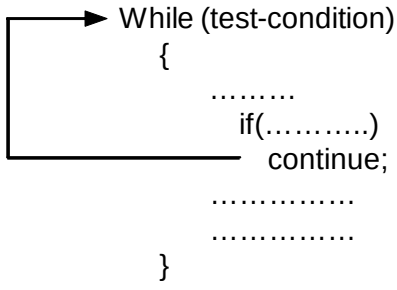



Fig -

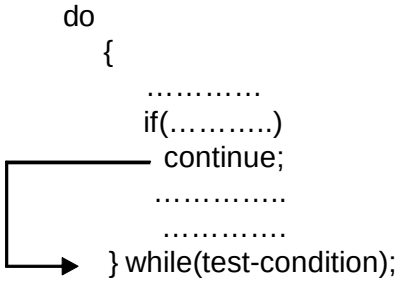
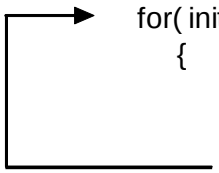



Fig -



```
for( initialization; test condition; increment)
   {
        ……..
        if(……..)
            continue;
            ……..
            .....
   }
```

Fig -

The following program shows the use of *continue* statement with the do-while loop :

```
#include<stdio.h>
void main()
  {
        int i, value;
        i = 0;

        do
         {
            printf (" Enter a number :\n");
            scanf ("%d", &value);
            if(value <= 0)
             {
                  printf("Zero or negative value found \n");
                  continue;
              }
               i++;
          }while (i <= 4);
   }
```

Output :

```
Enter a number :
1
Enter a number :
2
Enter a number :
3
Enter a number :
0
Zero or negative value found
Enter a number :
-1
Zero or negative value found
Enter a number :
4
Enter a number :
5
```

## 3.8 *goto* STATEMENT

The goto statement is used to transfer the control in a program from one point to another point unconditionally. This is alsoo called unconditional branching. The syntax of the goto statement is :

*goto label;*

where label is a valid identifier used in a programming language C to indicate the destination where a control can be transferred. The syntax of a label is :

*label :*

The various 'case' statements in a switch construct also serve as labels used to transfer execution control.
The following program shows the use of *goto* statement with the for loop :

```
#include<stdio.h>
void main()
 {
      int i, value;
      for(i=0; i<=10; ++i)
      {
          printf ("Enter a number \n");
          scanf ("%d", &value);
          if (value <= 0)
            {
                  printf ("Error :");
              printf (" Zero or negative value found \n");
              goto error;
             }
       }
      error :
      ;          // Null statement
 }

Output :
Enter a number
1
Enter a number
2
Enter a number
0
Error : Zero or negative value found
```

## CHECK YOUR PROGRESS

7. Consider the following code fragment

```
for(digit = 0; digit<9; digit++)
 {
        digit = 2*digit;
        digit --;
 }
```

How many times the loop will be executed -
a) Infinite b) 9 c) 4 d) 0

8. How many times will the following loop be executed-

```
ch = 'b';
while (ch>= 'a' && ch <= 'z')
```

a) 0 b) 25 c) 26 d) 1

9. Write the output of the following program -

```
#include<stdio.h>
void main()
 {
    int i=0, sum =0;
     while(i<20)
       {
          if( i%5 = = 0)
            {
               sum += i;
               printf("%d", sum);
             }
               ++i;
          }
        printf ( "\n Sum = %d", sum);
    }
```

10. Write a C program to check whether a given number is a palindrome or not.

A palindrome is a sentence, phrase, name or number that reads the same forwards as it does backwards. A good exaple would be the word ''racecar." R-a-c-e-c-a-r. r-a-c-e-c-a-r or radar. R-a-d-a-r. r-a-d-a-r.
or the number : 16461, 1234321....etc

## 3.9 LET US SUM UP

1. C library functions used for I/O operations can be classified into three categories i.e. console I/O function, disk I/O function and port I/O function.
2. Console I/O function is classified into two categories : formatted and unformatted function.
3. *scanf* and *printf* are formatted console I/O function.
4. *getch(), getche(), getchar(), gets(), putch(), putchar()* and *puts()* are unformatted console I/O function.
5. Control statements allows a programmer to change the sequence of instructions for execution.
6. The *if* statement is a conditional control statement that test a particular condition. The *if* statement also allows answer for the kind of either-or condition by using an *else* clause.
7. The *switch* statement is available in programming language C for handling multiple choices.
8. The *loop or iterative statement*, directs a program to perform a set of operations again and again until a specifiedcondition is achieved.
9. The *while* and *do-while* loop constructs are more suitable in situations where prior knowledge of the terminating condition is not known.
10. *While* loop evaluates a test expression before allawing entry into the loop, whereas *do-while* loop is executed atleast once before it evaluates the test expression which is available at the end of the loop.
11. The *for* loop construct is appropriate when in advance it is known as to how many times the loop will be executed.
12. The *break* statement causes an immediate *exit* from the innermost loop structure.
13. The *continue* statement causes the loop to be continued with the next iteration after skipping any statement in between.
14. The *goto* statement is used to transfer the control in a program from one point to another point unconditionally.

## 3.1O FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.

## 3.11 ANSWERS TO CHECK YOUR PROGRESS

1. a)
```
a = getchar();
b = getchar();
c = getchar();
```

b)
```
putchar(a);
putchar(b);
putchar(c);
```

2.
```
a) printf ("%d %d %d",  i, j,k);
b) printf("%d %d", (i+j), (i-k));
c) printf ("%f %d", sqrt(i+j), abs(i-k));
```

3.
```
a) scanf ("%d %d %d", &i, &j &k);
b) scanf ("%d %o %x", &i, &j, &k);
```

4. if - else  5. c)  6. 35

7. a) Infinite  8. b) 25  9. 0 5 15 30
sum = 30

10.
```
#include<stdio.h>
void main()
    {
        long int n, digit, sum = 0, rev = 0;
        long int num;
        printf ("Input the number \n");
         scanf ("%ld", &num);
         n = num;
         do
          {
            digit = num % 10;
            sum += digit;
            rev = rev * 10 + digit;
            num /= 10;
          }
            while (num != 0);
            Printf ("Sum of the digits of the number = %ld\n", sum);
            printf ("Reverse of the number = %ld \n", rev);
```

```
        if (n == rev)
                printf ("The number is a palindrome \n");
          else
                printf ("The number is not a palindrome \n");
}
```

## 3.12 MODEL QUESTIONS

1. What is the purpose of the getchar function ? How it is used within a program ?
2. What is the purpose of the putchar function ? How it is used within a program ? Compare with the getchar function ?
3. Write down the main purpose of the scanf and printf function. Summarize the meaning of the more commonly used conversion charactes within the control string of a scanf function.
4. Compare the use of gets and puts functions to transfer strings between the computer and the standard input/output devices.
5. What is the purpose of the while statement ? When is the logical expression evaluated ? What is the minimum number of times that a while loop can be executed ?
6. What is the purpose of the do-while statement ? How does it differ from the while statement ?What is the minimum number of times that a while loop can be executed ?
7. How does a for loop differ from a while and do-while statement ?
8. Why mainfunction is special ? Give two reason ?
9. What is the difference between entryh controlled loop and exit controlled loop ?
10. What is the similarity and difference between break and continue statements ?
11. Write a C program to find the factorial of a number ?
12. Write a C program which accepts a number and prints the sum of digits of this number ?
13. Write a program to find the odd and even numbers upto 100.
14. Write a program to convert a binary number to its equivalent decimal number.
15. Write a program to read a string of characters and print out hte following :
    a) Number of uppercase alphabets (A,B,C,.....)
    b) Number of lower case alphabets (a,b,c,...)
    c)Number of special characters (+,-,/,*,....)
16. Write a program to obtain the following output

a)
```
*
* *
* * *
```

b)
```
  *
 * * *
* * * * *
```

c)
```
    1
  2 2 2
3 3 3 3 3
```

```
*****
```

# UNIT- 4 STORAGE CLASS

## UNIT STRUCTURE



## 4.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- learn about Storage Class
- describe Automatic, External, Static and Register Variable
- describe Scope of Variables
- define lifetime of variables
- define Macros
- describe Preprocessor Directives

## 4.2 INTRODUCTION

Already we have the basic idea about the variable. Variable is nothing but the memory location where we can store the values of a particular data type. The value stored in the variable may be changed during the program executions; it will depend on your program coding.

Every C variable has a storage class and a scope. This storage class determines the part of memory where storage is allocated for an object and how long the storage allocation continues to exist during the execution of program. Also it determines the scope which specifies the part of the program over which a variable name is visible, i.e. the variable is accessible by name. We will be much clear about it when go for example.In this unit we will discuss the various storage classes and explain them with examples.

## 4.3 STORAGE CLASS

Storage class refers to the permanence of a variable and its scope within the program.It determines the part of memory where storage is allocated for an object (particularly variables and functions) and how long the storage allocation continues to exist.

The storage class specifier used within the declaration determines whether:

- the object is to be stored in memory or in a register.
- the object receives the default initial value or an indeterminate default initial value.
- the object can be referenced throughout a program or only within the function, block,or source file where the variable is defined.

Here, the term 'object' refers to variable, function and paramneters in which storage class is going to be used. Depending upon the above, specified storage class can be classified into four categories:

i. Automatic
ii. Register
iii. Static
iv. External

Now, we will go to details for each storage class in the next sections with examples.Then it would be more clear to us.

## 4.4 AUTOMATIC VARIABLE

We are already aware of the variable used with the C program. So we can now use storage class in the variable declarations. Actually, we have already used the automatic storage class in our program in the earlier sections. Can you identify that?. Take an example as shown below:

**Example1:**

```
void main()
  {
    int a,b,s;              // or we can write here as auto int a,b,s;
    scanf("%d %d", &a,&b);
    s=a+b;
    printf("Sum is %d",s);

  }
```

Yes ! you are thinking right. By default all the variables declared are automatic.We can also explicitly write it using **auto** keyword before the data type. So what are the special properties of automatic storage that makes it different from other storage class ?

Already we have discussed that storage class is related to the location i.e. where it is stored ? What is its default intial value? What is its scope i.e., whether variable is active within the block or program etc?
Automatic variable are declared at the start of a program's block such as in the curly braces ({ }).Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block.

The automatic variable has the following characteristics:

a)This variable stores the value in the memory of a computer.

b) By default this variable has grabage value. It means that If the variable is not initialized then the variable contains useless value.

c) Automatic variable is local to the block in which the variable is declared.Outside this block the variable can not be accesed. This is known as the **scope** of the variable.

d) Life time of this variable is till the control of the program remains within the block where it is declared.

**Example 2:**

```
void  main ()
 {
     auto int i,j;
     i=10;
      printf("i= %d \n j= %d",i,j);
 }
```

Output :

```
i=10
j=8214  ( or any garbage value )
```

Since here we initialized **i** to 10 explicitly, therefore the value of **i** is printed as 10; But in case of **j** we do not assign any value so the garbage value is printed. This example describes about the default intial value.Now, we go for the life time and scope of automatic variables. Take another example:

**Example 3:**

```
void main ()
 {
    auto int i=1;
      {
        auto int i=2;
          {
              auto int i=3;
```

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

Output:

```
3
2
1
```

The program has three blocks and each block initializes the value of **i**. Note that variable **i** allocates extra memory for each declarations and each **i** is different from one another. So, the first inner block is executed and in this block **i =3;** therefore 3 is printed. After that the second inner block is executed resulting 2 be printed and last 1 is printed for the first block.

So we have seen that the value of **i** is different in each block.Whatever the value we have initialized with **i,** it remains valid only within the block where we have declared.This is known as **scope** of the variable.

## 4.5 EXTERNAL STORAGE CLASS

Already we have learnt about automatic variable which is local to the block in which it is declared. But sometimes we need a variable which should be available to all the functions and blocks within the program. External storage class makes it possible.The properties of the external variable are:

a) This variable also stores value in the memory.

b) Unlike the automatic variable, by default zero is initialized in this type of variable.

c)The scope of this variable means its availabiltiy.It is available to all the funcitons and blocks within the program.

d) Life time of this variable is until the program execution does not stop.

The main difference between automatic and external variable is in the scope and life time of the variables; they have similarities in storage and default initial value. External variables are declared outside all funcitons.Now we go for example of external variable regarding the scope and life time of the variable :

**Example 4** :

```
int var;   // external variable
 void main()
 {
       printf("%d",var);   //  just print the value of  'var'
        var_add_two();           // add 2 to var and display it
         var_sub_one();
 }

 void  var_add_two()
 {
            var=var+2;
             printf("\n%d",var);
 }
 void  var_sub_one()
{
            var=var-1;
             printf("\n%d",var);
          }
```

Output:

0
2
1

Here first output is 0 since default initial value of external variable is 0.Next we increment 'var' by 2, so the next output is 2 and after that we decrement the value of 'var' by one so the output is 2-1 i.e. 1. Note that the value of 'var' is visible to the funcitons var_add_two() and var_sub_one() each of which modifies value of 'var'.

## CHECK YOUR PROGRESS

Q1. Write true or false:

a) The default initial value of external variable is same with automatic variable.
b) By default the variables declared are automatic.
c) The storage for the automatic variable is in the registers.

Q2. Write down one similarity between automatic and external variables.

## 4.6 STATIC VARIABLE

Static variable is mostly similar to the automatic variable; Like the automatic variables static variables are local to the block in which they, are declared.The difference between them is that static variables value does not disapear when the function is no longer active.Their last updated value always persists.That is, when the control comes back to the same function again the static variables have the same value as they leave at the last time. Properties of static variables are:

a) Storage location for static variable is in Memory.

b) These variables are automatically initialized to zero upon memory allocation just as external variables are.

c) The scope of static automatic variables is same with the automatic variables, i.e. it is local to the block in which it is defined;

d) Static storage class provides a lifetime over the entire program.

**Example 5:**

```
voi d  add_one();
void main()
{
 add_one();
 add_one();
}
void  add_one()
{
  static int  var=3;
  var=var+1;
   printf("\n %d",var);
}
```

Ouput:
```
4
5
```

We have seen that the output of the above program is 4 and 5. Since we have initialized 'var' with 3 and then increment 'var' by 1 so first output is 4 during the first call of the add_one() function. The variable 'var' retains its previous value 4 and thus in the second call of the add_one() functon increments 4 by 1; thus the output is 5.If we write the above function defination as:-

```
  void  add_one()
        {
          int  var=3;  // or auto int var=3;
          var=var+1
         printf("\n %d",var)
        }
```

The ouput of the **Example 5** program should be -

4
4

Do you get the point why ? The main reason behind this is that the automatic storage class variable does not retain its previous value.Whenever the add_one() function is called 'var' has always initialized value 3 and then increment by 1; so the output is always 4.

## 4.7 REGISTER STORAGE CLASS

We have already discussed about the automatic,static,external storage classes;each class stores variables in the memory of the computer.We all know that there are mainly two areas for storing data in the computer:- Memory and CPU register. Accessing data from the register is faster than from memory. This concept makes the program to run faster.Register storage class makes it possible.We use the register class with the variables which are going to be accesed frequently. Can you identify such situations in C program where a variable is frequently used ? Yes ! in the loop (such as for(),do-while() etc ) variables are accesed mostly.

The characteristics of such variables in terms of scope,life time etc are:-
a) Unlike other storage class, the storage area for such varibale is in CPU register.
b) By default,garbage value is initialized to such variable.
c) The scope of this variable is local to the block where it is declared.
d) Life time of such variable is till the control remains within the particular block where it is declared.

**Example 6:**

```
void main()
    {
        register int var;
        for(var=1; var<=10; var++)
            printf("%d",var);
    }
```

One confusion may arise in your mind that if we declare most of the variable as register then every program should run faster.But then why do we not use this concept always? Because, the number of registers is limited in a computer system and so we can not use register class to all variables.If in a program the total number of register variables exceeds the system register quantity; then all the variables that exceed are default automatically.We need not take care for this.
For example, if we write a program with loops that uses 20 register variables (assume) and run it with the computer that has only 16 CPU registers, then the rest 4 variables are automatically transformed to automatic storage class variable.One important point is that the register variable can not be used for float and double data type since CPU registers capacity usualy 16 bit. Both float and double data type require 4 byte ( 4 x 8 =32 bit ) and 8 byte ( 8 x 8 = 64 bit ).

