PG MA S1- 04 (DSE)

TWO YEAR MASTER'S DEGREE COURSE

SEMESTER - I

Computer Programming using C

MATHEMATICS

Discipline Specific Elective Course
MASTER OF SCIENCE IN MATHEMATICS

Hiranya Chandra Bhuyan School of Science and Technology

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakoti Mahanta, Gauhati University
Prof. Hiren Kumar Deva Sarma,Gauhati University
Dr. Lipi B Mahanta, IASST, Guwahati
Dr. Sanjib Kr. Kalita, Gauhati University

Course Coordinator :

Dr. Tapashi Kashyap Das, KKHSOU

SLM Preparation Team

Units	Contributors
1,4,6,11,13	Dr. Tapashi Kashyap Das, KKHSOU
2,3,7	Arabinda Saikia, KKHSOU
5,12	Dr. Sanjib Kr. Kalita, Gauhati University (Former Faculty of KKHSOU)
8,14,15	Binod Deka, KKHSOU
9,10	Dr. Sruti Sruba Bharali, KKHSOU

Editors

Content : Dr. Tapashi Kashyap Das, KKHSOU (Unit 2,3,5,7,9,10,12,15)
Dr. Sruti Sruba Bharali, KKHSOU (Unit 1,4,6,8,11,13,14)

Language Editor: Prof.(Retd.) Robin Goswami, Cotton College

Structure, Format & Graphics: Dr. Tapashi Kashyap Das, KKHSOU

November, 2024

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State University is made available under a Creative Commons Attribution-Non Commercial-ShareAlike4.0 License (International) : http.//creativecommons.org/licenses/by-nc-sa/4.0

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

Headquarters: Patgaon, Rani Gate, Guwahati-781017
City Office: Resham Nagar, Khanapara, Guwahati-781022 Web: www.kkhsou.in

MASTER OF ARTS /MASTER OF SCIENCE IN MATHEMATICS

COMPUTER PROGRAMMING USING C

DETAILED SYLLABUS

		Page No
UNIT 1:	**Introductory Concepts** Basic definition of Algorithm, Pseudocode, Flowchart.	**7 - 28**
UNIT 2:	**Elements of C Programming** Characters used in C, Tokens, Identifiers, Keywords (Reserved words), Constants,Variables.	**29 - 36**
UNIT 3:	**Variables and Data types** Basic Data types in C : Integer, Character, Floating point, String, Variables and its declarations, Symbolic Constants.	**37 - 45**
UNIT 4:	**Operators and Expressions** Different types of Operators : Arithmetic, Relational, Logical, Assignment, Conditional, Increment and Decrement, Bitwise, Special Operator (size of, comma etc), Precedence and Associatively of Operators, Expression in C, Type Casting.	**46 - 73**
UNIT 5:	**I/O Functions** Header Files (stdio.h, conio.h, math.h etc.), Formatted Input/Output Functions (scanf, printf etc.), Control Strings used in printf() and scanf() functions, Escape Sequences, Unformatted Input/Output Functions (getch, getchar, putchar, gets, puts, getche, clrscr etc.)	**74 - 86**
UNIT 6:	**Conditional Statements** Decision Control Statements, Conditional Statement-if, if-else, nested if-else, switch-case; break, continue, goto; Conditional Operator Statement.	**87 - 104**
UNIT 7:	**Loop Control Structures** Concept of Loops, Types of loop: while, do-while, for;	**105 - 118**

UNIT 8: Storage Class **119 - 128**

Storage Class, Automatic, External, Static, Register.

UNIT 9: Arrays **129 - 145**

Array, Array Declaration, 1-Dimensional array, 2-Dimensionalarray.

UNIT 10: Strings **146 - 162**

String, String Handling Functions: strlen(), strcmp(), strcpy(), strrev(), strcat(), etc.

UNIT 11: Functions **163 - 188**

Function, Function Declaration, Function Definition, Function C all, Formal and Actual Parameter, Recursive Function.

UNIT 12: Pointers **189 - 203**

Pointer, Pointer Declaration, Passing Pointer to a Function, Pointer and One-Dimensional Arrays, Dynamic Memory Allocation.

UNIT 13: Structure and Union **204 - 226**

Structure Declarations, Definitions, Array of Structure, Structure within a Structure, Pointer to Structure, Union Declaration, Definition, Enumerated Data Types.

UNIT 14: File Handling **227 - 244**

Concept of File, File Pointer, File System, File Opening in various modes, Closing a file, Reading and Writing on files, Formatted Input/Output, fseek(), ftell() etc.

UNIT 15: Preprocessor Directives **245 - 254**

Preprocessor Directives, Macro, Macros with arguments, #if and #elif directives

COURSE INTRODUCTION

"Computer Programming Using C" is the first Discipline Specific Elective course of MSc(Mathematics) programme. Since computers cannot understand human languages, special programming languages are designed for this purpose. C is one of the most popular programming languages. It is used in several different software platforms such as system software and application software. C language was developed in the early 1970s by *Dennis Ritchie* at Bell Laboratories, USA.

This course introduces Computer Programming using C language. The objective of this course is to introduce the learners to the C programming language and enable them to apply these concepts for solving problems. Programming is a skill best developed by rigorous practice. A few other programming languages are such as C++ and Java are also based on C. Hence, mastering the C language is good for learning such languages.

C language uses a compiler as its translator to translate or compile the complete C program. A linker is used to link the input (usually keyboard) and output (usually monitor) devices and generate an executable program from an object program. On executing the executable program, user is allowed to input values and get the output. C language commonly uses a Turbo editor in MS-DOS system and VI editor in UNIX system.

Each unit 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: "LETUSKNOW". 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 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.

UNIT 1: INTRODUCTORY CONCEPTS

UNIT STRUCTURE

1.1 Learning Objectives

1.2 Introduction

1.3 Algorithm

- 1.3.1 Conventions used in Writing Algorithm
- 1.3.2 Method for Developing an Algorithm

1.4 Pseudocode

1.5 Flowchart

- 1.5.1 Symbols of Flowchart
- 1.5.2 Advantages and Limitations of Flowchart

1.6 Let Us Sum Up

1.7 Further Reading

1.8 Answers to Check Your Progress

1.9 Model Questions

1.1 LEARNING OBJECTIVES

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

- define algorithm
- learn the conventions used in writing algorithms
- develop algorithm for various computational problems
- define pseudocode
- write pseudocode for different problems
- pictorially represent algorithm in the form of flowchart
- learn different symbols of flowchart
- draw flowcharts for computations, decision making, loops etc.

1.2 INTRODUCTION

With this unit learners will be acquainted with the key elements of programming to design accurate and efficient programs. Algorithms, pseudocodes, and flowcharts are used in the process of program development to help the programmers as well as the users to clearly understand the solution to the problem at hand.

In this unit we will learn the method of writing algorithms and pseudocodes, and pictorially represent a schematic flow of logic in the form of flowcharts.

1.3 ALGORITHM

Algorithm is a sequence of instructions to solve a problem. An algorithm gives the logic of the program, that is, a step-by-step description of how to arrive at a solution. In general terms, an algorithm provides a blueprint for writing a program to solve a particular problem. Once we have an blueprint of a solution, we can implement it in any high-level language, such as C, C++, Java etc.

An algorithm has a finite number of steps and some steps may involve decision-making and repetition.

1.3.1 Conventions used in Writing Algorithms

The following are the conventions used in writing algorithms:

- **Name of the algorithm**: Every algorithm is assigned a name which reflects the task to be performed by it.
- **Introductory Comments**: The task performed by the algorithm is described briefly in this section. The variables used along with their data types are mentioned here.
- **Steps**: An algorithm comprises of a sequence of steps which should be numbered. The statements within a step are executed in a left to right manner.