**CHECK YOUR PROGRESS**

Q3. Identify from the following which are / is true:
a) Default initial value of Register storage class variable is Zero.
b) Scope of External and Register variables are not same.
c) There is no limit of using register variable.

Q4. Write one difference between register and auotmatic storage class variable.

## 4.8 MACROS

The general form for a simple macro definition is

*#define macro-name value*

and it associates with the **macro-name** whatever **value** appears from the first blank after the **macro-name** to the end of the line. The value constitutes the body of the macro. Previously defined macros can be used in the definition of a macro. Notice that the **value** of the macro does not end with a semicolon. The preprocessor replaces every occurrence of a simple macro in the program text by a copy of the body of the macro, except that the macro names are not recognized within comments or string constants. Because the macros are used within expressions in the body of the program, it is not appropriate to end a macro with a semicolon. Macros that represent single numeric, string or character values can also be referred to as defined constants. Some examples of simple macro definitions are

```
#define PI 3.1415926          /* the value of Pi */

#define  ELECTRON 9.107e-28  /*mass of an electron at rest in
                                grams */

#define  PROTON 1837 * ELECTRON      /*mass of a proton at
                                       rest  in grams */
```

The **#define** directive can also be used for defining parameterized macros. The general form for defining a parameterized macro is

```
#define macro-name(param1, param2, ...)      body-of-macro
```

Parameterized macros are primarily used to define functions that expand into in-line code. Some examples of parameterized macro definitions are

```
#define   ABS(N)    ( (N) >= 0 ? (N) : -(N) )

#define   READ(I)   scanf( "%d", &I )
```

Consider the simple C program shown below :

```
#include <stdio.h>

#define SQR(x)     x * x     /* square a number */

int main()
 {
   int result;
   int a=5, b=6;

    result = SQR( 4 );          /* everything is OK */
    result = SQR( a+b );        /* not what was desired */

 }
```

In the above example, the passing of **a+b** to the macro results in the expanded code:

```
result = a+b * a+b;
```

which is evaluated as **a + (b * a) + b** which will not give the answer that was expected. What was expected was **121** and what was received as **41**. The problem of course is the evaluation of the operators involved in with the expanded macro. The multiplication operator is evaluated before the addition operator. By adding parenthesis the following is produced:

```
#define SQR(x)      (x * x)
```

which when passed **a+b**, expands to **(a + (b*a) + b)** which does not give what is desired. Therefore, more parenthesis are required:

```
#define SQR(x)      ( (x) * (x) )
```

This will finally give the desired results to the macro expansion.

## 4.9 PREPROCESSOR

A unique feature of c language is the preprocessor. A program can use the tools provided by preprocessor to make his program easy to read and modify as well as portable and more efficient.

The prepocessor for 'C' is a collection of special statements, called *directives,* which are executed at the begining of the program compilation. The commands for the preprocessor are inserted in C source-code, that is, (.c) files and are called ***compiler directives***. Each compiler directive is prefixed by a hash sign (#).
The general rules for defining a preprocessor are as follows :

a) All the preprocessor directives begin with hash (#) sign.
b) They must start in the first column.
c) The preprocessor directive should not be terminated by semicolon (;)
d) There should be only one preprocessor directive on one line.

Examples of preprocessor directives are :

| *Directive* | *Function* |
|---|---|
| #define | Defines a macro substitution |
| #undef | Undefines a macro |
| #include | Specifies the files to be included |
| #ifdef | Test for a macro definition |
| #endif | specifies the end of #if |
| #ifndef | Tests whether a macro is not defined |
| #if | Test a compile time condition |
| #else | Specifies alternatives when #if test fails |

**#define (Macro Directrive):** 'C' allows defining an identifier having constant value using #define directive. This is called a preprocessor directive. This directive is placed at the begining of a C program. The symbol # occurs at the first column and no semicolon is allowed at the end. The following program shows the use of macro directive.

```
#include<stdio.h>
#define PI 3.14
main()
 {
     float r= 5.25;
     float area;
      area = PI * r * r;
       printf("\n Area of a Circle = %f", area);
}
```

In the program the variable PI will be replaced by its value 3.14, which is defined using #define function.

**#include (File Directive):** #include is also a preprocessor directive. This directive causes one file to be included in another. For example, *#include<stdio.h>,* which appears at the start of a program.

#include statement causes the contents of the file stdio.h to be inserted into the program at the start of the compilation process. The information contained in the file is essential for the proper functioning of the library functions, such as getchar() and putchar(), etc.

**#undef (Macro Directive) :** Sometimes it is essential to undefine a macro that is already defined. This can be accomplished by #undef directive. #undef removes a macro definition from the macro symbol table. A macro is then no longer defined, unless it is redefined.

**#Conditional (Compilation Directive) :** The preprocessor conditional compilation comand allows lines of source text to be parsed through or eliminated by the preprocessor on the basis of a computed condition. Some examples of preprocessor conditional commands are **#if, #else, #endif, #elif** etc.

The #if directive is used to test whether an expression evaluates to a nonzero value or not.

For example,

```
#if MAXMARKS >= 40
   Statement1;
   Statement2;
#else
   Statement4;
   Statement5;
#endif
```

## 4.10 LET US SUM UP

1. Storage class of a variable determines the locations, default intial value,scope and life time.

2. By default all the variables declared are automatic in nature; it is not mandatory to specify it explicitly.

3. Except the register storage class all other storage class variable locations are in the memory.

4. Static storage variable retains its previous value during the program executions. Such class variable is usually used when the program needs to share a variable.

5. Since locations of register storage class variable are in the CPU registers, so those programs run faster than where the other class varaible is used.

6. Because, the number of CPU registers is limited in a computer system so we can not use register class for all variables. We generally use for the loop counter which is used most frequently.

## 4.11 FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.

2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.

## 4.12 ANSWERS TO CHECK YOUR PROGRESS

Q1. a) False
b) True
c) False

Q2. Storage location for both class variables are in the memory.

Q3. a) False b) True c) False.

Q4. Register class variable stores value in CPU registers whereas automatic class variable use memory for storing data.

## 4.13 MODEL QUESTIONS

1. What do you mean by the storage class of a variable ? How many types of storage class exist ?

2. Compare the external and the automatic storage class variable.

3. Explain the static and automatic storage class variable with examples.

4. Why are the register storage class variables generaly used with loop counter variables ?

5. What is common among automatic , static and external variables ?

6. What is a macro ? Summarize the similarities and differences between macros and function.

7. Summarize the various preprocessor directives, other than *#include* and *#define*. Indicate the purpose of the more commonly used directives.

8. What is the scope of a preprocessor directive within a program file ?

9. Summarize the special preprocessor operators # and ##. What is the purpose of each ?

10. What do you mean by life time of a variable, explain with an example?

11. What do you mean by the scope of a variable?

*****

# UNIT- 5 FUNCTIONS

**UNIT STRUCTURE**



## 5.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about the concept of functions and their usefulness
- declare a function
- define a function
- learn to call a function
- learn about function parameters
- learn about argument passing mechanism
- learn about nesting of function
- describe function categories
- illustrate recursive function

## 5.2 INTRODUCTION

A number of segments grouped into a single logical unit is referred to as ***function***. A function is a set of program statements that carries out some specific task and can be processed independently. In our earlier units while writing programs, we have already used functions like *scanf( )*, *printf( )*, *clrscr( )*, *sqrt( )* etc. We have seen that C supports the use of such ***library*** (or ***built-in***) functions, which are used to carry out a number of commonly used operations or calculations.

However, C also allows the users to define their own functions for carrying out various individual tasks of programming. This unit concentrates on the creation and utilization of such ***user-defined*** functions. With the proper use of such user-defined functions, a large program can be broken down into a number of smaller, self-contained components, each of which has some unique purpose.

## 5.3 USEFULNESS OF FUNCTION

Every C program consists of one or more functions. One of these functions must be called ***main()***. The function *main()* is also a used defined fucntion except that the name of the function, the number of arguments, and the types of the argument are defined by the language. The statements are written by the programmer in the body of the *main()* function. Program execution will always begin by carrying out the instructions in *main()*. There are many advantages in using functions in a program. They are:

- Many programs require that a specific function is repeated many times. Instead of writing the function code as many times as it is required, we can write it as a single function and access the same function again and again as many times as it is required.
- The length of the source program can be reduced by using functions at appropriate places.
- It is easy to locate and isolate a faulty function instead of modifying the whole program.
- Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.

A single function written in a program can be used in other programs also.

## 5.4 GENERAL FORMS OF FUNCTION

The main components of a function are:

- Function Prototype
- Function Definition
- Function Call

### 5.4.1 Function Prototype

***Function prototype*** or ***function declaration*** are usually written at the beginning of the program, ahead of any user defined functions including *main*. It hints to the compiler that the function is going to call the function which is declared, later in the program. The general format of a function prototype is as follows:

*return_type function_name( type1 arg1, type2 arg2,..,typen argn);*

where, ***return_type*** represents the data type of the item that is returned by the function, ***function_name*** represents the name of the function, ***type1, type2, . . . ,typen*** represent the data types of the arguments ***arg1, arg2 , , , , , argn.*** It is necessary to use a semicolon at the end of function prototype. Arguments name *arg1, arg2* etc. can be omitted. However, the argument data types are essential.

A function prototype or declaration provides some information to the compiler. These are:

- The name of the function
- The type of the value returned (by default, interger)
- The number and type of arguments that must be supplied in a call to the function.

For example, in the following function declaration statement

**int add(int, int);**

*int* is the data type of the item returned by the function, *add* is the name of the function and *int* within the brackets '(' and ')' are the data types of the arguments.

**Program1:** Program to find the summation of two numbers.

```
#include<stdio.h>
#include<conio.h>
int add(int,int);          //function prototype or declaration
void main()
{
   int a,b,s;              // integer variable a,b,s are declared
    clrscr();
   printf("Enter two integer number \n");       // display statement
   scanf("%d%d", &a,&b);
   s=add(a,b);                                  //function call
   printf("\nThe summation is  %d", s);
   getch();
 }
```

```
int add(int a, int b)
{
        int sum=0;      // local variable sum and it is intialised to zero
        sum=a+b;
        return sum;     // value of sum is returned
 }
```

**CHECK YOUR PROGRESS**

1.State whether the following statements are true(T) or false (F).

(i) Every C program should have atleast one function.

(ii) In a function declaration arguments are separated by semi colon.

(iii) The function prototype ends with a semicolon.

2. Declare a function with function name “calculate” with two fractional type data as argument and which returns nothing to the calling function.

3. What is the meaning of the statement *int multiply(int,int);* ?

## 5.4.2 Function Definition

Functions can be defined anywhere in the file with a proper declaration, followed by the declaration of local variables and statements. A function definition should contain the following elements:

- name of the function
- list of parameters and their types
- body of the function
- return type

No function definition is allowed within a function definition.General format of function definition is given below :

```
return_type  function_name(parameter list)
{
        local variable declaration;
        executable statement1 ;
        executable statement2 ;
        . . . . . .
        . . . . . .
        return statement ;
}
```

The first line of function definition is known as ***function header*** which contains *return_type*, *function_name* and *parameter_list*. Function

header is followed by an opening '{' and a closing brace '}'. The statements within the opening and closing braces constitute the ***function body.***

Function name should be appropriate to the task performed by the function. In C, two function name should not be same in a single file. If the function does not return anything then the return type will be ***void*** otherwise, it is the type of the value returned by the function. If the return type is not mentioned explicitly, C compiler assumes that it is an ***integer*** type.

***Return*** statement contains the output produce by the function and its type. The *return* statement serves two purposes:

- On executing the *return* statement, it immediately transfers the control back to the calling program.
- It returns the value to the calling program.

**Example:** Let us consider the following program segment

```
int add(int a, int b)
{
      int sum = 0;     // local variable sum and it is intialised to zero
      sum = a + b;
      return sum;    // value of sum is returned to the calling function
}
```

In the above program segment, the summation of the value stored in the variable *a* and *b* are returned. As the summation of two interger is also an integer, so the return type is *int .*

**Example:** Let us consider the following program segment

```
void display( )
{
      printf("State Open University ");
}
```

***display( )*** function does not return anything. So, the return type is ***void***. We can also write the ***display( )*** function as ***void display(void)*** by mentioning ***void*** explicitly within the bracket because no argument is passed.

### 5.4.3 Function Call

Once a function has been declared and defined, it can be called from anywhere within the program: from within the *main()* function, from another function, and even from itself. A function comes to life when a call is made to the function. We can call a function by simply using the function name followed by a list of parameters if any, enclosed in parentheses. For example,

**s = add (a,b) ;** //Function call

In the above statement **add(a,b)** function is called and value returned by it is stored in the variable **s**. When the compiler encounters a function call, the control is transferred to **int add(int x, int y )**. This function is then executed line by line as described and a value is returned when a return statement is encountered. In our example, this value is assigned to **s**. This is illustrated below:

**Program2:** Program to find the summation of two numbers using function

```
#include<stdio.h>
#include<conio.h>
int add(int, int);                    // function declaration
void main()
{  int a,b,s;
     clrscr();
    printf("Enter two integer number \n");
    scanf("%d%d", &a,&b);
    s=add(a,b);                       // function call
    printf("\nThe summation is  %d", s);
    getch();
 }
int add(int x, int y)                 //function header
{
    int sum=0;    // local variable sum and it is intialised to zero
    sum=x+y;
   return sum;  // value stored in sum is returned to the calling function
}
```

**Program3**: Program to find the maximum of two numbers

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

```
void main()
{
      int max(int, int);
      int a,b, big;
      printf("\nEnter two numbers:");
     scanf("%d%d", &a,&b);
     big=max(a,b);
     printf("\nThe maximum of two numbers is: %d", big);
     getch();
}
int max(int x, int y)
{
      int large;
      if(x>y)
            large = x;
      else
            large = y;
     return large;
}
```

The above program will give the maximum of two integer numbers. The statement **int max(int, int);** is the declaration or prototype of the function. The function definition includes statements that test the two input argument numbers and returns the larger number. The variable ***a***,***b***, ***big*** are local to the function ***main*** and the variable ***large*** is a local variable to the function ***max***.

The ***scope*** of the variable is local to the function, unless it is a global variable. For example,

```
int function1(int i)
{
        int j = 50;
        double function2(int j);
        function2(j);
}
double function2(int p)
{
       double m;
        return m;
 }
```

The variable ***j*** in ***function1*** is not known to ***function2***. We pass it to ***function2*** through the argument ***j***. This will be assigned as equal to ***int p***. In the same way, ***m*** in ***function2*** is not known to ***function1***. It can be made known to ***function1*** through the return statement. This makes the scope rules of variables in function quite clear. The scope of variables is local to the function where defined. However, global variables are accessible by all the functions in the program if they are defined above all functions.

## 5.5 FUNCTION PARAMETERS

Functions in C exchange information by means of parameters. The term ***parameter*** refers to any declaration within the parentheses following the function name in a function declaration, definition or function call. Function parameters can be classified into formal and actual parameters.

- **Formal Parameters**

The parameters which appear in the first line of the function definition are referred to as *formal parameter* (commonly called *parameters*). Formal parameters are written in the function prototype and function header of the definition. Formal parameters are local variables which are assigned values from the arguments when the function is called.

- **Actual Parameters**

When a function is called, the values (expressions) that are passed in the call are called the *actual parameters* (often known as *arguments*). At the time of the call each actual parameter is assigned to the corresponding formal parameter in the function definition. It may be expressed in constants, single variables, or more complex expressions. However, each actual parameter must be of the same data type as its corresponding formal parameter.