- **Comments in Steps**: Each step is preceded by a brief comment describing its function. Comments within a step are enclosed in parentheses.

The algorithm shown in ***Example 1.6:****(Finding the largest of three numbers)* is written considering all the above conventions. For a better understanding, let us write some algorithms for solving simple problems.

1.3.2 Method for Developing an Algorithm

- State the problem you are trying to solve in clear and concise terms.
- List the *inputs* (information needed to solve the problem) and the *outputs* (what the algorithm will produce as a result)
- Identify the steps needed to convert or manipulate the inputs to produce the outputs.
- Test the algorithm: choose different data sets and verify that your algorithm works.

***Example 1.1*: Write an algorithm to add two numbers.**

Solution:

```
Step 1: Input the first number as A
Step 2: Input the second number as B
Step 3: Set  Sum = A+B
Step 4: Display Sum
Step 5: End
```

The algorithm for adding two numbers can also be written as:

```
Step 1: Read a
Step 2: Read b
Step 3: Sum→ a + b
Step 4: Write Sum
Step 5: Stop
```

Both the two algorithms for adding two numbers are correct, but only the words/texts are different. While writing algorithms, it is good to keep the following points in mind:

- Usually, words like *Read*, *Input* or *Accept* can be used to represent input operation to give values of variables to the computer.
- *Display*, *Show*, *Write* or *Print* can be used to represent output operation to show the result computed by the computer.
- Back arrow '$\rightarrow$' represents the value obtained by evaluating the right side variables or expression and assigning it to the left side variable. The symbol '='can also be used instead of '$\rightarrow$'.
- In case of branching or conditional statements, ***If-Then*** or ***If-Then-Else*** is used. The conditional statement usually contains relational operators such as <, >, <=, >= etc.
- The iterative or repetitive statements can be written between ***Repeat For*** or ***Repeat While*** *loops*.

***Example 1.2*: Write an algorithm to find whether a number is even or odd.**

Solution :

```
Step 1: Input the first number as A
Step 2: if A % 2 = 0
             then print "Even"
        else
              print "Odd"
Step 3: End
```

***Example 1.3:* Write an algorithm to find the larger of two numbers.**

Solution:

```
Step 1:  Input first number as A
Step 2:  Input second number as B
```

```
Step 3:  if  A > B
             then display A
           else if A < B
                then display B
           else
              print "The numbers are equal"
Step 4: End
```

Example 1.4: **Write the algorithm to convert the temperature on °F (Fahrenheit) to °C(Centigrade) using the formula °C = 5/ 9 (°F – 32) .**

Solution: The input variable is F (temperature in °Fahrenheit) and the output variable is C (temperature in °*Centigrade*)

```
Step 1: Read F
Step 2: C→5/9*(F - 32)
Step 3: display C
Step 4: Stop
```

Example 1.5**: Write an algorithm to find the sum of first N natural numbers.**

Solution:

```
Step 1: Input N
Step 2: Set  i = 0, sum = 0
Step 3: Repeat Step 3 and 4 while i <=N
Step 4: Set sum = sum + i
           set  i = i + 1
Step 5: display sum
Step 6: End
```

Example 1.6: **Write an algorithm to compute the largest of three numbers.**

Solution:

Algorithm: Largest

This algorithm computes the largest of three numbers.

The variable names are :

a, b, c : type integer

large : type integer, storing the value of the largest number

```
Step 1:  [Input three integers]
          Read a, b, c
Step 2: [Compute the largest of three numbers]
           large =a;
              If (b > large)   large = b
       if (c > large)   large = c
Step 3:  [Display the largest number]
              Print (large)
Step 4: [Finished]
              Exit
```

The steps in an algorithm are normally of three different types:- ***sequence***, ***selection***, and ***iteration*** type statement.

- ***Sequence*** means that each step of the algorithm is executed in the specified order. The algorithm in ***Example 1.1*** performs the steps in a sequential order.
- ***Selection*** or ***Decision*** statements are used when the outcome of the process depends on some condition. The general form is

```
if condition
            then  statement1
else statement2
```

For example,

```
if x = y
     then print "Equal"
else
    print "Not Equal"
```

The algorithm shown in ***Example 1.2*** contains decision statements.

- ***Iteration*** or ***Repetition*** involves executing one or more steps for a number of times. This can be implemented using constructs ***Repeat-For***, ***Repeat-While***. Repetition occurs in one or more steps untill some condition is true. The algorithm shown in ***Example 1.5*** contains repetitive

Example 1.7: Write an algorithm to find the area of a triangle whose sides are a,b and c.

Solution: The formula for finding the area of a triangle is

$\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{a + b + c}{2}$

Step 1: Input a,b,c

Step 2: $s = \frac{a + b + c}{2}$

Step 3: Area = $\sqrt{s(s-a)(s-b)(s-c)}$

Step 4: Display Area

Step 5: End statements.

Example 1.8: Write an algorithm to count the number of zon-zero data.

Solution:

```
Step 1:  Initialize the non-zero data counter c to 0
Step 2: Repeat 100 times all steps up to Step 4
Step 3: Read d
Step 4: If d ≠ 0 then
                    c ← c+1
Step 5: Display c
Step 6: Stop
```



CHECK YOUR PROGRESS

Q1: State True/ False:

i) An algorithm solves a problem in a finite number of steps.

ii) The conditional statement usually contains relational operators.

iii) Repetition occurs in one or more steps untill some condition is false.

Q2: Write an algorithm for interchanging or swapping two values.

1.4 PSEUDOCODES

Pseudocode are statements written in structured english for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of programming language syntax. At the same time, the pseudocode needs to be complete.

There are no clearly defined standards for writing a pseudocode. Indentation is used to increase clarity while writing pseuodocodes. It helps even non-programmers to understand the logic of the problem. The aim of writing pseudocode is to get the idea quickly and to be able to read easily without details. It is like a young child putting sentences together without any grammar. For simplicity, let us represent few works done in our daily life in the form of pseudocode:

Brush teeth
Wash face
Comb hair
See in mirror

Some more examples of pseudocodes in terms of computer programming are given below:

Example 1.9: Write a pseudocode to find area of a rectangle.

Solution:

```
READ length of rectangle
READ breadth of rectangle
COMPUTE area as length * breadth
```

Example 1.10: Write a pseudocode for finding the average of 5 numbers.

Solution:

```
Input 5 numbers
sum = add numbers together
avg = sum / 5
Display avg
```

Example 1.11: Write a pseudocode to read the marks of 10 students. If marks are greater than 150, the student passes, or else the student fails. Count the number of students who haved passed and failed.

Solution: The variables used in this example are : totalpass, totalfail, no.of students, marks

```
1. Set totalpass to 0
2. Set totalfail to 0
3. Set no.of students to 0
4. While no. of students < 10
      a.  Input the marks
      b.  If  marks >=150
                Set totalpass = totalpass +1
          Else
                Set totalfail = totalfail + 1
          EndIf
    EndWhile
5. Display totalpass, totalfail
6.  End
```

Some **advantages** of pseudocodes are given below:

- The language independent nature of pseudocode helps the programmer to express the design in plain natural language.
- It can be designed based on the logic of the problem without being concerned about programming syntax or rule.

The main **disadvantages** of pseudocode are:

- It does not have any standard format or syntax of writing.
- It cannot be compiled or executed.



CHECK YOUR PROGRESS

Q3: Is there any standard rule when writing pseudocode?

Q4: What is pseudocode?

1.5 FLOWCHART

We are already acquainted with the meaning of "*pseudocode*" and "*algorithm*" in the previous section. Before we start coding a program, it is necessary to plan the step-by-step solution to the problem. Such a systematic plan can be symbolically represented with the help of a diagram. This diagram is called a *flowchart*. A flowchart is a symbolic representation of a solution to a given task. In this section we will learn to draw flowcharts using various symbols associated with it.

A ***flowchart*** is a pictorial representation of an algorithm. It shows the logic of the algorithm and the flow of control. The flowchart uses symbols to represent specific actions and arrows to indicate the flow of control.

Normally, an algorithm is expressed as a flowchart and then the flowchart is converted into a program using some programming language. Flowcharts are independent of the programming language that are being used. Hence, one can fully concentrate on the logic of the problem solving at this stage.

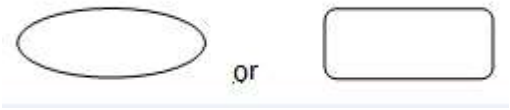
It is always recommended for a beginner, to draw flowcharts prior to writing programs in the selected programming language.

1.5.1 Symbols of Flowchart

Flowcharting has many standard symbols. The boxes which are used in flowcharts are standardized to have specific meanings. These flowchart symbols have been standardized by the *American National Standards Institute* (ANSI). The symbols of a flowchart include:

- **Start and End** (or, Start and Stop): Every flowchart has a unique starting point and an ending point. The *Start* and *End* symbols are also known as *terminal symbols* and are represented as **o*vals***, or ***rounded rectangles***. Flowchart begins at the start terminator and ends at the stop terminator. The starting point is indicated with the word START inside the terminator symbol.

The ending point is indicated with the word STOP inside the terminator symbol.



- **Input/Output** : Input/Output symbols are used to denote any input/output function in the program. These are represented using a ***parallelogram*** and are used to get inputs from the users or to display the results to them.



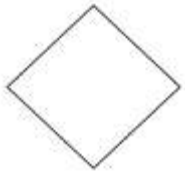
Thus, if there is any input to the program via an input device, like a keyboard, tape etc. it will be indicated in the flowchart with the help of the Input/Output symbol. Similarly, all output instructions, for output to devices like printers, monitors etc. are indicated in the Input/ Output symbol.

- **Process**: A process or computation represents arithmetic and data movement instructions in the flowchart. It is generally represented by using a ***rectangle***. All arithmetic processes of addition, subtraction, multiplication and division are indicated in the process symbol. If there are more than one process instructions to be executed sequentially, they can be placed in the same process box (rectangle), one below the other in the sequence in which they are to be executed.

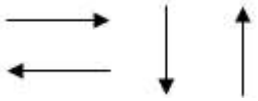


- **Decision :** The decision symbol is represented by using a ***diamond***. It is used in a flowchart to indicate the point where a decision is to be made and branching done upon the result of the decision to one or more alternative paths. The criteria for decision making is written inside the decision box.

It is basically used to depict a Yes/No question or a True/False test. The two arrows coming out of it, one from the bottom point and the other from the right point, corresponds to Yes or True, and No or False., respectively. The arrow should always be labeled.



- **Flow lines/Arrows:** Flow lines (or, Arrows) are solid lines with arrowheads which indicate the flow of operation. They show the exact sequence in which the instructions are to be executed. The normal flow of the flowchart is depicted from top to bottom and from left to right.



- **Connectors:** In situations, where the flowcharts become big, it may so happen that the flow lines start crossing each other at many places causing confusion. This will also result in making the flowchart difficult to understand. Also, the flowchart may not fit in a single page for big programs. Thus, whenever the flowchart becomes complex and spreads over a number of pages connectors are used.

  The connector represents entry from or exit to another part of the flowchart. A connector symbol is indicated by a ***circle*** and a letter or a digit is placed inside the circle. This letter or digit indicates a link. A pair of such identically labelled connectors is used to indicate a continued flow in situations where flowcharts are complex or spread over more than one page.

  Connectors do not represent any operation in the flowchart. Their use is only for the purpose of increased convenience and clarity.



The flowchart symbols discussed above are given in together in Figure 1.1:

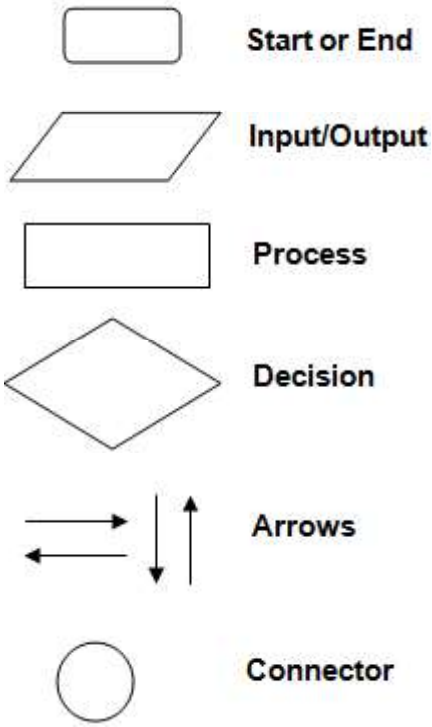


Figure 1.1: Flowchart Symbols

1.5.2 Advantages and Limitations of Flowchart

Advantagesof Flowchart : There are a number of advantages of flowcharts in problem solving.

- The flowchart being a pictorial representation of a program makes it easier for the programmer to explain the logic of the program to others rather than using a program.
- It shows in a simple way the execution of logical steps without the syntax and language complexities of the program.
- In real life programming situations a number of programmers are associated with the development of a system and each programmer is assigned a specific task of the entire system. Hence, each programmer can develop his own flowchart and, later on, all the flowcharts can be combined for depicting the

overall system. Any problems related to linking of different modules can be detected at this stage itself and suitable modifications can be carried out. Flowcharts can thus be used as working models in design of new software systems.

- Flowchart has become a necessity for better documentation of complex programs.
- Flowchart also enables us to trace and detect any logical or other errors before the programs are written. Hence, a flowchart is very helpful in the process of debugging a program.
- Flowcharts are very helpful during the testing of the program as well as incorporating further modifications

Limitations of Flowchart :

- Drawing flowchart for large complex problem is a laborious and time-consuming activity. Many a time, the flowchart of a complex problem becomes complex and clumsy.
- Sometimes, a little bit of alteration in the solution may require a complete re-drawing of the flowchart.



CHECK YOUR PROGRESS

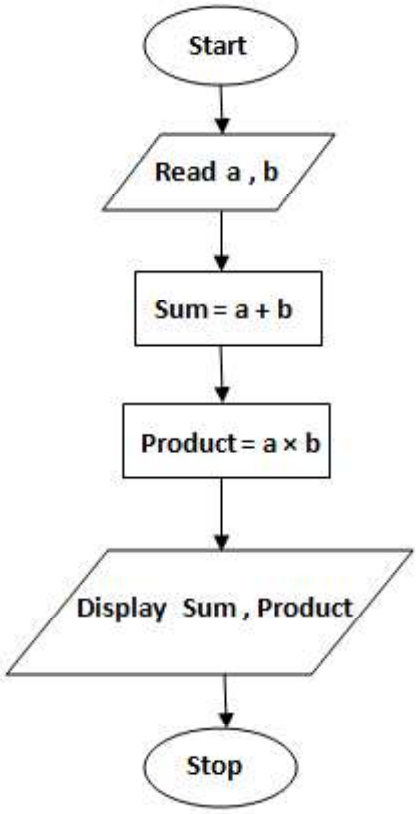
Q5: What is a flowchart?