The following rules apply to parameters of C functions:

- Except for functions with variable-length argument lists, the number of arguments in a function call must be the same as the number of parameters in the function definition. This number can be zero.
- Arguments are separated by commas.
- The scope of function parameters is the function itself. There-

fore, parameters of the same name in different functions are unrelated.

Let us consider the following example to illustrate formal and actual parameters:

**Program4:**

```
#include<stdio.h>
#include<conio.h>
void display(int, int);
void main()
{
      int a,b;
      display(a,b);
       getch();
 }
 void display(int x, int y)
 {
      printf("%d%d",x,y);
 }
```

Here, **x** and **y** are formal parameters and take the value (**a,b**) from the calling function **display(a,b)**.

### 5.5.1 Passing Arguments to a Function

Parameter passing is a method for communication of data between the *calling function* and *called function*. C provides two mechanisms to pass arguments to a function.

- pass by value
- pass by reference

***Pass by value***

Passing arguments by value means that the contents of the arguments in the calling function are not changed, even if they are changed in the called function. This is because the content of the variable is copied to the formal parameter of the function definition, thus preserving the contents of the arguments in the calling function. Pass by value method thus not allow information to be transferred back to the calling portion of the program via arguments.

**Program5:** Program to illustrate pass by value.

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

```
void func(int, int);
void main(void)         //calling function
{
     int x = 5, y = 10;
      clrscr();
     func(x, y);
     printf("In main, x = %d    y = %d\n", x, y);
}
void func(int a, int b)    //called function
{
      a = a + b;
      printf("In func, a = %d    b = %d\n", a, b);
}
```

**Output :**

```
In func, a = 15 b = 10
In main, x = 5  y = 10
```

In the above example, the *calling function* ***main()*** passes two values 5 and 10 to the *called function* ***func()***. The function ***func()*** receive copies of these values and accesses them by the identifiers **a** and ***b***. The function ***func()*** changes the value of ***a***. When control passes back to ***main()***, the actual values of **x** and **y** are not changed.

***Pass by reference***

Pass by reference refers to a method of passing the address of an argument in the calling function to a corresponding parameter in the called function. For better understanding the concept of pass by value and pass by reference, let us consider the following two examples. In the example, our aim is to swap (interchange) two values.

**Program6:** Example to illustrate calling a function by value.

```
#include<stdio.h>
#include<conio.h>
void swap(int, int);   //function prototype or declaration
void main()
{
      int a,b;
      a=5;
      b=10;
      printf("In main(), a and b before interchange: %d %d", a, b);
      swap(a,b);   //function call
      printf("\nIn main(), a and b after interchange: %d %d", a, b);
```

```
    getch();
}
void swap(int i, int j)   //function definition
{
    int t;
   printf(\nWithin swap(), i and j before interchange: %d%d", i,j);
    t = i;
    i = j;
    j = t;
  printf(\nWithin swap(), i and j after interchange: %d%d", i,j);
}
```

Here, the values of **a** and **b** are passed through **swap(a, b);** When we execute this program, no interchange takes place within the **main()** function before and after calling **swap(),** although the interchange takes place within **swap()**. The output will be like this:

```
In main(), a and b before interchange: 5  10
Within swap(), i and j before interchange: 5 10
Within swap(), i and j after interchange:  10 5
In main(), a and b afetr interchange:   5  10
```

To make this function work correctly we can use *pointers*, as shown below. Our next unit will help you in understanding pointers. So, instead of passing two integers to the **swap()** function, we can pass the addresses of the integers that we want to swap.

**Program7:** Example to illustrate passing arguments by reference

```
#include <stdio.h>
#include<conio.h>
void swap(int *, int *);    //function declaration
void main( )
{
    int a,b;
    a=5;
    b=10;
    clrscr( );    // clearing the screen
   printf("a and b before interchange: %d %d\n",a,b);
   swap(&a,&b); //function call, address of variable a and b are passed
   printf("a and b after interchange: %d %d\n",a,b);
}
void swap(int *i, int *j)
{
```

```
    int t;
    t = *i;
   *i = *j;
   *j = t;
}
```

Here, the arguments are passed by reference. To accomplish this, first, the function definition must be changed to accept the addresses of the two integers. This is done by specifying

**void swap(int *i, int *j)**

instead of

**swap(int i, int j)**

Inside the function, instead of using **i**, which now means the memory address of an integer, we have to access the value that is addressed by **i**. This is done by using ***i** and ***j** instead of **i** and **j** respectively. Secondly, the call in **main()** must be changed to pass the addresses of **a** and **b** instead of their values. This is done by calling

**swap(&a, &b);**

Addresses are passed by using the symbol **&** and the valuees are accessed by using the symbol *. When the function **swap** is called, addresses of **a** and **b** are passed. Thus, **i** points to **a** and **j** points to **b**. Once the pointers are initialized by the function call, ***i** is another name for **a**, and ***j** is another name for **b**. When the code uses ***i** and ***j**, it really means **a** and **b**. When the function **swap()** is called, the actual values of the variables **a** and **b** are exchanged because they are passed by reference. The output will be :

```
a and b before interchange:  5  10
a and b after  interchange:  10  5
```

***Pointer :***
*A pointer is a variable that holds the address of another variable.*

**Program8**: Program to generate fibonacci numbers using iteration.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    void Fibo(unsigned int n);     //function prototype
    unsigned int n;
    printf(”Program to generate fibonacci numbers: ”);
    printf(“\nEnter the total number to be generated:”);
    scanf(“%d”,&n);
    Fibo(n);            //function call, n is the argument to the function
}
```

```
void Fibo(unsigned int num)
{
   unsigned int p=1, q=0;
   unsigned int current;
   printf("\nFibonacci numbers:\n");
   printf("0\n");
   do
   {
            current=p+q;
             printf("\n%u\n", current);
             q=p;
             p=current;
             num - -;
  }while(num>1);
//return;
}
```

## CHECK YOUR PROGRESS

4 Fill in the blacks:

(i) When a function returns nothing then the return type is_____.

(ii) If a C program has only one function then that function is _____.

(iii) The parameters used in a function call are_____.

(iv) When a variable is passed to a function by value, its value remains_____in the calling program.

(v) A function can be called either by _____ or _____or both.

5. Write down the syntax of function definition.

6. Write the first line of the function definition, including the formal argument declarations, for each of the situations described below:

(i) A function called ***average*** accepts two integer arguments and returns a floating-point result.

(ii) A function called ***convert*** accepts a character and returns another character.

## 5.6 NESTING OF FUNCTIONS

C permits nesting of functions freely. There is no limit to how deeply functions can be nested. A *nested function* is encapsulated within another function.

For example, a function **a** can call function **b** and function **b** can call function **c** and so on. We have taken the following example to illustrate nesting of function.

**Program9 :** Program to illustrate the nesting of function

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,c;
        float r;
        clrscr();
        float ratio(int,int,int);      // function ratio( ) declared
        printf("Enter a,b and c :");
        scanf("%d%d%d",&a,&b,&c);
        r=ratio(a,b,c);              // ratio( ) function called
        printf("%f\n",r);
        getch();
}
float ratio(int x, int y, int z)
{
        int difference(int,int);     // function difference( ) declared
        if(difference(y,z))
                return(x/(y-z));
        else
                return(0,0);
}
int difference(int p, int q)
{
        if(p!=q)
            return(1);
        else
            return(0);
}
```

The above program calculates the ratio $\frac{a}{b-c}$ and prints the result. We have the following three functions:

main( )
ratio( )
difference( )

**main( )** reads the value of a,b,c and calls the function **ratio( )** to calculate the value a / (b-c). This ratio cannot be evaluated if (b-c) =0. Therefore, **ratio( )** calls another function **difference( )** to test whether the **difference(b-c)** is zero or not.

## 5.7 CATEGORIES OF FUNCTION

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

- Functions with no arguments and no return values
- Functions with arguments and no return values
- Functions with arguments and one return value
- Functions with no arguments but a return value
- Functions that return multiple values

Now we are going to illustrate the above categories taking one example “Multiplication of two integer numbers”.

**• Functions with no arguments and no return values**

**Program10:**

```
#include<stdio.h>
#include<conio.h>
void multi(void);          //function declaration with no argument
void main( )
 {
     clrscr( );
     multi( );
     getch( );
 }
void multi(void)
 {
    int a,b,m;
    printf(“Enter two integers:”);
    scanf(“%d%d”, &a,&b);
     m=a*b;
    printf(“\nThe product is: %d”,m);
 }
```

- **Functions with arguments and no return values**

**Program11:**

```
#include<stdio.h>
#include<conio.h>
void multi(int,int);          //function declaration with two argument
void main( )
{       int a,b;
        clrscr( );
        printf("Enter two integers:");
        scanf("%d%d", &a,&b);
        multi(a,b);
        getch( );
}
void multi(int a,int b )
{       int m;
        m=a*b;
        printf("\nThe product is: %d",m);
}
```

- **Functions with arguments and one return value**

**Program12:**

```
#include<stdio.h>
#include<conio.h>
int multi(int,int);          //function declaration with two argument
void main( )
{
        int a,b,m;
        clrscr( );
        printf("Enter two integers:");
        scanf("%d%d", &a,&b);
        m=multi(a,b);
        printf("\nThe product is: %d",m);
        getch( );
 }
int multi(int a,int b )
 {
        int z;
        z=a*b;
        return z;     /*return statement. the value of z is returned to
 }                        the calling function*/
```

- **Functions with no arguments but a return value**

**Program13:**

```
#include<stdio.h>
#include<conio.h>
int multi(void);          //function declaration with no argument
void main( )
{
    int m;
     clrscr( );
     m=multi( );
    printf("\nThe product is: %d",m);
    getch( );
 }
int multi(void)
 {
     int a,b,p;
     printf("Enter two integers:");
     scanf("%d%d", &a,&b);
     p=a*b;
     return p;
 }
```

**Return** statement can return only one value. In C, the mechanism of sending back information through arguments is achieved by two operators known as the *address operator* (&) and *indirection operator* (*). Let us consider an example to illustrate this.

- **Functions returning multiple values**

**Program14:**

```
#include<iostream.h>
#include<conio.h>
void calculate(int, int, int *, int *);
void main( )
{
    int a,b,s,d;
     clrscr( );
    printf("\nEnter two integer:");
     scanf("%d%d",&a,&b);
     calculate(a,b,&s,&d);
     printf("\nSummation is:%d \n Difference is:%d", s,d);
    getch();
}
```

```
void calculate(int x,int y, int *sum, int *diff)
{
    *sum=x+y;
   *diff=x-y;
}
```

In the fuction call, while we pass the actual values of a and b to the function calculate(), we pass the address of locations where the values of s and d are stored in the memory.

When the function is called, the value of **a** and **b** are assigned to **x** and **y** respectively and address of **s** and **d** are assigned to **sum** and **diff** respectively. The variables ***sum** and ***diff** are known as pointers and **sum** and **diff** as pointer variables. Since they are declared as int, they can point to locations of **int** type data.

## 5.8 RECURSIVE FUNCTION

When a function calls itself it is called a ***recursive function***. Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. A very simple example is presented below:

**Program15:**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        printf("Recursive function\n");
        main( );
       getch( );
}
```

The output of the above programme will be like this:

```
Recursive function
Recursive function
Recursive function
Recursive function
.....
.....
```

We should terminate the execution abruptly; otherwise the program will execute indfinitely.

The factorial of a number can also be determined using recursion. The factorial of a number **n** is expressed as a series of repeatitive multiplications as shown below:

Factorial of n = n(n-1)(n-2)(n-3).....1

For example, Factorial of 5= 5*4*3*2*1 =120

**Program16:** Factorial of an integer number

```
#include<stdio.h>
 #include<conio.h>
 long int factorial(int);
 void main( )
 {
          int n ;
          long int f ;
          clrscr( ) ;
          printf("\nEnter an integer number:") ;
          scanf("%d", &n) ;
          f=factorial(n) ;
          printf("\nThe factorial of %d is : %ld",n,f) ;
          getch() ;
 }
 long int factorial(int n)
 {
          long int fact ;
          if(n<=1)
               return(1);
           else
                fact=n*factorial(n-1);
            return(fact);
 }
```

Let us see how recursion works assuming n = 5. If we assume n=1 then the factorial( ) function will return 1 to the calling function. Since $n \neq 1$, the statement

fact = n * factorial (n-1);

will be executed with n=5. That is,

fact = 5 * factorial (4);

will be evaluated. The expression on the right-hand side includes a call to factorial with n = 4 .This call will return the following value :

4 * factorial(3)

In this way factorial(3), factorial(2), factorial(1) will be returned. The sequence of operations can be summarized as follows:

fact = 5 * factorial (4)
= 5 * 4 * factorial (3)
= 5 * 4 * 3 * factorial (2)
= 5 * 4 * 3 * 2 * factorial (1)
= 5 * 4 * 3 * 2 * 1
=120

When we write recursive functions, we must have an *if* statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

**Program17:** Find the sum of digits of a number using recursion.

```
#include<stdio.h>
#include<conio.h>
 void main()
 {
     int sum(int);    //function prototype
      int n,s;
       clrscr();
     printf("\nenter a positive integer:");
      scanf("%d",&n);
       s=sum(n);
      printf("\nSum of digits of %d is %d ", n,s);
     getch();
 }
 int sum(int n)
 {
      if(n<=9)
           return(n);
       else
           return(n%10+sum(n/10));    // recursive call of sum()
 }
```

**Output :**

Enter a positive integer: 125
Sum of digits of 125 is 8

Recursion is used for repetitive computations in which each action is stated in terms of a previous result.

## EXERCISE

Q. Write a C program to find the GCD (Greatest Common Divi sor) of two positive integers using recursion.

Q. Write a C program to read in three number and print their maximum and minimum with the help of function.

Q. Write a C program to find the sum of the squares of the even numbers between 1 and 20.

Q. Write a function which returns the area of a triangle when the base and height are given to it as parameters. The function should have proper checking to ensure that both base and height are positive numbers. If not, it is to print an error message and return the area as zero. Write a complete C program to use this function.

## CHECK YOUR PROGRESS

7. State whether the following statements are true(T) or false(F)

(i) The parameters which appear in the first line of the function definition are formal parameter

(ii) Arguments are separated by semicolon.

(iii) The 'C' language does not support recursion.

(iv) The main( ) function can call itself recursively.

(v) You can call main() from any other function.

(vi) The same variable names can be used in different functions without any conflict.

8. Write a C program to generate first *n* Fibonacci terms using recursion.

## 5.9 LET US SUM UP

- A function is a self-contained program segment that carries out some specific, well-defined task.
- A function has three principal components: function prototype or declaration, function call, function definition.
- Function prototype or declaration is always followed by a semicolon.
- Call-by-value copies the *value* of an argument to the corresponding parameter in the called function
- Call-by-reference passes the *address* of an argument to the corresponding parameter in the called function.
- The argument that is passed is often called an actual argument while the received copy is called a formal argument or formal parameter.
- We can pass parameters to a function by value and by reference.
- Functions can return any type that we declare, except for arrays and functions. Functions returning no value should return void.
- A return statement is required if the return type is anything other than void.
- A function definition may be placed either after or before the main () function.
- Functions taking a variable number of arguments must take at least one named argument; the variable arguments are indicated by ... as shown:

```
int func(int  x, float  y, ...);
```

- When a function calls itself, then it is called a recursive function.

## 5.10 FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.

## 5.11 ANSWERS TO CHECK YOUR PROGRESS

1. (i) True (ii) False (iii) True

2. void calculate(float, float);

3. The statement **int multiply(int, int);** is a function declaration where the function name is multiply which has two integer type arguments and its return type is integer.

4. (i) void
   (ii) main( )
   (iii) actual parameters
   (iv) unchanged
   (v) call by value, call by reference

5. General format of function definition is given below :