Q6: Give any two advantages of flowcharts.

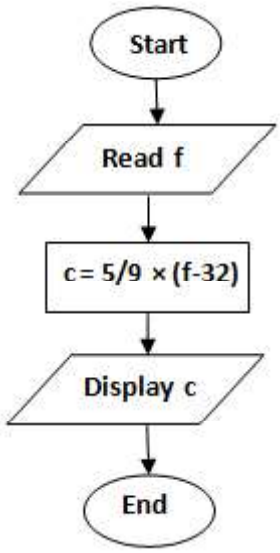
Let us look at few examples of flowcharts:

***Example 1.12*:** Draw the flowchart to find the sum and product of two given numbers.

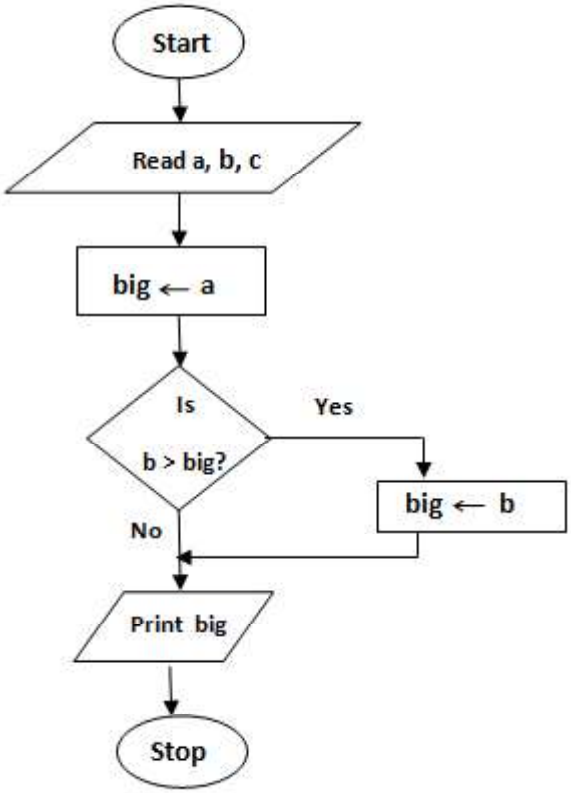
Solution:



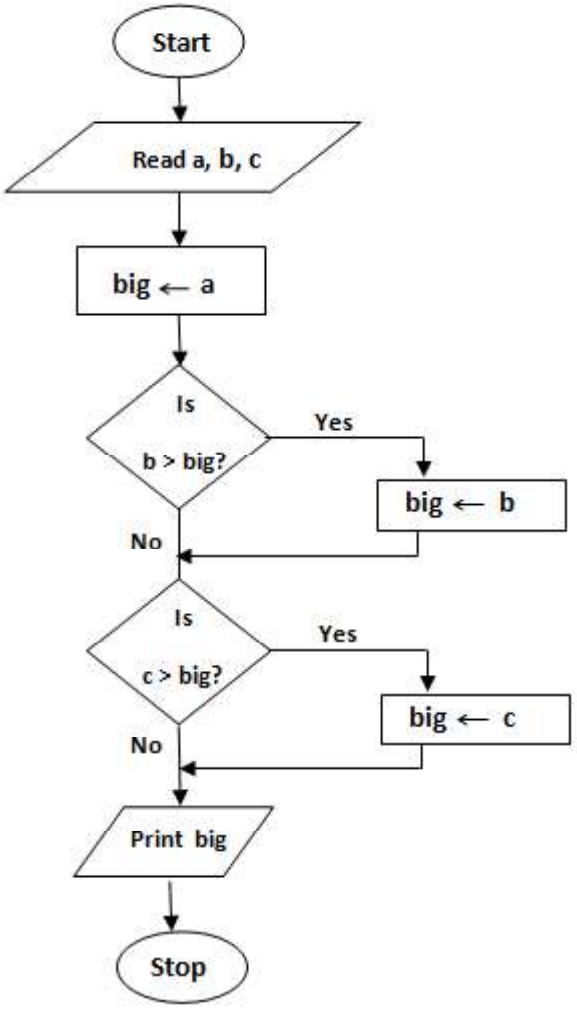
Example 1.13: Draw the flowchart to convert the temperature in Fahrenheit (^{0}f) to Centigrade (^{0}c).



Example 1.14: Draw the flowchart to find the biggest of the two given numbers.



Example 1.15: Draw a flowchart to find the biggest of three given numbers.





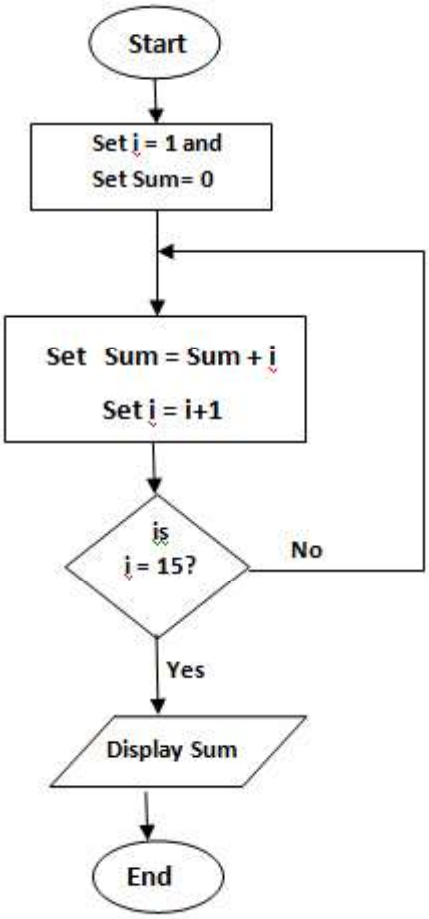
ACTIVITY

Q1: Draw a flowchart to calculate the salary of a daily wager using the formula given below:

[Salary=(no_of_hours*pay_per_hour)+Travel_allowance]

Q2: Draw a flowchart to subtract two numbers.

Example 1.16: Draw a flowchart to calculate the sum of first 15 natural numbers.





CHECK YOUR PROGRESS

Q7: In a flowchart, which symbol is represented using a rectangle?

a) Decision b) Connector

c) Input/Output d) Process

Q8: Which one of the following is a graphical or symbolic representation of a process?

a) Algorithm b) Pseudocode

c) Flowchart d) Program

Q9: The _____ symbol is always the first and the last symbol in a flowchart.



1.6 LET US SUM UP

- An *algorithm* is a precise specification of a sequence of instructions to be carried out in order to solve a given problem.
- In the context of computer programming, an *algorithm*, is defined as a well-ordered collection of unambiguous and effectively computable operations which, when executed, produces a result and halts in a finite amount of time.
- *Pseudocode* consists of English-like phrases describing an algorithm. An ideal pseudocode must be complete, describing entire logic of the algorithm, so that it can be translated straightway into a program using any programming language. It facilitates programmers to focus on the logic of the algorithm.
- A flowchart is a pictorial representation of an algorithm. It shows the logic of the algorithm and the flow of control.
- The flowchart uses symbols to represent specific actions and arrows to indicate the flow of control.
- For ***Start*** and ***End/Stop***, ***oval*** or ***rounded rectangle*** is used.
- For ***input/output*** opereation, ***parallelogram*** is used.
- For ***process*** or ***computation***, ***rectangle*** is used.
- For ***decision***, the ***diamond*** symbol is used.
- For indicating ***flow of control***, ***arrows*** with different heads are used.
- For indicating ***connections***, labelled ***circle*** are used.



1.7 FURTHER READING

1. Balagurusamy, E. (2002). programming in ANSI C. Tata McGraw-Hill Education.
2. Gottfried Byron, S. Programming with C. Tata McGraw-Hill



1.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q. No. 1 : i) True ii) True iii) False

Ans to Q. No. 2 : Algorithm for interchanging/swapping two values:

Step 1: Input first number as A

Step 2: Input second number as B

Step 3: Set temp = A

Step 4: Set A = B

Step 5: Set B = temp

Step 6: Display A, B

Step 7: End

Ans to Q. No. 3: There are no such set rules, but the code should provide clear descriptions of the algorithms being outlined.

Ans to Q. No. 4: Pseudocode is not an actual programming language. It uses short phrases to describe an algorithm before you actually create it in a specific programming language. Once you know what the program is about and how it will function, then you can use pseudocode to create statements to achieve the required results for your program.

Ans to Q. No. 5: A flowchart is a diagrammatic or symbolic representation of an algorithm. It uses various symbols to represent the operations to be performed.

Ans to Q. No. 6: The advantages of flowcharts are:

i) Flowcharts are helpful for better documentation of complex programs.

ii) Flowchart also enables us in tracing and detecting any logical or other errors before the programs are written.

Ans to Q. No. 7 : d) Process

Ans to Q. No. 8 : c) Flowchart

Ans to Q. No. 9 : Start and End



1.9 MODEL QUESTIONS

Q1. Define algorithm. How is it useful in the context of software development?

Q2. What do you understand by the term pseudocode?

Q3. Differentiate between algorithm and pseudocode.

Q4. Write an algorithm to find the smallest of three given numbers.

Q5. Write the algorithm to find the average of three given numbers.

Q6. Write an algorithm to find the sum of natural numbers upto N.

Q7. Write an algorithm to find the area of a triangle.

Q8. Write an algorithm to find the sum of a set of numbers.

Q9. Write an algorithm to find the factorial of a given number.

Q10. Write a pseudocode to compute the average of *n* numbers.

Q11: What is a flowchart? What are the significance of flowchart?

Q12: With the help of an example explain the use of a flowchart.

Q13: How is a flowchart different from an algorithm? Do we need to have both of them for developing a program?

Q14: Draw the flowchart to find the area and circumference of a circle of radius r.

Q15: Draw the flowchart to find the smallest of three given numbers.

Q16: Discuss the advantages and limitations of flowchart.

Q17: Describe the symbols used in flowchart.

Q18. Draw the flowchart to find the sum of digits of a number.

*** ***** ***

UNIT 2: ELEMENTS OF C PROGRAMMING

UNIT STRUCTURE

2.1 Learning Objectives
2.2 Introduction
2.3 C Character Set
2.4 Tokens
2.5 Identifiers
2.5.1 Rules for forming Identifier Names
2.6 Reserved Words
2.7 Constants
2.8 Variables
2.9 Let Us Sum Up
2.10 Further Reading
2.11 Answers to Check Your Progress
2.12 Model Questions

2.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- describe the C character set
- describe C tokens
- describe the primary elements like identifiers, reserved words, constants, types of constants etc.
- define C variables

2.2 INTRODUCTION

We have already learned about basic programming concepts including pseudocode, algorithm and flowcharts in the introductory unit. In this unit, we will introduce basic concepts in C like identifiers, keywords and constants. The concept of variables are also included as an important part of the unit.

2.3 C CHARACTER SET

C language 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 :	uppercase letters A to Z lowercase letters a to z
Digits :	0 to 9
Special Characters :	# & | ! ? _ ~ ^ { } [ ] ( ) < > space . , : ; ‘ $ “ + - / * = %

2.4 TOKENS

C tokens are the basic buildings blocks in C language. The smallest individual units in a C program are known as C tokens. There are six types of C tokens. They are :

1. Keywords (eg: int, do),
2. Identifiers (eg: main, total),
3. Constants (eg: 5, 25),
4. Strings (eg: “kkhsou”, “university”),
5. Special symbols (eg: (), {}),
6. Operators (eg: +, /,-,*)

We will discuss some of these basic building blocks in this unit.

2.5 IDENTIFIERS

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

2.5.1 Rules for forming Identifier Names

To construct an identifier we have to follow some rules which are as follows:

- Only alphabet, digit and underscores are permitted. i.e., it cannot include any special characters or punctuation marks (e.g., &, %, #, $,@, ..,etc.) except the underscore.
- There cannot be two successive underscores.
- An identifier name cannot start with a digit. It must begin with an alphabet or underscore. However,use of underscore as the first character must be avoided because in C library, several compiler-defined identifiers have underscore as their first character.Hence, duplicated names may cause definition conflicts.
- Identifiers are case sensitive, i.e. uppercase and lower case letters are distinct.For example,'SUM' is different from 'sum' and 'Sum'.
- It is a good practice to choose a meaningful identifier name but it should be short. Maximum length of an identifier is 32 characters.

Let us give some examples of valid identifier names :

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)

2.6 RESERVED WORD

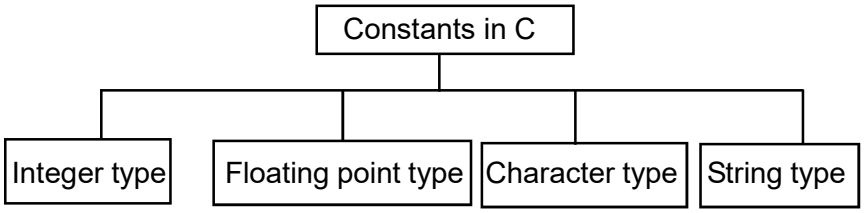
Reserved words are the essential part of a language definition. The meaning of these words has already been explained to the C compiler. Since these reserved words have some special meaning in C, therefore these words are often known as "keyword". C language uses the following keywords which are not available to users to use them as variables/function names:

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

2.7 CONSTANTS

Constants are identifiers whose value does not change. Constants are used to define fixed values so that their fixed value does not get changed in the program even by mistake.Thus, the value of a constant remains constant through the complete program.

Depending on the type of data, C language allows programmers to specify the constants of interger type, floating point type, character type, and string type.



- **Integer Constant**: An integer constant consists of a digit or a sequence of digits. For example, 5, 53, 120, 3451. By default, a literal integer such as 1432 is of type *int* . For a long integer constant the literal is succeeded with either ‘L’ or ‘l’. Similarly, an unsigned int literal is written with a ‘U’ or ‘u’ suffix. For example, 1432l, 1432L, 1234u, 1234U, 1234ul are all valid integer constants.

Integer constant can again be subdivided into *decimal* (base-10), *octal* (base-8), and *hexadecimal* (base-16) integer constant. Octal integer constant preceded by zero and consist of a set of digits, 0 through 7.

For example, octal integers include 01, 012, 01234, 0 etc.
In the same way hexadecimal integer constants are preceded by 0x or OX. Hexadecimal numbers are from 0 to 9 and alphabets A through F. The alphabets A through F represent numbers 10 through 15. For example, decimal 65 is equivalent to 0101in octal notation and 0x4• 1 in hexadecimal notation.

- **Character Constant**: A character constants consists of single character enclosed with *single quotation mark*. For example, 'a', '1', '@', etc.

- **String Constant**: One important variation of character constant is *string constant*. A string constant is a sequence is a sequence of characters. A string constant is always enclosed with a *double quotation mark*. For example, "a", "kkhsou", "153" are valid string constants.