```
return_type  function_name(parameter list)
{
        local variable declaration;
        executable statement1 ;
        executable statement2 ;
        . . . . . .
        . . . . . .
        return statement ;
}
```

6. (i) float average(int a, int b)
   (ii) char convert(char a)

7. (i) True, (ii)False, (iii) False, (iv) True (v) False (vi) True

8.Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    unsigned long fibo(int);
     int i,n;
```

```
    clrscr();
    printf("\nHow many fibonacci terms do you want ?\n");
    scanf("%d",&n);
    printf("\n%d fibonacci terms are: \n\n",n);
    for(i=1;i<=n;i++)
        printf("%4lu",fibo(i));          //function call
    getch();
}
unsigned long fibo(int n)
{
    if(n==1)
        return(0);
    else
    {
        if(n==2)
            return(1);
        else
            return(fibo(n-1)+fibo(n-2));     //recusive call
    }
}
```

## 5.12 MODEL QUESTIONS

1. Explain the meaning of following function prototypes.
   (a) char func(void);
   (b) double f(double a, int b);
   (c) int calculate(int a, int b);
   (d) void change(int *, int *);
   (e) void display( );

2. What is a function? Are functions require when writing a C program? State three advantages to the use of functions.

3. What is meant by a function call? From what part of a program can a function be called?

4. What are arguments? What is their purpose? What is the relationship between formal and actual argument?

5. What is the purpose of *return* statement?

6. Can a function be called from more than one place within a program?

7. What is recursion? Explain it with example.

8. Write a complete C program that will calculate the real roots of the quadratic equation $ax^2+bx+c=0$.

9. Write a C program using function to find the square of an integer number without using the library function sqrt( ).

10. What is the purpose of the keyword void? Where is this keyword used?

*****

# UNIT- 6 ARRAYS AND POINTERS

## UNIT STRUCTURE



## 6.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn the concept of arrays in C programming
- learn to define and declare array
- pass array to a function as argument
- learn about character arrays
- learn about different string library functions and their usefulness
- learn about pointer variable and use them in programs
- pass pointer to a function
- learn about dynamic memory allocation and its implementation

## 6.2 INTRODUCTION

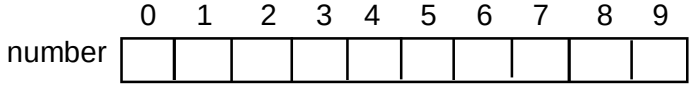
The previous unit gives us the concept of functions in C language. Many applications require the processing of multiple data items that have common characteristics. In such situations it is convenient to place the data items into a linear data structure, where they will all share a common name. The individual data items can be characters, integers, floating-point numbers etc. This unit discusses one of the most important linear data structure called *array*. The unit also introduces pointers and their manipulation. C language uses pointers to represent and manipulate complex data structures. At the end of this unit the concept of dynamic memory allocation is introduced.

## 6.3 ARRAYS

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. This data structure is a finite and ordered set of homogeneous elements.

Suppose we want to store marks of 100 students. In such a case, we may use two options. One way is to construct 100 variables to store marks of 100 students i.e., each variable containing one student's mark. The other method is to construct just one variable which is capable of holding all hundred students marks. Obviously, this option will be easier and for this we can use array.

Again, let us consider another example to understand array more clearly. Suppose we are to store 10 integer values. For this, we can store 10 values of integer type in an array without having to declare 10 different variables, each one with a different identifier. Instead of that, using an array we can store 10 different values of the same type, ***int*** for example, with a unique identifier. An array to contain 10 integer values of type ***int*** called ***number*** could be represented like this:




where each blank panel represents an element of the array, that in

this case are integer values of type ***int***. These elements are numbered from 0 to 9 where 0 indicates the first location. Like a regular variable, an array must be declared before it is used.

## 6.4 DECLARATION OF ARRAY

An array declaration is very similar to a variable declaration. We can declare an array by specifying its data type, name, and the number of elements the array holds between square brackets immediately following the array name. Here is the syntax:

**data_type array_name[size in integer] ;**

For example, int number[5];

In this declaration, 'number' is an interger array of 5 elements which can hold maximum 5 elements. Each array element is referred to by specifying the array name followed by one or more *subscripts*, with each subscript enclosed in square brackets. For the above declaration, the array elements are *number[0], number[1]*, *number[2]*, *number[3]*, *number[4]*.

An array can be initialized at the time of declaration. The general syntax for initializing a one dimensional array at the time of declaration is:

data_type array_name[n] = {element1, element2, ..., element(n-1)};

where,***n*** is the size of the array and ***element1***, ***element2***,....,***elementn*** are the elements of the array. The total number of elements between braces { } must not be larger than the number of elements that we declare for the array between square brackets[ ]. For example,

```
int  num[5] = {16,17, 2,3,4};      /* array initialization at the time of
                                       declaration  */
```

In the example, we have declared an array "num", which has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }. For example, we can write the above statement as:

```
int num[ ] = {16,17, 2,3,4};
```

***Row major order***:
The elements of the first row in a sequence, followed by those of the second, and so on.

Initilization of two dimensional array during declaration is done by specifying the elements in *row major* order. For example,

```
int number[3][4] = {
                        {1, 3, 5, 7 },
                        {11,13, 17, 19},
                        {23, 27, 29, 31}
                    };
```

The *number* is a two dimensional array of integers with certain initial values. The first subscript can be omitted as shown below:

```
int number[ ][4] = {
                        {1, 3, 5, 7 },
                        {11,13, 17, 19},
                        {23, 27, 29, 31}
                    };
```

The inner braces can also be omitted. This can be written as:

```
int number[ ][4] = {1, 3, 5, 7, 11, 13, 17, 19,  23, 27, 29, 31 };
```

We will discuss two dimensional array again while discussing multi dimensional array.

## 6.5 DEFINING AN ARRAY

Arrays are defined in much the same manner as ordinary variables, except that each array name must be accompanied by a size specification. The simplest form of the array is one dimensional array. For a one dimensional array, the size is specified by a positive integer enclosed in square brackets. For example,

```
int num[100];
```

Here, ***num*** is an ***one dimensional*** array of size 100 i.e., maximum number of elements in this array will be 100.

In case of a ***two dimensional array***, an element in the array can be accessed using two indices, ***row number*** and ***column number***. Elements can be accessed randomly. For example:

```
int matrix[20]20];
```

Here, *matrix* is a two dimensional array with 20 rows and 20 columns. The number of rows or columns is called the range of the dimension.

Representation of one dimensional array in memory is straight forward. Elements from *index 0* to some maximum are stored in some contiguous memory locations. Elements are always stored in **row major** fashion. But in case of two dimensional array there are two methods of representation in memory, which are ***row major*** and ***column major***. In row major representation, the first row of the array occupies the first set of memory locations reserved for the array, the second row occupies the next set and so forth. On the other hand, in column major representation the first column of the array occupies the first set of memory locations reserved for the array, the second column occupies the next set and so forth.

## 6.6 ACCESSING ARRAY ELEMENTS

We can access an array using its *name* and the *index* of a particular element within square braces. The *array index* indicates the particular element of the array which we want to access. The numbering of elements starts from zero. The smallest index in an array is called the *lower bound* and the highest index is called *upper bound*. In case of C, the lower bound is always 0. If the lower bound is 'lower' and the upper bound is 'upper', then the number of elements is:

```
upper- lower + 1
```

The following statement

```
int num[5]= {22, 24, 26, 28, 30};
```

represents an one dimensional array of 5 integer numbers. Each element of the array can be accessed with the index *num[0]*, *num[1]*, *num[2]*, *num[3]* and *num[4]*. It is assumed that each element of the array occupies two bytes. The general expression for accessing ***one dimensional*** array 'num' is

```
num[ i ]
```

For example, let us consider the following lines of code:

```
int num[5];
printf("\nEnter the numbers into the array:");
for( i = 0; i < 5 ; i++)
{
   scanf("%d", &num[i]);
}
```

Here, the variable ***i*** varies from 0 to 4. The function *scanf()* is called to input the integer values. The address of $i^{th}$ location is passed to *scanf()* which makes *scanf()* store the integer input into successive locations each time the loop is executed. ***&num[i]*** in the *scanf()* statement refers to the memory location of the integher at the $i^{th}$ position.

The elements of ***two dimensional*** array can be accessed by the following expression:

```
marks[ i ][ j ];
```

where *i* and *j* refers to row and column numbers respectively.

## 6.7 PASSING ARRAY TO FUNCTION

To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within a function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declarations. When declaring a one dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

**Program 1:** Program to find the sum of elements of an array where the array is passed as argument to the function.

```
#include<stdio.h>
#include<conio.h>
int  addition (int a, int x[ ] ) ;          //function prototype
void main()
{          int n,i,add;
           int y[20] ;  //array declaration of maximum size 20
           clrscr();
           printf("\nEnter the size of the array:");
           scanf("%d",&n);
           printf("\nEnter the array elements:");
           for(i=0;i<n;i++)
              scanf("%d",&y[i]);
           printf("\nEntered elements are:\n");
           for(i=0;i<n;i++)
              printf("%d\t",y[i]);
           add=addition(n,y) ;    //array size and name is passed
           printf("\nResult is: %d",add);
           getch();
}
int addition (int a, int x[ ])    // function definition , formal argument
{
           int i,sum=0;
           for(i=0;i<a;i++)
                sum=sum+x[i];
           return sum;          //sum is returned to the main function
}
```

## EXERCISE

Q. Find the average of *n* numbers using array where *n* is the size of the array.

Q. Write a program to find the summation of 10 even numbers where numbers are entered through the keyboard.

## 6.8 MULTIDIMENSIONAL ARRAYS

An array with more than one index value is called a ***multidimensional array*** . Multidimensional arrays can be described as "arrays of arrays". The syntax for declaring a multidimensional array isas follows:

**data_type array_name[ ][ ][ ];**

The number of square brackets specifies the dimension of the array.

***Matrix :***
*A rectangular array of elements (or entries) set out by rows and columns*

**Initialization of multidimensional arrays:**

We have already seen the declation of two dimensional arrays in our previous sections. Like the one dimensional arrays, two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example:

```
int table[2][3]={1,1,1, 2, 2, 2};
```

The above statement initializes the elements of first row to 1 and second row to 2. The initialization is done row by row. The above statement can be equivalently written as

```
int table[2][3] ={ {1,1,1}, {2, 2, 2} };
```

Arrays of three or more dimensions are not used very often because of the memory required to hold them. The computer takes time to generate each index and this can cause the access of multidimensional arrays very slow as compared to a single dimensional array with the same number of elements. Some examples of multidimesional array declation are:

```
int count[3][5][12];
float table[5][4][5][3];
```

Here, ***count*** is a three dimensional array declared to contain 180 integer elements. Similarly, ***table*** is a 4-dimensional array containing 300 elements of floating point type.

Often there is a need to store and manipulate two dimensional data structure such as matrices and tables. In case two dimensional (2D)

array, there are two subscripts. One subscript denotes the row and the other denotes the column. We have already used two dimensional array in our previous sections. The declaration of two dimension arrays is as follows:

**datatype array_name[row_size][column_size];**

For example, int marks[3][4] ;
Here *marks* is declared as a matrix having 3 rows( numbered from 0 to 2) and 4 columns(numbered 0 through 3). The first element of the matrix is marks[0][0] and the last row last column is marks[2][3] .

**Elements of two dimensional arrays:**

A two dimensional array ***marks[3][5]*** is shown below. The first element is given by ***marks[0][0]*** contains 50 and second element is ***marks [0][1]*** and contains 75 and so on.

| | | | |
|---|---|---|---|
| marks[0][0] 50 | marks[0][1] 75 | marks[0][2] 70 | marks[0][3] 61 |
| marks[1][0] 51 | marks[1][1] 35 | marks[1][2] 65 | marks[1][3] 78 |
| marks[2][0] 45 | marks[2][1] 67 | marks[2][2] 28 | marks[2][3] 55 |

To represent a matrix a two dimensional array is required. Suppose there are two matrices A and B having the following elements

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 3 & 8 \\ 6 & 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 3 & 5 \\ 1 & 1 & 1 \\ 1 & 5 & 4 \end{pmatrix}$$

Then the addition matrix will be :

$$C = \begin{pmatrix} 3 & 5 & 8 \\ 6 & 4 & 9 \\ 7 & 8 & 8 \end{pmatrix}$$

One can write the following program for addition of two matrices.

**Program 2:** Program to add two matrices and store the results in the third matrix.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10][10],b[10][10],c[10][10],i,j,m,n,p,q;
        clrscr();
        printf("Enter the order of the matrix a\n");
        scanf("%d%d",&p,&q);
        printf("Enter the order of the matrix b\n");
        scanf("%d%d",&m,&n);
        if(m==p && n==q)
        {
                printf("\nMatrix can be added\n");
        }
        printf("\nEnter the elements of the matrix a:");
        for(i=0;i < m;i++)
           for(j=0;j < n;j++)
                scanf("%d",&a[i][j]);
        printf("\nEnter the elements of the matrix b:");
        for(i=0;i < p;i++)
          for(j=0;j < q;j++)
                scanf("%d",&b[i][j]);
        printf("\nThe sum of the matrix a and b is:\n");
        for(i=0;i<m;i++)
        {
                 for(j=0;j<n;j++)
                {
                        c[i][j]=a[i][j]+b[i][j];
                        printf("%d\t",c[i][j]);
                }
                printf("\n");
        }
        getch();
}
```

**Program 3**: Program to find the sum of the diagonal elements of a matrix

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int a[10][10], i, j, n, trace;
  clrscr();
 printf("\nEnter the order of the matrix:");
 scanf("%d", &n);
 printf("\nEnter the elements of the matrix:\n");
 for(i=0;i<n;i++)
    for(j=0;j<n;j++)
          scanf("%d", &a[i][j]);
 trace=0;
 for(i=0;i<n;i++)
   trace= trace+a[i][i];
 printf("\nThe sum of the diagonal elements = %d",trace);
 getch();
}
```

If we execute the program with the order 3 and the lements of the matrix as 1,2 3, 4, 5, 6, 7,8, 9 the the sum of the diagonal elements will be 15.

## CHECK YOUR PROGRESS

1. State whether the following statements are true(T) or false(F)

(i) Arrays are sets of values of the same type which have a single name follwed by an index.

(ii) If N[10] is an array, then N+2 points to the third element of the array.

(iii) The following is a correct array definition char num(30);

(iv) “B” is a string but ‘B’ is a character.

(v) Elements of one dimensionl arrays are always stored in row major fashion.

(vi) Return statement cannot be used to return an array.

2. Fill in the blanks:
(i) An array is a collection of _____ data items.
(ii)The _____ indicates to the compiler that we are dealing with an array.
(iii) The values used to initialize an array are separated by _____ and surrounded by braces.
(iv) Array elements can be accessed _____.

3. Which of the following declaration is wrong?
(a) int p[10]={1,2,3,4,5}; (b) char ch[ ][3]={{'a','b','c'},{'d','e','f'}};
(c) int j=0, i=j; (d) All are correct

4. What will be the output of the following program code?

```
#include<stdio.h>
#include<conio.h>
void main() {  int n, result=0, number[ ]={2,5,4,6,1};
        clrscr();
        for ( n=0 ; n<5 ; n++ ) {
                if(number[n]==2)
                        continue;
                        result= result+ number[n];   }
      printf("%d",result);
      getch(); }
```

5. Find the number of elements of the following array declarations:
(a) int x[2][3]; (b) float p[9];

## 6.4 STRINGS

A character *(char)* variable can hold a single character. While writing program , sometimes we may have to store a sequence of characters like a person's name, address etc. We need a way to store these sequence of characters. Although there is no special data type for strings, C handles this type of information with array of characters.

A ***string*** is just an array of characters with the one additional convention that a "*null*" character is stored after the last real character in the array to mark the end of the string. Null character is a character with a numeric value of zero and it is represented by '**\0**' in C. So when we define a string we should be sure to have sufficient space for the null terminator. An array of characters representing a string is defined with the following syntax :

**char array_name[size];**

In general, each character of a string is stored in one byte, and successive characters of the string are stored in successive bytes. If we want to store the name KKHSOU, then we have to declare an array of *char* of size 6. Although there are five characters in the name but we need an array of six characters. This extra space is to store the "null" character.