- **Floating Point Constants**: A floating point constants consists of an integer part, s decimal point, a fractional part, and an exponent field containing an e or E followed by an integer where the fraction part and integr parts are a sequence of digits. (e or E indicated expoent part). Some valid examples of floating point numbers are: 0.01, 125.34, -0.5,+0.15, +0.55, 0.4 etc.

Literals like 0.05, 0.07 are treated as double by default. We have to use the suffix 'F' or 'f' to make it a float type literal, and suffix L for long double. Some examples of floating point literals are 0.01F, 0.35f, 3.141592654L etc. Again, 0.2e3,1.5e+3 are two examples of floating point numbers in scienctific notation.

2.8 VARIABLES

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. Suppose

you want to store value 153 to a variable. In C programming language a varibale can be assigned a value using the following statements :

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

Here, first we declare the data type of the variable followed by a variable name, in this case “A”. After that, A is assigned the value 153. Similarly, we can store strings in char variables and decimal numbers in floating point variables. We will discuss initialization and declaration of variables in detail in the next unit.



CHECK YOUR PROGRESS

Q 1. What is an identifiers ?
Q 2. What is a keyword in C ?
Q 3. What are the types of constants ?
Q 4. What is a variable ?



2.9 LET US SUM UP

- C character set includes uppercase and lowercase alphabets, digits and several special characters. Altogether there are 93 valid characters allowed in C.
- Identifiers are the name given to the various program elements - variables, functions, arrays etc.
- 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.
- An octal integer constant always starts with 0 and a hexadecimal constant with 0x.



2.10 FURTHER READING

1. Balagurusamy, E. (2002). programming in ANSI C. Tata McGraw-Hill Education.
2. Gottfried Byron, S. Programming with C. Tata McGraw-Hill publication.
3. Thareja Reema. Computer Fundamentals and Programming in C. Oxford University Press



2.11 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: 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.

Answer to Q2: 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.

Answer to Q3: There are two broad categories of constant in C, ***literal constant*** and ***symbolic constant.***

Answer to Q4: 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 a program.



2.12 MODEL QUESTIONS

Q 1. What is the difference between a keyword and an identifier ?

Q 2. List the rules of naming an identifier in C ?

Q 3. Differentiate between a variable and a constant.

Q 4. Define variable and tokens in C language.

Q 5. Define constants in C. What are the different types of constants in C? Give suitable examples.

*****

UNIT 3 : VARIABLES AND DATA TYPES

UNIT STRUCTURE

3.1 Learning Objectives
3.2 Introduction
3.3 Basic Data Types in C
3.4 Variables in C
3.4.1 Declaration of Variables
3.4.2 Initialization of Variables
3.5 Symbolic Constants
3.6 Let Us Sum Up
3.7 Further Reading
3.8 Answers to Check Your Progress
3.9 Model Questions

3.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- learn the basic data types used in C language
- declare and initialize C variables
- learn about symbolic constants

3.2 INTRODUCTION

We have already learned about basic programming concepts including pseudocode, algorithm and flowcharts in the first unit. Concepts of C character set, identifiers and variables have also been intoduced in the previous unit.

In this unit, you will discuss the elementary data types used in C language. The concept of variables and constants are also included as an important part of the unit.

3.3 BASIC DATA TYPES IN C

Let us look 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 3.1 shows the different categories of C data type.

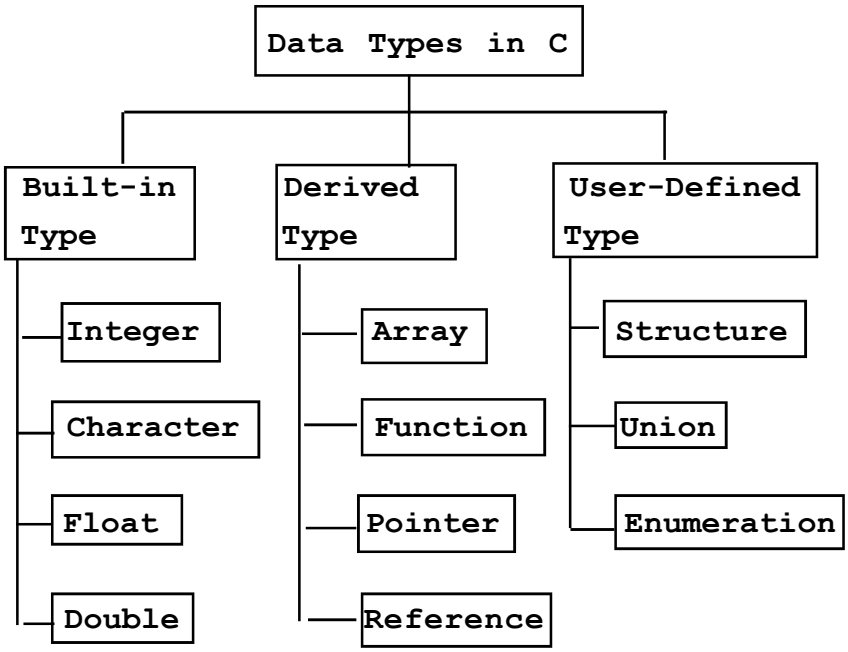


Fig 3.1: C Data Types

C language provides very few *basic data* types which are also known as ***built-in*** data type or ***elementary*** 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 3.1 lists all combinations of the basic data types and modifiers along with their size and ranges on a 16-bit computer .

Moreover, besides these basic data types another special data type is ***void***. The void type holds no value. 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. The other two categories of data types will be discussed in subsequent units of this course.

Table 3.1: Size and range of basic data type and its modifier

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

At this we would like to mention that C language does not provide any data type for storing text. This is because text is made up of individual characters. In memory *characters* are stored using ASCII codes. For example, for the character 'A', the ASCII equivalent is 65. In memory the binary equivalent of 65 is stored for the letter 'A'.

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.3.2.

Table 3.2: Format specifier for usual variable type

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 (string)	sequence of characters

3.4 VARIABLES IN C

A variable is defined as name given to the storage location in computer memory. When using a variable, we actually refer to an address of the memory where the data is stored.

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.

3.4.1 Declaration of Variables

Each variable to be used in the program must be declared. For declaration of variable, we first specify the data type of the variable followed by its name. The data type indicates the kind of data that the variable will store. A variable cannot be of type *void*. In C, variable declaration always ends with a semi-colon. The general syntax of declaration of a variable is :

data_type variablename;

We can also declared more than one variable in a single statement as follows:

data_type variable1, variable2, ... , variableN;

For example,

```
int roll;
float salary;
char grade;
int m1,m2,m3;
int total_marks;
```

In C, variables can be declared at any place in the program but one thing should be kept in mind. Variables should be declared before using them. By declaring a variable we usually tell three things to the compiler :

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

3.4.2 Initialization of Variables

While declaring the variables,we can also initialize them with some value. Following are few examples of initialization statements:

```
int roll_no= 5;
float average=200.75;
char grade= 'A';
```

The initializer applies only to the variable defined immediately before it. Therefore, the statement

```
int number, sum=0;
```

intializes the variable *sum* and not *number*.

Let us take a simple example for explaining declaration and initialization of variables. Suppose we want to store value 153 to a variable. We first create the name of the variable, suppose A. Since 153 is integer, so we declare the variable A as *integer* type, and then assign 153 to that variable. These can be done as follows :

```
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 cannot store value into A other than integer. Therefore, this type of statement is known as *declaration* statement. 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.* We can store values to a variable in two ways :

- using assignment statement, and
- using a read statement.

The use of assignment statement is already shown. In the second approach you can make a call of C standard input function like *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.

A few examples of variable declaration and initialization 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.

3.5 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 π (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

Q 1. What are the basic data types used in C?

Q 2. Define variable.

Q 3. What are symbolic constants ?

Q 4. State true or false:

i) Variable name 'Salary' is same as 'salary'.

ii) Semicolon is necessary in variable declaration statement.

iii) char grade= F; is a valid initialization statement.

iv) Variables should be declared before using them.



3.6 LET US SUM UP

- 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.



3.7 FURTHER READING

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



3.8 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: The built in data types are *integer, character, float, double*.

Answer to Q2: 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.

Answer ot Q3: 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.

Answer to Q4: i) False ii) True iii) False iv)True



3.9 MODEL QUESTIONS

Q 1. Name and describe the four basic data types in C?

Q 2. What is a variable?

Q 3. How are variable initialize? Explain with example.

Q 4. What are symbolic constants? How are they different from variables? Explain in brief.

Q 5. What does the data type of a variable signify?

Q 6. Write a short note on basic data types that the C language supports?

Q 7. Explain the term ‘variable’ and ‘constants’?

Q 8. Explain the utility of #define and #include statements.

*****

UNIT 4: OPERATORS AND EXPRESSIONS

UNIT STRUCTURE

4.1 Learning Objectives

4.2 Introduction

4.3 Operators

- 4.3.1 Arithmetic Operators
- 4.3.2 Relational Operators
- 4.3.3 Logical Operators
- 4.3.4 Assignment Operators
- 4.3.5 Increments and Decrement Operators
- 4.3.6 Conditional Operators
- 4.3.7 Bitwise Operators
- 4.3.8 Special Operators

4.4 Precedence and Associativity

4.5 Expressions

4.6 Type Conversion

4.7 Let Us Sum Up

4.8 Further Reading

4.9 Answers to Check Your Progress

4.10 Model Questions

4.1 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
- use expession in programming
- perform type conversion.

4.2 INTRODUCTION

We have already learnt how variables and different data types can be used in programming. These variables, constants and other elements can be joined together by various operators to form expressions.

In this unit we will learn about different types of operators and how these operators are used to form expressions.

4.3 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. They are used with ***operands*** to build expressions. We will discuss expression at the end of this unit.

For example, the following is an expression containing two *operands* (i.e., 2 and 4) and one operator (i.e., +)

2 + 4

C language has a rich set of operators which can be classified as follows:

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

4.3.1 Arithmetic Operators

For arithmetic operations such as plus, minus, multiplication, division etc., we need some operators. C provides all the basic arithmetic operators. They are listed in Table 4.1. The operators +, -, * and / all work in the same way as they do in other programming

languages. These operators can operate on any built-in data types allowed in C.

Table 4.1: Arithmetic Operators

Operator	Meaning
+	Addition or Unary Plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulo Division

Some uses of arithmetic operators can be seen in the following expressions:

x + y

x - y

- x + y

a * b + c

- a * b

Here ***a***, ***b***, ***c***, ***x***, ***y*** are ***operands***.

- **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 (Modulo Division)

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

Example 4.1: Write a program to show the 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 n1 = 5 and n2 = 2, then the output of the above program will be as follows:

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 requirement. The modulus operator is not applicable for fractional numbers.

Let x = 15.0 and y = 2.0 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 ***mixed mode arithmetic***. If any one operand is of real type then the result will always be real, thus 15 / 10.0 = 1.5.

4.3.2 Relational Operators

Often it is required to compare the relationship between operands and bring out a decision accordingly. For example, we may compare the salary of persons, age of two persons, marks of students, or the price of two items, and so on. These comparisons can be done with the help of relational operators. C supports the following relational operators:

Table 4.2 Relational operators

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

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. The value of a relational expression is either *zero* or *one*. The value is *zero* if the relation is *false* and *one* if the relation is *true*.

Some examples of relational expressions and their evaluated values are listed below:

3.5 <= 12	TRUE
-6 > 0	FALSE
10 < 7 + 5	TRUE
4 == 2	FALSE
4! =2	TRUE

Let us consider the following example where there are two arithmetic expressions on either side of the relational operator >.

```
p+q > m+n
```

In such cases, the arithmetic expressions will be evaluated first and then the result is compared. The result will be TRUE only if the sum of the values of *p+q* is greater than the sum of the values of *m+n*. This is because, arithmetic operators have higher precedence over relational operators. We will discuss operator precedence later in this unit.

Relational expressions are used in decision making statements such as ***if***, ***while*** and ***for*** of C language. We shall learn about these statements in later units.

4.3.3 Logical Operators

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

Table 4.3: Logical Operators

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

The AND (&&) and OR (||) allow two or more conditions to be combined in an if statement 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**. The following example is given to show the usage of logical AND with an ***if*** statement:

if (age>18 && marks>=300)

- **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 also when a is less than both **m** and **n**.

- **Logical NOT (!):** The NOT operator is denoted by !. It 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 it operates on. For example,

! (x < 5)

This means if x is less than 5, the result will be FALSE, since (x<5) is TRUE. The NOT operator is often used to reverse the logical value of a single variable.

4.3.4 Assignment Operators

The assignment operator denoted by = is used to assign the result of an expression to a variable. For example:

x = a + b ;

In the above statement, the value of *a* + *b* is evaluated and substituted to the variable *x*.

Again, in the statement x = x + 1; the value of x is incremented

by 1 and it is assigned to x in the left hand side. This can also be written as x + = 1; This notation of assigning is known as *shorthand* form.

The commonly used shorthand assignment 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 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. These operators cannot be used in place of one another.

Program 4.2: Program to 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



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 expression
```

4.3.5 Increment and Decrement Operators

The increment and decrement operators are one of the unary operators which are very useful in C language. These are:

+ + and - -

The increment operator ++ adds 1 to the operand, while the decrement operator - - subtracts 1. If i is an integer type variable, then we can write these operators in program as follows:

```
    + + i;   pre-increment
or  i + +;   post-increment
    - - i;   pre-decrement
or  i -  -;  post increment
```

Again, the statement + + i; is same as i = i+1; or i + =1; and - - i; is same as i = i -1; or i - =1;

The ++i and i++ means the same thing when they form statements independently. But they behave differently when they are used in expressions on the right-hand side of an assignment statement.

When the operator ++ is written before the variable name (e.g., ++i) then the operator is called **prefix operator** and when the operator ++ is written after the variable name (e.g., i++) then it is called **postfix operator**. 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.

Suppose, 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.

Rules for increment (+ +) and decrement (- -) Operators:

- 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 -.

4.3.6 Conditional Operators

The ternary operator "**?:**" is used to denote conditional operator. Three operands are used here so it is called ternary operator. The syntax is as follows:

exp1 ? exp2 : exp3

where **exp1**, **exp2**, and **exp3** are expressions. **exp1** is evaluated first. If the expression is TRUE then **exp2** is evaluated and its value becomes the value of the expression. If **exp1** is FALSE, **exp3** is evaluated and its value becomes the value of the expression. Only one of the expressions is evaluated. For example:

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

Here, value of **b** will be assigned to **x**. Since, a=10 and b=15, the condition a>b will be FALSE; therefore **b** is assigned to **x**.

Program 4.3: 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 cannot  vote");
      getch( );
}
```

Output :

```
Enter your age in years:
 26
 You can 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
```

Program 4.4: 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 larger of two numbers is %d \n",
large);
}
```

Output : Enter 2 integers: 14 25

The larger of two numbers is: 25

4.3.7 Bitwise Operators

Bitwise operators are used for manipulation of data at bit level. A bitwise operator operates on each bit of data. Those operators are used for testing, complementing or shifting bits to the right or left hand side. Bitwise operators may not be applied to a float or double.