***String constants :***

*String constants* have double quote marks around them, and can be assigned to char pointers as shown below. Alternatively, we can assign a string constant to a *char* array - either with no size specified, or we can specify a size, but we shouldn't forget to leave a space for the null character.

```
char *text = "Hello";
char text[ ] = "Hello";
char text[6] = "Hello";
```

In the third statement the total numbers of characters in the word "Hello" is 5, but as it is a character array so we have considered the size of array text as 6. i.e., one extra space for null.

***Reading and Writing Strings:***

One possible way to read in a string is by using ***scanf()***. However, the problem with this, is that if we were to enter a string which contains one or more spaces, scanf() would finish reading when it reaches a space, or if return is pressed. As a result, the string would get cut off. So we could use the ***gets()*** function. A gets takes just one argument - a *char pointer*, or the name of a *char array*, but we have to declare the array or pointer variable first.

A ***puts()*** function is similar to gets() function in the way that it takes one argument - a *char pointer*. This also automatically adds a newline character after printing out the string. Sometimes this can be a disadvantage, so *printf()* could be used instead. The concept of pointers will be covered later in this unit.

## 6.9.1 Initialization of Strings

C allows to initialize a string at the time of its declaration. Let us consider the following declaration:

**char month[ ]={'A', 'p', 'r', 'i', 'l', '\0'};**

month is a string which is initialized to *April*. This is a valid statement. But C provides another way to initialize strings which is:

**char month[ ]= "April";**

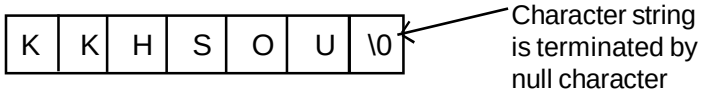
The characters of the string are enclosed within double quotes. The compiler takes care of storing the ASCII (*American Standard Code for Information Interchange*) codes of the characters of the string in memory and also the null terminator in the end.
For example,

```
char name[ ] = {'K','K','H','S','O','U','\0'};
```

We can also have a simpler choice by giving the following declaration:

```
char name[ ] = "KKHSOU";
```

Here the string is surrounded by double quotes (" "). In this method of initialization we do not need to insert the null character '\0' . this will be inserted automatically.




For example let us consider the following two character array definitions. Each includes the assignment of string "KKHSOU" .

```
char university [6] = "KKHSOU" ;    // defined as 6 element array
char university [ ] = "KKHSOU";     // here size is not specified.
```

The results of these initial assignments are not the same because of the null character "\0", which is automatically added at the end of the second string. Thus the elements of the first array are:

university [0] = 'K'
university [1] = 'K'
university [2] = 'H'
university [3] = 'S'
university [4] = 'O'
university [5] = 'U'

Whereas the elements of the second array are:

university [0] = 'K'
university [1] = 'K'
university [2] = 'H'
university [3] = 'S'
university [4] = 'O'
university [5] = 'U'
university [6] = '\0'

The first form is incorrect, since the null character '\0' is not included in the array. So we can define it as:

char university [7] = "KKHSOU" ;

**Program 4:** Reading a sting of characters from the keyboard and displaying it.

```
#include<stdio.h>
#include<conio.h>
void main( )
{       char university [7] ;    //declaration of a string of characters
        university[0] = 'K';
        university[1] = 'K';
        university[2] = 'H';
        university[3] = 'S';
        university[4] = 'O';
        university[5] = 'U';
        university[6] = '\0';          // Null character - end of text
        printf("University name: %s\n", university);
        printf("\nOne letter is: %c\n", university [2]);
        printf("\nPart of the name is: %s\n", &university [3]);
```

```
        getch();
}
```

**Output :**

University name: KKHSOU
One letter is:H
Part of the name is: SOU

## 6.9.2 Arrays of Strings

Arrays of strings (arrays of character arrays) can be declared and handled in a similar manner to that described for two dimensional arrays. Let us consider the following example:

**Program 5:**

```
#include< stdio.h>
#include<conio.h>
void main( )
{
        char names[2][8] = {“KKHSOU”, “IDOL”};
        printf(“Names = %s, %s\n”,names[0],names[1]);
        printf(“\nNames = %s\n”,names);
        printf(“Initials = %c. %c.\n”,names[0][0],names[1][0]);
        getch();
}
```

**Output :**

names = KKHSOU, IDOL
names = KKHSOU
Initials = K. I.

Here we declare a 2-D character array comprising two “roes” and 8 “columns”. We then initialise this array with two character strings, KKHSOU and IDOL.

## 6.9.3 String Manipulations

C language does not provide any operator which manipulate entire strings at once. Strings are manipulated either via pointers or via spe-

cial routines available from the standard string library ***string.h***. The file ***string.h*** available in the library of C has several built in functions for string manipulation. Some of them are :

- strlen( )
- strcpy( )
- strcat( )
- strcmp( )
- strrev( )

To use these funtions we have to include the header file ***string.h*** as shown below:

#include<string.h>

**String Length**

The strlen() function is used to find the number of characters in a given string including the end-of-string character (null). The syntax is as follows:

**len = strlen(ptr);**

where *len* is an integer and *ptr* is the array name where the string is stored. The following program determines the length of a string which is entered through the keyboard.

**Program 6:** Finding the lengh of a string

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
      int len;
      char name[25];
       clrscr();
      printf("\nEnter the name:");
      gets(name);
      len=strlen(name);
     printf("\nLength of the string :%d",len);
      getch();
```

}

If the entered name is *KKHSOU*, then the result will be 6. If we enter *KKHSOU Assam* then strlen() will return 12 counting the blank space as one character.

**EXERCISE**

Q. Find the lengh of a particular string without using string library function strlen().

**String Copy**

The strcpy() function copies the contents of one string to another i.e., source string to destination string. The syntax is:

**strcpy(str1,str2);**

where str1 is the destination string and str2 is the sourse string.

**String Concatenation**

The strcat() function joins or places together two strings resulting in a single string. It takes two strings as argument and the resultant string is stored in the destination string. The syntax is:

**strcat(s1,s2);**

where s1 is the destination string and s2 is the source string.

**String Compare**

The strcpy() function compares two strings, character by character. It accepts two strings as parameters and returns an integer. The syntax is: **strcmp(s1,s2);**

The return value of strcmp() function depends on both the two strings which we compare. If s2<s1, it returns -1
If s2==s1, it returns 0
If s2>s1, it returns 1

Two strings are equal if their contents are identical.

**Program 7:** Program for checking two string which comes first in the English dictionary.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
   int i;
   char s1[20],s2[20];
    clrscr();
   printf("\nEnter a string:");
    gets(s1);
   printf("\nEnter another string to compare:");
    gets(s2);
    i=strcmp(s1,s2);
   if(i==0)
          printf("\nStrings are identical");
   else if(i<0)
          printf("\nFirst string comes first");
   else if(i>0)
          printf("\nSecond string comes first");
   getch();
 }
```

**Program 8:** Combining two strings

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{       char s1[50],s2[20];
        printf("\nEnter your first name:");
        gets(s1);
        printf("\nEnter your last name:");
        gets(s2);
        strcat(s1,s2);
        printf("\nYour full name :%s",s1);
        getch();
```

}

## CHECK YOUR PROGRESS

6. Fill in the blanks:

(i) All strings must end with a _____ character.

(ii) String function strrev() belongs to _____ header file.

(iii) _____ function appends a string to another string.

(iv) strcmp() is string library function which_____ two strings.

(v) A string is an array of _____ .

7. Consider the following code segment:

```
void main()
{
        char s[100];
        scanf("%s", s);
        printf("%3s",s);
}
```

If ***Programming Language*** is entered upon the execution of the program for s, then what will be the output? Options are given below:

(a) Pro

(b) Programming Language

(c) Programming

(d) none of these

8. State whether the following statements are true(T) or false(F).

(i) Two strings are equal if their contents are identical.

(ii) gets() function belongs to the header file *stdio.h* .

(iii) strlen() reverses a string.

(iv) If blank space exists in a string, gets() function reads the string including blank space.

(v) strcat() function takes only one string as argument.

## 6.10 POINTERS

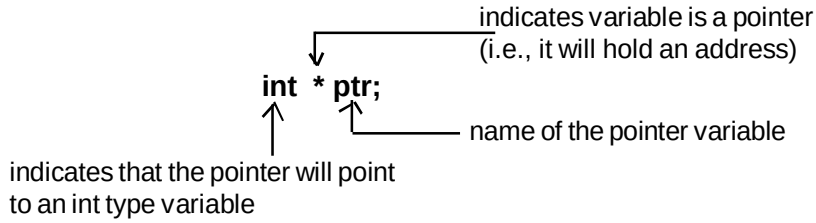
A pointer is a variable that holds the memory address of another variable. We can have a pointer to any variable type. The *unary* operator **&** gives the "address of a variable". The *indirection* or *dereference operator* * gives the "contents of a variabel pointed to by a pointer".

### 6.10.1 Declaration of Pointer

Pointer variables must be declared before they may be used in C program. When a pointer variable is declared, the variable name must be preceded by an asterisk " * ". The data type that appears in the declaration refers to the object of the pointer. The general syntax of pointer declaration is:

**data_type *variable;**

For example, the following is a pointer declaration statement.




where *ptr* is the name of the pointer variable. The following program illustrates the use of pointer.

**Program 9:**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int age,*ptr1,*ptr2;
        age = 50;                    /* any numerical value */
        ptr1 = &age;                 /* the address of age variable */
        ptr2 = ptr1;
        printf("The value is %d %d %d\n", age,*ptr1,*ptr2);
```

```
        *ptr1 = 29;                /* this changes the value of age */
        printf("The value is %d %d %d\n", age,*ptr1,*ptr2);
        getch();
}
```

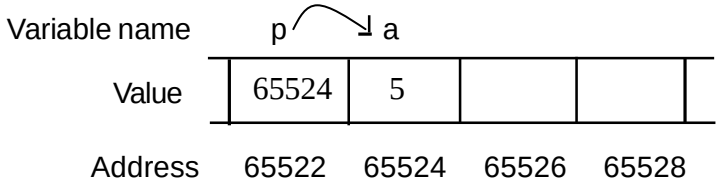
Here "ptr1" and "ptr2" are two pointer variables. So they do not contain a variable value but an address of a variable and can be used to point to a variable. Line 7 of the above program assigns the address of "age" variable to the pointer "ptr1". Since we have a pointer to "age", we can manipulate the value of "age" by using either the variable name itself, or the pointer. Line 10 modifies the value using the pointer "ptr1". Since the pointer "ptr1" points to the variable "age", putting a star in front of the pointer name refers to the memory location to which it is pointing. Line 10 therefore assigns the value 29 to "age". Any place in the program where it is permissible to use the variable name "age", it is also permissible to use the name "*ptr1" since they are identical in meaning until the pointer is reassigned to some other variable.

**Program 10:** Program to demonstrate the relationships among * and & operators.

```
#include<stdio.h>
#include<conio.h>
void main()
{
     int a=5;
     int *p;
     p=&a;
      clrscr();
     printf("\nAddress of a=%u", &a);
     printf("\nAddress of a=%u", p);
     printf("\nAddress of p=%u", &p);
     printf("\nValue of p=%u", p);
     printf("\nValue of a=%d", a);
     printf("\nValue of a=%d", *(&a));
     printf("\nValue of a=%d", *p);
      getch();
}
```

**Output :** 65524
65524
65522
65524
5
5
5

Memory address may be different with different computer. In our case if the memory address of variable **a** is 65524 then the diagrametic representation will be like this:




**Pointer expressions and pointer arithmetic**

Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
x = *p1**p2;
sum =sum+*p1;
y = 5* - *p2/p1;
*p2 = *p2 + 10;
```

C language allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc. We can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

## 6.10.2 Passing Pointer to a Function

Pointers are often passed to a function as arguments. This allows data type within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form. This is called passing arguments by reference or by address.

Here is a simple C program that illustrates the difference between ordinary arguments which are passed by value, and pointer arguments which are passed by reference.

**Program 11:** Arguments are passed by value

```
#include<stdio.h>
#include<conio.h>
void function1(int, int);
void main()
{
        int a =1;
        int b =2;
        printf("Before calling function1 :  a=%d      b=%d" , a  , b );
        function1(a,b);        //passed by value
        printf("After calling function1 :  a=%d      b=%d" , a  , b );
}
void function1(int a , int b )    //function definition
{
    a=5;
    b=5;
    printf("Within the  function1 :  a=%d      b=%d" , a  , b );
    return;
}
```

**Output :**

```
Before calling function1 : a=1 b=2
Within function1 :          a=5, b=5
After calling function1 :    a=1, b=2
```

When an argument is passed by value, the data item is copied to the function. Thus, any alteration made to the data item within the function is not carried over in to the calling routine.

When an argument is passed by reference, the address of the data item is passed to the function. Here, the above example is considered again to see the differences in output. In this case we observe that original value is changed after execution of function2.

**Program 12:** Arguments are passed by reference

```
#include<stdio.h>
#include<conio.h>
void function2(int *, int *);
void main()
{       int a =1;
        int b =2;
        printf("Before calling function2 :  a=%d     b=%d" , a  , b );
        function2(&a,&b);       passed by reference
        printf("After calling function2 :  a=%d     b=%d" , a  , b );
}
void function2(int *pa , int  *pb )         //function
{
        *pa=5;
        *pb=5;
  printf("Within the  function2 :  *pa=%d     *pb=%d" , *pa  , *pb );
        return;
}
```

**Output :**

```
        Before calling function2 :  a=1  ,  b=2
        Within function2:           *pa=5,  *pb=5
        After calling function2 :     a=5,     b=5
```

## 6.10.3 Pointers and One-Dimensional Arrays

An array in C is declared as:

```
int X[5]={2,1,6,9,5};
```

X is an array of integers and it has five elements. If X is a one dimensional array, then the address of the first element can be expressed as either &X[0] or X. Moreover, the address of the second array element can be written as either &X[1] or as X+1 and so on. In general, the address of array element (i+1)can be expressed as either &X[ i ] or X+i . Since &X[ i ] and X+i both represent the address of the $i^{th}$ element of X , it would seem reasonable that X[ i ] and *(X+i) both represent the contents of that address. i.e , the value of $i^{th}$ element of X.

An array is actually a pointer to the $0^{th}$ element of the array. Dereferencing the array name will give the $0^{th}$ element. This gives us a range of equivalent notations for array access. In the following examples, ARR is an array.

| **Array access** | **Pointer equivalent** |
|---|---|
| ARR[0] | *ARR |
| ARR[2] | *(ARR+2) |
| ARR[n] | *(ARR+n) |

There are some differences between arrays and pointers. The array is treated as a constant in the function where it is declared. This means that we can modify the values in the array, but not the array itself, so statements like ARR ++ are illegal, but ARR[n] ++ is legal. Let us consider an **int** variable called **i**. Its address could be represented by the symbol **&i**. If the pointer is to be stored as a variable, it should be stored like this:

```
int *p = &i;
```

**int *** is the notation for a pointer to an **int**. The operator & returns the address of its argument.

The other operator which gives the value at the end of the pointer is *. For example: i = *p;

## 6.11 DYNAMIC MEMORY ALLOCATION

When an array is declared as above, memory is allocated for the elements of the array when the program starts, and this memory remains allocated during the lifetime of the program. This is known as **static** array allocation.

Until this point, the memory allocation for our program has been handled automatically when compiling. However, sometimes the computer doesn't know how much memory to set aside (for example, when you have an unsized array). It may happen that you don't know how large an array you will need (or how many arrays). In this case it is convenient to allocate an array while the program is running. This

is known as **dynamic memory allocation**. Dynamic data structures provide flexibility in adding , deleting or rearranging data items at run time . Dynamic memory management permit us to allocate additional memory space or to release unwanted space at run time.

**malloc and free :**

A block of memory may be allocated using the function **malloc**. The **malloc()** function reserves a block of memory of specified size and returns a pointer of type void. Syntax is as follows:

**ptr = (cast type *) malloc(byte size);**

For example, x =(int *)malloc (100 * sizeof(int));

Here a memory space equivalent to 100 times the size of an integer byte is reserved and the address of the first byte of the memory allocated is assigned to the pointer **x** of type integer.

***malloc*** requires one argument - the number of bytes which we want to allocate dynamically. If the memory allocation was successful, malloc will return a void pointer. We can assign this to a pointer variable, which will store the address of the allocated memory. If memory allocation failed (for example, if you're out of memory), malloc will return a NULL pointer. Passing the pointer into free will release the allocated memory - it is good practice to free memory when you've finished with it. The general format of free() function is:

```
int free(pointer);

char *pointer;
```

The following program will ask you how many integers you would like to store in an array. It will then allocate the memory dynamically using malloc and store a certain number of integers, print them out, then releases the used memory using free.

**Program 13:**

```
#include <stdio.h>
#include <stdlib.h>  /* required for the malloc and free functions */
#include<conio.h>
void main()
```

```
{
        int number;
        int *ptr,i;
         clrscr();
        printf("How many ints would you like to store? ");
        scanf("%d", &number);
        ptr =(int *)malloc(number*sizeof(int));  // allocate memory
        if(ptr!=NULL)
        {
                for(i=0 ; i<number ; i++)
                {
                        *(ptr+i) = i;
                }
                for(i=number ; i>0 ; i--)
                {
                        printf("%d\n", *(ptr+(i-1)));  /* print out in
                                                reverse order */
                }
        free(ptr);     // free allocated memory
    }
    else
    {
          printf("\nMemory allocation failed - not enough memory");
     }   //end bracket of if-else
}// end of main
```

If we enter 4

**Output :** How many ints would you like store? 4

3
2
1
0

**calloc :**

***calloc*** is similar to malloc, but the main difference between the two is that in case of callocc the values stored in the allocated memory space is zero by default. With malloc, the allocated memory could have any value. Calloc requires two arguments. The first is the num-

ber of variables which we like to allocate memory for. The second is the size of each variable. Like malloc, calloc will return a void pointer if the memory allocation was successful, else it'll return a NULL pointer. Syntax of calloc is as follows:

**ptr = (cast type *) calloc (n, element size);**

**realloc :**

The realloc() function is used to change the size of previously allocated block. Suppose, we have allocated a certain number of bytes for an array but later find that we want to add values to it. We could copy everything into a larger array, which is inefficient, or we can allocate more bytes using realloc, without losing our data. *realloc()* takes two arguments. The first is the pointer referencing the memory. The second is the total number of bytes you want to reallocate. Passing zero as the second argument is the equivalent of calling free. Once again, realloc returns a void pointer if successful, else a NULL pointer is returned.

## CHECK YOUR PROGRESS

9. State whether the following statements are true(T) or false(F).

(i) The statement *p++ ; increments the content of the memory location pointed by p.

(ii) The address operator is obtained by *.

(iii) *p++ ; increments the integer pointed by p.

(iv) The address operator (&) is the inverse of the de-referncing operator (*).

(v) Arrays cannot be returned by functions, however pointer to array can be returned.

10. Choose the correct option:

(i) Which is the correct way to declare an integer pointer ?

(a) int_ptr x; (b) int *ptr; (c)*int ptr; (d)*ptr;

(ii) In the expression **float *p;** which is represented as type float?

(a)The address of p (b)The variable p

(c) The variable pointed to by p (d) None of the above

(iii) A pointer is

(a) Address of variable

(b)A variable for storing address

(c) An indirection of the variable to be accessed next.

(d)None of the above.

(iv) Assuming that **int num[ ]** is an one-dimensional array of type int, which of the following refers to the third element in the array?

(a) *(num+4); (b)*(num+2);

(c)num+2; (d)p=&a[3];

(v) Consider the following two definitions int a[50]; int *p;
which of the following statement is incorrect?

(a) p=a+3; (b) a=p;

(c) p=&a[3]; (d)None of these

11. How does the use of pointers economize memory space?

## 6.12 LET US SUM UP

- Array by definition is a variable that hold multiple elements which has the same data type.
- Array elements are stored in contiguous memory locations and so they can be accessed using pointers.
- A string is nothing but an array of characters terminated by null character "\0".
- The header file of string library function is *string.h*
- *strlen()* returns the number of characters in the string, not including the *null character*
- *strcmp()* takes two strings and compares them. If the strings are equal, it returns 0. If the first is greater than the 2nd, then it returns *some* value greater than 0. If the first is less than the second, then it returns *some* value less than 0.
- *strrev()* reverses a string.
- *strcat()* joins two strings.
- A pointer variable can be assigned the address of an ordinary variable (Eg, PV=&V).
- A pointer variable can be assigned the value of another pointer variable (Eg, PV=PX) provided both pointers point to

object of the same data type.

- A pointer variable cannot be multiplied by a constant; two pointer variables cannot be added.
- On incrementing a pointer it points to the next location of its type.

## 6.13 FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.
3. Venugopal, K.R, Prasad, S.R: *Mastering C,* Tata McGraw-Hill publication.

## 6.14 ANSWERS TO CHECK YOUR PROGRESS

1. (i) True (ii) False (iii) False (iv) True (v)True (vi) True
2. (i) Homogeneous (ii) square bracket [ ]
   (iii) commas (iv) randomly
3. (d) All are correct
4. 16
5. (a) 6 (b) 9
6. (i) null (ii) string.h (iii) strcat() (iv) compares (v) characters
7. (c) Programming
8. (i) True (ii) True (iii) False (iv) True (v) False
9. (i) False (ii) False (iii) True (iv) False (v)True
10. (i) (b) int *ptr;
    (ii) (c)The variable pointed to by p
    (iii) (b) A variable for storing address
    (iv) (b)*(num+2);
    (v) (b) a=p;
11. Pointers are variables which hold the addresses of other variables. A compiler allocates an address at runtime for each variable and retains this till program execution is completed. Thus, entire

memory is not used at a time and only that part of memory is used that is required for execution.

## 6.15 MODEL QUESTIONS

1. Write a C program to find the largest and the smallest of n numbers.
2. Find the minimum element and the sum of array elements.
3. What is the header file for string library function. Describe five string library functions with there syntax and examples.
4. Write a program to compute the length of string.
5. Write a program to reverse a string without using strrev().
6. Write a program to check whether a given string is a palindrome or not. (Palindrome is a string which is same from both ends.)
7. What do you mean dynamic memory allocation? How it is useful?
8. If an integer array *text* is a stored from location 900 onwards, what is the location of the 8th element in the array.
9. Write a function to check whether characters +, -, * , / present in a string.
10. Write a function to swap two float variables using call by reference.
11. Write a C program to find the kth smallest elemnt in an array using pointers.
12. How are one dimensional arrays and two dimensional arrays represented using pointers?
13. Which of the following declarations can be invalid? Give reasons.
    (a) float array[5.3];
    (b) int num[50]
    (c) int m[5][ ];
14. Describe the following with some suitable examples:
    (a) Pointers and two dimensional array
    (b) Advantages and operations of
        (i) malloc() (ii) calloc() (iii) Any six string library functions.
15. Write a C program to add two matrices. Use function for the addition purpose.
16. Write a C program to multiply two matrices of same order.

*****

# UNIT- 7 STRUCTURE AND UNION

## UNIT STRUCTURE



## 7.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- use structure in a program
- learn how structures are defined and how their individual members are accessed and processed within a program
- declare structure variables
- learn about array of structures
- declare and use pointer to structure
- learn about union
- describe enumerated data type and type definition

## 7.2 INTRODUCTION

In the previous unit (i.e.,Unit 6) we have studied the array which is an

example of data structure. It takes simple data types like *int*, *char* or *double* and organises them into a linear array of elements. The array serves most but not all of the needs of the typical C program. The restriction is that an array is composed of elements all of which are of the same type. If we need to use a collection of different data type items it is not possible by using an array. When we require using a collection of different data items of different data types we can use a *structure*.

This unit will help you to learn about *structure* and *union*, giving values to members, initializing structure, functions and structures, passing structure to elements to functions, arrays of structure, structure within a structure and union.

## 7.3 STRUCTURE

A structure is a heterogeneous user defined data type. It is a convenient method of handling a group of related data items of different data types. A structure contains a number of fields or variables. The variables can be of any of the valid data types. In order to use structures, we have to first define a unique structure.

### 7.3.1 Defining a Structure

Let us consider for a moment a book in detail which records *title*, *author, page* and *price*. The book name (i.e., title) and author name would have to be stored as **string**, i.e., array of chars terminated with an ASCII null character, and the page and price could be **int** and **float** respectively. At the moment, the only way we can work with this collection of data is as separate variables. This is not as convenient as a single data structure using a single name and so the C language provides **struct**. General format of defining a structure is as follows:

```
struct tag_name
{
        data_type  member1;
        data_type  member2;
        ..............
        ..............
        data_type   membern;
};
```

In this declaration, ***struct*** is the keyword for structure; ***tag_name*** is the name that identifies sructure of particular type; and ***member1***, ***member2***,....***membern*** are individual member declarations. The individual members can be ordinary variables, pointers, arrays or other structures.

For example:

```
struct lib_books
{
        char title[25];
        char author[20];
        int pages;
        float price;
};
```

The keyword *struct* declares a structure to hold the details of four fields namely title, author, pages and price. These are members of the structures. Each member may belong to different or same data type. The *tag_name* can be used to define objects that have the tag names structure. It is not always necessary to define the structure within the *main()* function.

## 7.3.2 Structure Declaration

Once the composition of the structure has been defined, individual structure-type variable can be declared as followes:

**struct tag_name variable 1, variable 2, ....., variable n;**

where ***struct*** is a required keyword, ***tag_name*** is the name that appeared in the structure declaration, and ***variable 1***, ***variable 2***,....., ***variable n*** are structure variable of type ***tag_name***. We can declare structure variables using the tag_name any where in the program. For example, the statement,

```
struct lib_books book1, book2, book3;
```

declares *book1, book2, book3* as variables of type *struct lib_books* . Each declaration has four elements of the structure lib_books. The

complete structure declaration might look like this:

```
struct lib_books
{
        char title[25], author[20];
        int pages;
        float price;
};
struct lib_books book1, book2, book3;
```

Structures do not occupy any memory until it is associated with the structure variable such as book1. The t*ag_name* such as *lib_books* can be used to declare structure variables of its data type later in the program. The *tag_name* such as *lib_books* can be used to declare structure variables of its data type later in the program.

We can also combine both template declaration and variables declaration in one statement, the declaration

```
struct lib_books
{
        char title[20];
        char author[15];
        int pages;
        float price;
} book1,book2,book3;
```

is valid. The use of tag_name is optional.

*book1*, *book2*, *book3* declare *book1*, *book2*, *book3* as structure variables representing three books but do not include a tag_name for use in the declaration. A structure is usually define before ***main()***. In such cases the structure assumes global status and all the functions can access the structure. For example:

```
struct employee
{
        char fname[15];
        char lname[15];
        int id_no;
        int bmonth;
        int bday;
        int byear;
} emp1;
```

Here, we have declared one variable, ***emp1***, to be structure with six fields, some integers, some strings. Right after the declaration, a portion of the main memory is reserved for the variable ***emp1***. This variable takes a size of 38 bytes for different members of struct ***employee***: 15 bytes for fname, 15 bytes for lname, 2 bytes for id_no, 2 bytes for bmonth, 2 bytes for bday, 2bytes for byear.

## 7.3.3 Giving Values to Members

The members of structure themselves are not variables. They should be linked to structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' Which is known as ***dot operator*** or ***period operator***.

For example,

```
book1.price;
book1.pages;
book2.price;
```

*book1.price* is the variable representing the price of *book1* and can be treated like any other ordinary variable. We can use *scanf()* function to assign values as follows:

```
scanf("%f",&book1.price);
scanf("%d",&book1.pages);
```

We can also assign values to the members of the structure *lib_books*. If we want to assign some values to *book1*, then the statements will be like this:

```
strcpy(book1.title,”C Language”);
strcpy(book1.author,”Kanetkar”);
book1.pages=255;
book1.price=325.00;
```

**Program 1:** Reading information of one student and displaying those information.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct studentinfo
    {
        int roll;
        char name[20];
        char address[30];
        int age;
     }s1;
     clrscr();
    printf("Enter the student information:");
    printf("\nEnter the student roll no.:");
     scanf("%d",&s1.roll);
    printf("\nEnter the name of the student:");
     scanf("%s",&s1.name);
    printf("\nEnter the address of the student:");
     scanf("%s",&s1.address);
    printf("\nEnter the age of the student:");
     scanf("%d",&s1.age);
    printf("\n\nStudent information:");
    printf("\nRoll no.:%d",s1.roll);
     printf("\nName:%s",s1.name);
     printf("\nAddress:%s",s1.address);
    printf("\nAge of student:%d",s1.age);
    getch();
}
```

The members of a structure variable can be assigned initial values in the same way as the lements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general syntax is as follows:

storage_class struct tag_name variable={value1, value2,...., value n};

where value1 refers to the value of the first structure member, value2 refers to the value of the second member, and so on. A structure variable, can be initialized only if its strorage class is either *static* or *external*.

```
struct date
 {
      int month;
      int day;
      int year;
 };
struct employee
 {
      char empid[5];
      char name[30];
      char dept[20];
      struct date dob;
 };
static struct employee e = {A25, 'Rajib Dutta', 'Sales', 6, 21,85 };
```

Here, e is a static data structure variable of type employee, whose members are assigned initial values. The first member (empid) is assigned the character value A25, the second member (name) is assigned the character value Rajib Dutta, the third member (dept) is assigned the character value Sales. The fourth member is itself a structure that contains three members (month, day and year). And the last member of customer is assigned the integer value 6, 21 and 85.

**EXERCISE**

Q. Modify the above program (Program 1) for storing and displaying information of two strudents.

## 7.4 ARRAY OF STRUCTURES

A useful program may need to handle many records. If we need to store a list of items, we can use an array as our data structure. In this case, the elements of the array are structures of a specified type. For example:

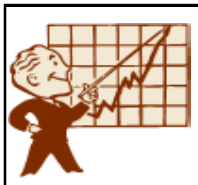
```
struct inventory
{
        int part_no;
        float cost;
        float price;
};
struct inventory table[4];
```

which defines an array with four elements, each of which is of type *struct inventory*, i.e., each is an inventory structure. Again, if we are maintaining information of all students in a school and if there are 100 students studying in the school, we need to use an array rather than single variables. It is possible to define an array of structures as shown in the program below :

**Program 2:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct studentinfo
    {
        int roll;
        char name[20];
        char address[30];
        int age;
```

```
    };
   struct studentinfo s[100];
/* s[100]  is an array of structure where information of maximum 100
   students can be stored */
    clrscr();
   int n,i;
   printf("\nHow many students information do you want to enter?");
    scanf("%d",&n);
   printf("Enter Student Information:");
    for(i=1;i<=n;i++)
    {
        printf("\nEnter Roll no.:");
        scanf("%d",&s[i].roll);
        printf("\nEnter the name of the student:");
        scanf("%s",&s[i].name);
        printf("\nEnter the address of the student:");
        scanf("%s",&s[i].address);
        printf("\nEnter the age of the student:");
        scanf("%d",&s[i].age);
    }
   printf("\n\nInformation of all studenst:");
    for(i=1;i<=n;i++)
    {
        printf("\nRoll no.:%d",s[i].roll);
        printf("\nName:%s",s[i].name);
        printf("\nAddress:%s",s[i].address);
        printf("\nAge of student:%d\n\n",s[i].age);
    }
    getch();
}
```



## CHECK YOUR PROGRESS

1. Define a structure consisting of two floating point members, called *real* and *imaginary.* Include the tag *complex* within the definition. Declare the variables c1,c2 and c3 to be structure of type *complex*.

2. Declare a variable ***a*** to be a structure variable of the following structure type ***struct account*** {

```
int ac_no;
char ac_type;
char name[30];
float balance;
};
```

and initiaze ***a*** as follows:
ac_no : 12437
ac_type: Saving
name: Rahul Anand
balance: 35000.00

3. State whether the following statements are true(T) or false(F)

(i) Collection of different datatypes can be used to form a structure.
(ii) Structure variables can be declared using the tag name any where in the program.
(iii) Tag-name is mandatory while defining a structure.
(iv) A program may not contain more than one structure.
(v) We cannot assign values to the members of a structure.
(vi) It is always necessary to define the structure variable within the main() function.

## 7.5 STRUCTURE WITHIN A STRUCTURE

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures. For example,

```
struct date
{
        int day;
        int month;
        int year;
};
struct student
{
        int roll;
        char name[20];
        char combination[3];
        int age;
        structure date dob;
}student1,student2;
```

the sturucture ***student*** constains another structure ***date*** as one of its members.

## 7.6 PASSING STRUCTURES TO FUNCTIONS

Structure variables may be passed as arguments and returned from functions just like other variables. A structure may be passed into a function as individual member or a separate variable. A program example to display the contents of a structure passing the individual elements to a function is shown below :

**Program 2:**

```
#include<stdio.h>
#include<conio.h>
void display(int,float);
void main()
{
```

```
    struct employee
    {
                        int emp_id;
                    char name[25];
                    char department[15];
                    float salary;
    };
   static struct employee e1={15, "Rahul","IT",8000.00};
    clrscr();
   /* only emp_id and salary are passed to the display fucntion*/
   display(e1.emp_id,e1.salary);
    getch();
}

void display(int eid, float s)
{
        printf("\n%d\t%5.2f",eid,s);
}
```

**Output :** 15 8000.00

When we call the display function using *display(e1.emp_id,e1.salary);* we are sending the emp_id and name to function display( ); it can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time.

**Passing entire structure to functions :**

There may be some structures having numerous structure members (elements). Passing these individual elements would be a tedious task. In such cases we may pass the whole structure to a function as shown below :

**Program 3 :**

```
#include<stdio.h>
#include<conio.h>
struct employee
{
        int emp_id;
```

```
        char name[25];
        char department[10];
        float salary;
};
static struct employee e1={12,"Dhruba","Sales",6000.00};
void display(struct employee e);  //prototype decleration
void main()
{
        clrscr();
        display(e1);    /*sending entire employee structure*/
        getch();
}
void display(struct employee e)
{
printf("%d\t%s\t%s\t%5.2f", e.emp_id,e.name,e.department,e.salary);
}
```

**Output :**

12 Dhruba Sales 6000.00

**Program 4:** A program using structure working within a function

```
#include<stdio.h>
#include<conio.h>
struct item{
        int code;
        float price;
};
struct item a;
void display(struct item i);  //prototype decleration
void main()
{
    clrscr();
   display(a);    /*sending entire employee structure*/
   getch();
}
void display(struct item i)
{
   i.code=20;
```

```
   i.price=299.99;
   printf("Item Code and Price of the item:%d\t%5.2f", i.code,i.price);
}
```

**Output :** 20 299.99

## 7.7 POINTER TO STRUCTURES

Instead of passing a copy of the whole structure to the function, we can pass only the address of the structure in the memory to the function. Then, the program will get access to every member in the function. This can be achieved by creating a pointer to the address of a structure using the indirection operator "*".

To write a program that can create and use pointer to structures, first, let us define a structure:

```
struct item
{
        int code;
        float price;
};
```

Now let us declare a pointer to struct type *item*.

```
struct item *ptr;
```

Because a pointer needs a memory address to point to, we must declare an instance of type *item*.

```
struct item p;
```

The following figure shows the relationship between a structure and a pointer.

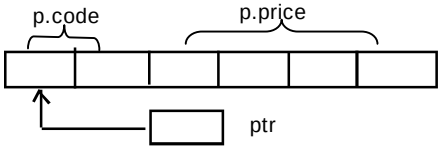



Fig.7.1: A pointer to a structure points to the first byte of the structure

**Program 5:** Program to demonstrate pointers to structure.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        struct item
        {
                int code;
                float price;
        };
        struct item i;
        clrscr();
        struct item *ptr;          //declare pointer to ptr structure
        ptr=&i;                    // assign address of struct to ptr
        ptr->code=20;
        ptr->price=345.00;
        printf("\nItem Code: %d",ptr->code);
        printf("\tPrice: %5.2f",ptr->price);
        getch();
}
```

**Output :** Item Code: 20 Price: 345.00

## CHECK YOUR PROGRESS

4.State whether the following statements are true(T) or false(F)

(i) It is possible to pass a structure to a function in the same way a variable is passed.

(ii)When one of the fields of a structure is itself a structure, it is called nested structure.

(iii) We cannot create structures within structure in C.

(iv) It is illegal for a structure to contain itself as a member.

(v) A sstructure can include one or more pointers as members.

5. Fill in the blanks:

(i) _____ can be used to access the members of structure variables.

(ii) The name of a structure is referred to as _____ .

## 7.8 UNION

In some applications, we might want to maintain information of one of two alternate forms. For example, suppose, we wish to store information about a person, and the person may be identified either by name or by an identification number, but never both at the same time. We could define a structure which has both an integer field and a string field; however, it seems wasteful to allocate memory for both fields. (This is particularly important if we are maintaining a very large list of persons, such as payroll information for a large company). In addition, we wish to use the same member name to access identity the information for a person.

C provides a data structure which fits our needs in this case called a *union* data type. A union type variable can store objects of different types at different times; however, at any given moment it stores an object of only one of the specified types. Unions are also similar to structure datatype except that members are overlaid one on top of another, so members of union data type share the same memory.

The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure template. For example, we can declare a union variable, person, with two members, a string and an integer. Here is the union declaration:

```
union human
{
        int id;
        char name[30];
 } person;
```

This declaration differs from a structure in that, when memory is allocated for the variable person, only enough memory is allocated to accommodate the largest of the specified types. The memory allocated for person will be large enough to store the larger of an integer or an 30 character array. Like structures, we can define a tag for the union, so the union template may be later referenced by the tag name:

Unions obey the same syntactic rules as structures. We can access

elements with either the dot operator ( . ) or the right arrow operator (->). There are two basic applications for union. They are:

(i) Interpreting the same memory in different ways.

(ii) Creating flexible structure that can hold different types of data.

**Program 6**: Program demonstrating initializing the member and displaying the contents.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    union data
    {
        int a;
        float  b;
    };
    union data d;
    d.a=20;
    d.b= 195.25;
    printf("\nFirst member is %d",d.a);
    printf("\nSecond member is %5.2f",d.b);
    getch();
}
```

Output: First member is 16384
Second member is 195.25

here only the float values are stored and displayed correctly and the interger values are displayed wrongly as the union only holds a value for one data type of the larger storage of their members.

## 7.9 ENUMERATED DATA TYPES

In addition to the predefined types such as int, char,float etc. , C allows us to define our own special data types, called enumerated data types.

An enumeration type is an integral type that is defined by the user . The syntax is:

**enum typename {enumeration_list};**

Here, *enum* is keyword, type stands for the identified that names the type being defined and *enumerator list* stands for a list of identifiers that define integer constants. For example:

enum color {yellow, green, red, blue, pink};

defines the type *color* which can then be used to declare variables like this:

color flower=pink;
color car[ ]={green, blue, red};

Here, *flower* is a variable whose value can be any one of the 5 values of the type *color* and is initialialized to have the value pink.

**Program 7:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
   enum month { jan, feb, mar, apr, may, jun, jul, aug, sep,
    oct, nov, dec };
    month m;
    clrscr();
    for(m=jan;m<=dec;m++)
                    printf("%d\t", m+1);
     getch();
}
```

**Output :** 1 2 3 4 5 6 7 8 9 10 11 12

In the above declaration, *month* is declared as an enumerated data type. It consists of a set of values, jan to dec. Numerically, jan is given the value 1, feb the value 2, and so on. The variable *m* is declared to be of the same type as month, m cannot be assigned any values outside those specified in the initialization list for the declaration of month.

## 7.10 DEFINING YOUR OWN TYPES (TYPEDEF)

Using the keyword ***typedef*** we can rename basic or derived data types giving them names that may suit our program. A typedef declaration is a declaration with typedef as the storage class. The declarator becomes a new type. We can use typedef declarations to construct shorter or more meaningful names for types already defined by C or for types that we have declared. Typedef names allow us to encapsulate implementation details that may change.

A typedef declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type. For example:

**typedef unsigned long int ulong;**

The new type (***ulong***) becomes known to the compiler and is treated the same as ***unsigned long int***. If we want to declare some more variables of type unsigned long int, wwe can use the newly defined type as :

**ulong distance;**

We can use the typedef keyword to define a structure as foln be lows:

```
typedef struct
{
   type member1;
   type member2;
   ....
}type_name;
```

type_name can be used to declare structure variables as follows:

**type_name variable1,variable2,...;**

## CHECK YOUR PROGRESS

6. State whether the following statements are true(T) or false(F).

(i) Union contains members of different data types which share the same storage area in memory.

(ii) The ⟶ operator can be combined with the period operator to access a member of a structure that is itself a member of another structure.

(iii) The keyword typedef is used to define a new data type.

(iv) A structure is a collection of data items under one name in which the item shares the same storage.

7. Choose the correct answer for the declaration:

```
typedef float height [100];
 height men,women;
```

(a) define men and women as 100 element floating point arrays.
(b) define men and women as floting point variables
(c) define height, men and women as floating point variables.
(d) All are illegal.

## 7.11 LET US SUM UP

- Structure is a data type used for packing logically related data of different types. Structure declaration includes the following elements: The keyword struct, the structure tag name, List of variable names separated by commas and the terminating semicolon.

- C permits the use of arrays as structure members.

- Unions are concept borrowed from structures and therefore they follow the same syntax.

- In structure, each member has its own storage location, where as all members of a union use the same location.

- C allows us to define our own special data types, called enumerated data types.

- The keyword *typedef* is used to define a new data type of our own choice. We can use the typedef keyword to define a structure.

## 7.12 FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.

## 7.13 ANSWERS TO CHECK YOUR PROGRESS

1.

```
struct complexo
{
        float real, imaginary;
};
struct complex c1,c2,c3;
```

2.

static struct account a={12437, “Saving”, “Rahul Anand”, 35000.00};

(*a* is a static structure variable of type account, whose members are assigned initial values.)

3. (i) True (ii) True (iii) False (iv)False (v) False (vi) False

4. (i) True (ii) True (iii) False (iv) True (v) True

5. (i)Pointers (ii)tag name

6.(i) True (ii) True (iii)True (iv) False

7 (a)

## 7.14 PROBABLE QUESTIONS

1. What is a structure? How is a structure different from an array?

2. How is structure declared? Define a structure to represent a date.

3.What is meant by array of structure?

4.How are the data elements of a structure accessed and processed?

5. Write a program in C to prepare the marksheet of a college examination and the following items will be read from the keyboard.
   Name of the student,
   Subject name,
   Internal marks,
   External marks
Prepare a list separately of those students who failed and passed in the examination.

6.What is meant by union?Differentiate between structure and union.

7. What is the purpose of typedef feature? How is this feature used with structure?

8. Write short notes on:
   (a) Enumerated data type
   (b) Type definition.

*****

# UNIT- 8 FILE HANDLING

## UNIT STRUCTURE



## 8.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- learn about various file operation
- read data from a file
- write data from a file
- describe various input/output function for file operations
- write programs that performs various operations on file

## 8.2 INTRODUCTION

In our previous unit, while discussing various programs, we have already used *scanf()*, *printf()* functions to read and write data. These functions are console oriented input/output (I/O) functions. For such functions there is always a need of a keyboard for reading inputs and a monitor to display the output. This works fine as long as the input/output data is small enough to read and write.

However in some situations, huge amount of data are to be read and write and the console oriented I/O function could not handle it efficiently. Because the entire data is lost when the program is terminated or the computer is terminated or the computer is turned off. It is, therefore, necessary to have a more flexible approach where large amount of data can be stored permanently in disks and read them whenever required.This method brings the concepts of file to store and handle the data easily. In this unit we are going to discuss handling of files in C.

A ***file*** is a place on the disk where a group of related data is stored. C language supports a number of functions that have the ability to perform the basic file operations:

- naming a file
- opening a file
- reading data from a file.
- writing data to a file.
- closing a file.

There are two ways to perform the file operation in C:

a) Low level Input/Output
b) High level Input/Output

The low level disk I/O functions are more closely related to the computer's operating system and it uses UNIX system calls.It is harder to program for general users as compared to the high level I/O. However lower level I/O functions are more efficient both in terms of operation and the amount of memory used by the program.

But high level I/O functions are more commonly used in C programs and are easier to use than low level I/O functions. We are here going to discuss high level I/O functions. Before we discuss let us take an overview of Disk I/O functions:

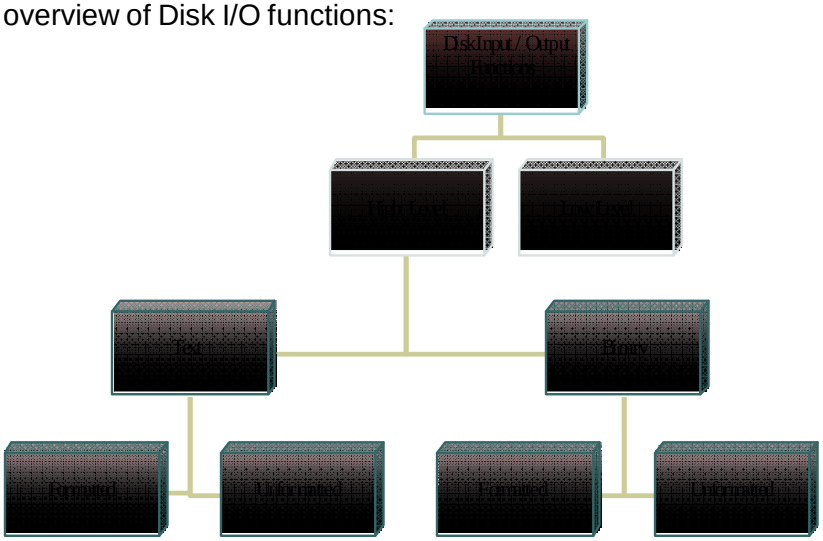



**Fig:1**

The High level input/output function includes the following which will be discussed in the next sectons broadly:

**i) fopen():** Used for opening an existing file. Also used to create a new file.
**ii) fclose():** Close a file which has been already opened.
**iii) getc():** Reads single character from a fie.
**iv) putc():** Writes single character to the file.
**v) fputs():** Writes strings to the file.
**vi) fgets():** Read string from file
**vii) fprintf():** Write a set of values to a file.
**viii) fscanf():** Read a set of values from a file.

## 8.4 OPENING AND CLOSING FILE

**OPENING A FILE:**

With the use of file we can store data to the secondary memory permenantly. But before going to store or retrieve data from file we need to learn some other information which includes:

a) naming the file (i.e., file name)
b) data structure that links the file.
and c) purpose of opening the file.

**Byte**:
*Group of 8 bit means 1 byte i.e., 1byte = 8 bit. where 1 bit means either 0 or 1.*

a) File name is a string or group of characters. It may contain two parts: a primary name and an optional period with the extension. Some example of some file names are:

i. add.c ( Here **add** is primary part and **.c** is the extension)
ii. kkshou.c
iii. output.txt

b) Data structure of a file is defined as **FILE** and we may call **FILE** as one kind of data type. Note that all files should be declared as type FILE before they are used.

c) Since we are opening a file using **fopen()** we should have clear idea about the file and the basic purpose of opening the file. These purposes include: reading or writing or both on the file. We may add data to an existing file also.
General Syntax for opening a file:

```
FILE *fptr;
fptr=fopen("filename","mode");
```

The first statement declares the variable **fptr** as a pointer to the data type FILE.The second statement opens the file named *filename* and asigns the identifier to the FILE type pointer **fptr**;*mode* specifies the purpose of opening the file. Depending upon the mode, a file can be used for a specific purpose as listed below.

| Mode | Meaning |
|---|---|
| "r" | Open an existing file for reading purposes |
| "w" | Open a new file for writing only. If a specified file name already exists, it will be destroyed and a new file is created in its place. |
| "a" | Open an existing file for adding new data at the end of a file. New file is created if file does not exist. |
| "r+" | Open a file both for reading and writing |
| "w+" | Open a file both for reading and writing. If file name already exists, it will be destroyed and a file is created in its place. |
| "a+" | Open a file for both reading and appending. New file is created if the file does not exist. |

Return value:

On successful execution, fopen() returns a pointer to opened stream. As shown in above, the return value is assigned to **fptr**. If the specified file could not be opened successfully, then it returns **NULL**; fopen() sometimes may fail in opening the file with the specified mode because of some unwanted reason. We can easily check it by comparing value of **fptr**. This is illustrated with the following program segments:

```
FILE *fptr;
fptr=fopen("data.txt","r")
if(fptr==NULL)
        printf("Error in opening the file ");
else
        printf("Sucessfully opening the file ");
```

If ***data.txt*** does not exist already, then the value of *fptr* will be NULL and the output will be:

Error in opening the file

**CLOSING FILE**

It is good a programming practice to close the file using fclose() as soon as all operations on the file have been completed. It ensures that all the information associated with the file is cleared out from the *buffer,* thus preventing any accidental misuse of the file.Closing the file releases the space taken by the FILE structure which is returned by the fopen(). General syntax of fclose() is :

```
fclose(file_pointer)
```

## 8.5 INPUT/OUPUT OPERATION ON FILES

**a) getc() and putc():**

The function **fopen()** can be used for opening a file for reading or writting purposes.Suppose we need to read data from a file; then obviously, we need a mechanisim to read the content of the file. Function **getc()** is used to read the content of a file which is opened by the *fopen()* function. A file pointer is associated with *fopen()* function after successful opening of the file which always points the first character of a file.

For example, let us consider the following statement:

```
ch= getc(fptr)
```

In the above statement, *getc()* reads a character from the current position of the file, incremens the pointer position so that it points to the next character. Then it returns the character that is read and is collected in the variable **ch**.

One question may arise in our mind that while reading from the file continously using loop, how can we determine that file has been completly read or we have reached the end of the file. Generally, we can say the file has reached its end, if there is no character to be read. The function ***getc()*** returns **EOF** or **-1** when the end of file is detected. So we can check it by comparing the value of **ch** with **EOF**.

Similarly, **putc()** can be used for writting data to a file character by character. The file must be opened in *write* mode before writing to the file using *fopen()*. For example,

```
putc(ch,fptr)
```

where **ch** is character to be write to the file refferenced by the file pointer **fptr**.

**Example1**: A file named "*kkhsou.org*" some contents are already written. Write a program to read the content of this file and display it.

**Solution:** We are already given a file *kkhsou.org* and our main purpose of writing the program is to read its contents and display it to the monitor screen. So, at first we have to open the file in reading mode and then read it using *getc()* until the end of file(EOF) is encountered. While reading from the file, we need to display it simultaneously to the monitor. After that we need to close that file using *fclose()*. So the program will look like:

**Program1:**

```
#include<stdio.h>
#inlcude<conio.h>
void main()
{
        FILE *fptr ;          // File pointer declaration
         char ch;
```

```
 fptr=fopen("kkshou.org","r");   // file open for reading
if(fptr==NULL)         // check the file is opened or not
     printf("\n Error in opening file");
 else
 {
      do
      {
            ch=getc(fptr);
           printf("%c",ch);  // display to the monitor screen
         }while(ch!=EOF);
       }
   fclose(fptr);   // closes the file
  getch();
}
```

**Example2:** Write a program to read your name, roll number from the keyboard and then write it to a file "address.txt ".

**Solution**: Here we are asked to read our name, roll number directly from the keyboard and then we need to write it to the file *address.txt*. So, we should read name and roll number using *scanf()* or *getchar()* or can use other input function.We can use any one of these. After reading it, we need to open the file *address.txt* in write mode since we are going to write to the file. So the program will look like:

**Program2**:

```
#include<stdio.h>
#iinclude<conio.h>
void main()
{
        FILE *fptr;
        char ch;
        fptr=fopen("address.txt","w");  // open  file  writting
        if(fptr==NULL)     // checking whether file is
                           //succesfully opened or not
              printf("\n Error in opening file");
        else
        {
              printf("\nEnter Your Name and Roll Number");
```

```
    do{
            ch=getchar();   // read from the keyboard
                            //character by character
            putc(ch,fptr);   // write it to address.txt refferenced by fptr
       }while(ch!='\n');  // untill  we press ENTER key (newline)
  }
 fclose(fptr);
 getch();
}
```

**Output:**

Enter Your Name and Roll Number **Nayan 4** [press enter ]

Nayan and 4 is then written to the file *address.txt* and we can then open it directly to view the output. Some of us may be confused about the '\n' that is used with while() condtions. Basically in C language '\n' is used to denote a new line character. In the above program when we get '\n' in gecthar() then we stop the loop,thinking that user has entered his data copmpletely. However, there is another way of reading more than one line.

**Example3**: Write a program to copy a file "first.txt " to another file "second.txt". Assume that file *first.txt* already exists.

**Solution**: The file *first.txt* aready exists and we need to write a program that will copy the content of *first.txt* to *second.txt.* So we need to open the *first.txt* file in *read* mode, then *second.txt* file in *write* mode. Program will look like:

```
#include <stdio.h>
#include <conio.h>
void main()
 {
       FILE *fp1,*fp2;  //  Since we need two file
        char ch;
       fp1=fopen("first.txt","r");
       fp2=fopen("second.txt","w");
      // checking whether two file succesfully opend or not
       if(fp1==NULL && fp2==NULL)
              printf("\n Error in Opening the File");
```

```
    else
     {
          do
          {
               ch=getc(fp1);   // read from the first file
               putc(ch,fp2);    // write to the second file
          }while(ch!=EOF);
    }
    fclose(fp1);
    fclose(fp2);
   getch();
}
```

To view the output we need to open the file second.txt from the menu.

**b) fputs() and fgets():**

In the previous section we have already discussed how to read from the files and also how to write to the file. But we are reading or writting to the files only character by character using loops.We can also read or write to the files as strings using *fputs()* and *fgets()*. We may compare it to the previous strings functions *gets()* and *fputs()* with a very few difference.These functions are related to only key-boards input/output, whereas *fgets()*,*fputs()* is related to files. The *fputs()* writes a string to a file. For example, say if **fptr** is a file pointer which open a file for writting, then :

```
fputs("Well Come to KKHSOU", fptr) ;
```

The sentence "Well Come to KKHSOU" is written to a file pointed by fptr. Also we can use fputs() in following manner:

```
char str[10] ;
gets(str) ;       // read from the keyboard
fputs(str,fptr);  // write to the file
```

So, the general way of representing fputs() is:

```
fputs(strings, file_pointer);
```

fgets() can be used for reading strings from a file. General way of representing fgets() is:

```
fgets(strings,no_bytes,file_pointer);
```

The statement reads a strings having (no_bytes-1) from a file pointed by file_pointer. no_bytes indicates how many bytes strings to be. We give here -1 cause file always read from starting 0 for first character.

**Example 4:** Write a program to read a line from the keyboard and then write it to a file *output.txt.*

**Solution**: Here we are asked to read a lines using keyboard, so we need a character array to store the line.Then we have to open a file *output.txt* in write mode and then write the lines (character array) to it. The program should be:

**Program4**:

```
#include<stdio.h>
#include<conio.h>
        void main()
         {
            char line[80];
            FILE *fp;
           fp=fopen("output.txt","w");
           if(fp==NULL)
              puts("Error in opening the file ")
            else
              {
               puts("Enter a line ");
               gets(line);  // read from keyboard
               fputs(line,fp);  // write to the file
               }
             fclose(fp);
            getch();
          }
}
```

**Output:**

Enter a line **Hello learners** (press enter key )

The text *Hello learners* given by you from the keyboard is written to the file output.txt. We can see it by opening the file output.txt directly.

**Program5:** Write a program to read from a file then display it to the monitor screen.

**Solution**:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   FILE  *fp;
   char strngs[80];
   fp=fopen("output.txt","r");
   if(fp==NULL)
      puts("Error in opening file");
   else
    {
      while(fgets(strngs,79,fp)!=NULL)
        puts(strngs);
    }
   fclose(fp);
}
```

In the above program we have read from the existing file and then display it to the output screen i.e., monitor. So we open an existing file *output.txt* assuming that the file exists with the content. Then uses fgets() functon to read the file. We used 79 size in fgets() since in a line of a file, it contains maximum 80 character. After reading from the file we store it **strngs** array using fgets() and then display it to the output screen with puts().

## CHECK YOUR PROGRESS

1. State whether the following statements are true or false:

   a) fopen() is used to open and closing a file.
   b) File concept is used mainly because to store data permanently and use them later.
   c) fopen("file.c","w") open file.c for reading purposes only

2. Fill in the blanks:
   a) putc() is used to __ to a file.
   b) fgets() is mainly used for ___ strings.
   c) fopen() returns __ on error.

## c) fprintf() and fscanf():

Already we are discussing how to read or write a single character from the file using getc(), putc() and also about reading and writing the strings using fgets() and fputs() to the files. All these functons are related to particular type data i.e. characters. For wrtting and reading data to the files simultanously having different data type variables we can use fprintf() and fscanf() respectively. These functions are mostly same as printf() and scanf() function earlier used except that fprintf() and fscanf() are used for writing and reading from the **files** only. The general form of fprintf() is:

**fprintf**(fptr,"control string",list);

where fptr is a file pointer associated with the file opened for writting.The control string contains output specification for the items in the list.You will get a clear idea of control string when you go for the examples.The list may include variables,constants and strings. For example,

**fprintf**(fptr,"%s,%d,%f",name,roll,per);

where name is an array variable of type character and roll is integer variable and per is float variable.These different data type variable is written to the file pointed by *fp*. The general format of fscanf() is:

**fscanf**(fptr,"control string",list);

The above statement causes the reading of the items in the list from the file specified by the fptr, according to the specifications contained in the control string. For example,

**fscanf**(fptr,"%s,%d,%f",name,&roll,&per);
name, roll and per is read from the file specified by fptr.

**Example**:
a) Write a program to read name,roll number from the keyboard for a student and then write to a file using fprintf().
b) Then use fscanf() to read the from the file written by the above program.

**Solution**: a) So first we need to read name and roll number from the keyboard and then open a file in write mode. After that we write name, roll number to the file using fprintf().

**Program6:**

```
#include<stdio.h>
 void main ()
{
        char name[10];
         int roll_no;
        FILE *fptr;

         /*Open the file */
         fptr=fopen("out.txt","w");
        if(fptr==NULL)
                printf("Error in opening file ");
        else
        {
                  /*Read from the keyboard */
                  printf("Enter name ");
                 scanf("%s",name);
                printf("Enter roll no ");
               scanf("%d",&roll_no);

                 /* Write to the file */
                 fprintf(fptr,"%s %d",name,roll_no);
        }
        fclose(fptr);
 }
```

Because of the above program the name and roll number given by the user is written to the file **out.txt** . To see the result we need to open the file by going to file -> open menu from the C program software.

b) Here we assume that name and roll number are already written to the file i.e. data exist previously. Next we need to open the file in read mode and then read these data using fscanf(). After that we may display it using printf() function in order to view the result.

Program7:

```
#include<stdio.h>
void main()
 {
        char name[10];
        int roll_no;
        FILE *fptr;
        fptr=fopen("out.txt","r");
        if(fptr==NULL)
                printf("Error in opening file ");
        else
        {
                /* Read from  file */
                 fscanf(fptr,"%s %d",&name,&roll_no);
                  /* Write to the screen */
                  printf("\nName: %s Roll No:: %d",name,roll_no);
         }
         fclose(fptr);
}
```

## 8.6 SEEKING FORWARD AND BACKWARD

Already we have discussed reading / writing files from the start ing position. But sometimes we need a situations where we have to read or write files randomly. Some basic functons that can be used for reading or writing files randomly are explained below briefly:

a) **ftell():** This function gives the current fille pointer positions in a particular file.

General Syntax: **n = ftell ( fptr )**

The return value of **n** is in bytes which indicates that **n** bytes have already been read or written.

b) **rewind():** The rewind() function is used to resets positions of file pointer to the start of the file. For example,

```
rewind(fptr);
 n=ftell(fp)
```

because of the rewind() function file pointer position is set to the starting of the file and therefore the value of **n** would be 0. Remember that first byte in the file is numbered as 0, second as 1 and so on.

c) **fseek():** This function is used to move the file position to a desired location within the file. General syntax:

**fseek ( file_ptr, offset , position);**

Here **file_ptr** denotes the file to be processed, **offset** is a variable of type long which specifies the number of position to be moved from the location specified by the **position.** The value of **position** can be any one the follwing:

| Value | Meaning |
|---|---|
| 0 | Begining of the file. |
| 1 | Current position |
| 2 | End of file |

The value of **offset** may be positive which means that file pointer to be moved forwards or negative meaning move back words. It would be clear for us if we go for the examples.Now see the follwing and try to understand.

| Statement | Meaning |
|---|---|
| **fseek ( fptr, 0L,0)** | Go to the begining of the file. |
| **fseek ( fptr, 0L,1)** | stay at the same i.e current position. |
| **fseek ( fptr, 0L,2)** | Go to end of the file. |
| **fseek ( fptr, m,0)** | Move m+1 bytes from begining. |
| **fseek ( fptr, m,1)** | Go forward by m bytes from current positions **.** |
| **fseek ( fptr, -m,1)** | Go backward by m bytes from current positions. |
| **fseek ( fptr, -m,2)** | Go backward by m bytes from end. |

fseek() return Zero after successful operations otherwise it returns -1 like fopen().

CHECK YOUR PROGRESS

3. Write one difference between fprintf and fputs.Also write the syntax of both function.
4. How can you get the current file pointer position of a file?
5. Write the putpose of rewind() function?
6. Using fseek() how can you move the fie pointer t o the end of a file and to the beginning of a file.

## 8.7 LET US SUM UP

- File concept is generally used to store data permanantly and use it later.Operations include on files are: Opening, Reading, Writting Closing etc.
- By specifying the modes in fopen() function we can treat the file in various ways such as: file can read (e.g **r**) or write(e.g **w**) or both(e.g **r+**) etc.

- If the specified file does not exist or there is error in the file while opening then fopen() returns NULL.

- putc(), fputs() and fprintf() is used mainly for reading data from a file pointed by a file pointer. Similarly getc(),fscanf(),fgets() is used for reading information from the file.

## 8.9 FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.

## 8.8 ANSWERS TO CHECK YOUR PROGESS

CHECK YOUR PROGRESS

1a) False. Since fopen() used only for opening file only; not for closing. b) True. c) False. For writting purpose it is used since it has **w** .
2. a) write a character b) reading c) -1

3. Ans: fputs is used to write strings to a particular a file. however fprintf() is used to write various types of data to a particular file.

```
fprintf(fptr,"%s,%d,%f",name,roll,per);
fputs("Well Come to KKHSOU", fptr) ;
```

Here fptr is the file pointer.

4. using ftell ( fptr ).

Here fptr is the file pointer.

4. using ftell ( fptr ).

5. The rewind() function is used to resets positions of file pointer to the start of the file.

6. To move the file pointer to the end of a file:fseek ( fptr, 0L,0)

To move the file pointer to the start of a file: fseek ( fptr, 0L,2)

## 8.10 MODEL QUESTIONS

1. Explain the functions of fopen() and fclose() with one example each.
2. Write the name of two input functions for the file.
3. What is the main difference between getc() and fgets() functions.
4. Mention the various modes used with the fopen() functions.
5. Explain briefly: ftell(), rewind(), fseek().
6. Write a program
   a) To read your name, roll and percentage from the keyboard and then write it to a file.
   b) To copy a file to another file using getc().
   c) To count number of vowel, blank space from an existing file.