Table 4.4: Bitwise operators

Operator	Meaning
&	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:

Table 4.5: Truth Table for the bitwise AND (&) operator

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

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

Table 4.6: Truth Table for the bitwise OR (|) operator

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

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:

Table 4.7: Truth Table for the exclusive OR(^)

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

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:

Table 4.8: Truth table for the Bitwise Complement (~)

bit x of op2	result
0	0
0	1

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.

5 << 3

Left operand Right 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;

Here, Z in binary can be represented as 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 in decimal

and Z << 15 is 10000000 00000000 or 32768 in 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, it means 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.

4.3.8 Special Operators

C language supports some special operators such as comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. and ->). The sizeof and the comma operators are discussed in this 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. It is used most often in the **for** loop statement.

The comma operator forces all operations that appear to the left to be fully completed before proceeding to the right of comma. This help to eliminate the 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 in the later units.

Program 4.5: Program to 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 4 and 8)

Enter the two integer numbers: 4 8

Entered numbers are: 4 8

Swapped numbers are: 8 4

- **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 for determining the lengths of arrays and structures when their sizes are not known to the programmer. It is also used for allocating memory space dynamically to variables during the execution of the program.

Program 4.6: 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 expression.



CHECK YOUR PROGRESS

Q.1: 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);  }
```

Q.2: 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);
    }
```

Q.3: 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



EXERCISE 4.1

1) Write a program that reads a floating-point number and then displays the right-most digit of the integral part of the number.

4.4 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, each operator has 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.

The following take (Table 4.9) 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.

Table.4.9: Precedence and Associativity

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

For example, 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

```
x = 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.

4.5 EXPRESSIONS

Expressions in C are based on algebraic expressions. These expressions 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.

Program 4.7: Program to show use of parenthesis:

```
#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 is evaluated.
- If parenthesis is 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 is used, the expressions within parenthesis assume the highest priority.

4.6 TYPE CONVERSION

The **type conversion** or **typecasting** refers to changing an entity of one data type into another. C allows programmers 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

Let us consider the case where we want to divide two integers a and 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.

Program 4.8: Program to show type conversion:

```
#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 the example above. Another use of the typecasting is shown in the example below.

```
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. For instance, truncating a float when typecasting to an int.



CHECK YOUR PROGRESS

Q.4: 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 evaluated 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.



4.7 LET US SUM UP

- A list of C operators with their symbol is summerized below:

Arithmetic Operator:

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

Relational Operator:

Oerator	Description
<	less than
>	greater than
>=	greater than or equal
==	equal to
!=	not equal

Logical Operator:

Operator	Description
!	NOT
&&	AND
||	OR

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:**

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

- **The Conditional Operator:** The conditional operator can be used to replace ***if-else*** logic in some situations. It is also known as ternary opeartor since three opearnds are used for this opeartor. The format of conditional operator is shown below:

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 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);

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



4.8 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
2) Gottfried Byron, S; *Programming with C*; Tata McGraw-Hill Education.



4.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) 55 99

b) 150 57

c) ASCII value of 'A' is 65

Size of s1 and s2 in bytes: 1 4

Ans. to Q. No. 2: 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

Ans. to Q. No. 3: i) (d) i=7, j=7

ii) (b) 0

Ans. to Q. No. 4: i) True, ii) False, iii) True, iv) True, v) True



4.10 MODEL QUESTIONS

Q.1: What is an operator? What are the different types of operators that are included in C.

Q.2: What is an operand? What is the relationship between operator and operand?

Q.3: Describe the three logical operators included in C?

Q.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.

Q.5: What is unary operator? How many operands are associated with a unary operator?

Q.6: What is meant by operator precedence?

Q.7: What is meant by associativity? What is the asslociativity of the arithmetic operators?

Q.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();
}
```

*** ***** ***

UNIT 5: I/O FUNCTIONS

UNIT STRUCTURE

5.1 Learning Objectives

5.2 Introduction

5.3 Header Files

5.4 Formatted Input/Output Functions

5.5 Control Strings used in printf() and scanf() Functions

5.6 Escape Sequences

5.7 Unformatted Input/Output Functions

5.8 Let Us Sum Up

5.9 Further Reading

5.10 Answers to Check Your Progress

5.11 Model Questions

5.1 LEARNING OBJECTIVES

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

- learn about header file and its use
- learn formatted input function like scanf()
- learn formatted output function like printf()
- describe control string used in printf() and scanf()
- describe unformatted input functions like getch(), getche(), getchar(), gets()
- formatted input functions like putch(), putchar(), puts()

5.2 INTRODUCTION

A computer program basically consists of three sections i.e, input, processing, and output. The input section receives data from the environment through some input device like keyboard, mouse, secondary storage etc. The processing section is responsible for calculating the required data or output. Different logic like selective logic and iterative logic are implemented in this section. The last section is the output section. This

section is responsible for providing output in the appropriate manner. C language provides a set of library functions for doing these activities. These functions are predefined and stored in some file known as header file.

5.3 HEADER FILES

C language is rich in terms of library functions. The library functions are stored in a special file called **header file**. A header file is a special system file with extension.h which contains C library function declarations and **macro** definitions. It is included in a C program using #include preprocessor directives. The **#** symbol at the beginning indicates that this is for the C preprocessor. The **#include** instruct the compiler to read the specified file and execute when necessary. For example, the header file **stdio.h** should be used if you want to use the two standard I/O functions **printf()** and **scanf()**.



CHECK YOUR PROGRESS

Q.1: What are the different input devices used to input data into a program?

Q.2: What is header file? Why is it used?

Q.3: To use **printf()** and **scanf()**functions in a program _____ header file is required.

5.4 FORMATTED I/O FUNCTIONS

Formatted functions can be both input and output type. Input functions can accept data from the input device like keyboard and output functions can display data on the output device like monitor. Formatted functions allow the user to accept and display the data as per requirements. For example, if you want to display a particular data on a particular position of the screen, it is possible to do so. Here, we will discuss one formatted input function i.e., **scanf()** and one formatted output function i.e., **printf()**.

a) scanf(): Commonly used standard input formatted library function to accept data form the keyboard. It can accept data until we are

pressing the enter key or space bar.

Syntax:

```
scanf("control string", list of addresses of variables);
```

For example, scanf("%d", &n);

Here %d represent that n is an integer variable. It can read a set of variable of same type or different type.

For example,

scanf("%d%f%c", &n, &p, &ch);

where n, p and ch are the variables of type **int**, **float** and **char** respectively. Here '&' is known as 'address of' operator. It is required, because the values read form keyboard should store in some address of the memory.

```
//Program 5.1: Write a C program to find the area
of a circle:
#include<stdio.h>
#include<conio.h>
void main()
{
   float r, area;
   clrscr();
   printf("Enter the radius : ");
   scanf("%f", &r);
   area=3.14*r*r;
   printf("Area= %10.3f", area);
   getch();
}
```

b) printf(): A standard output library function to display a constant or content of a variable as per the user requirements. Library functions must be written using lower case letters. As such, printf( ) must also be written in lower case letters only. The syntax for writing printf() is:

printf(“control string”); or printf(“control string”, variable1, variable2, ..., variable n);

```
//Program 5.2: Sample C program showing the use
of printf():
#include<stdio.h>
#include<conio.h>
void main()
{
   printf("WEL-COME TO WORLD OF C ");
   getch();
}
```

5.5 CONTROL STRINGS USED IN printf() AND scanf() FUNCTIONS

A control string is a string of characters. If the first character is either % or \ then it specifies some meaning, otherwise the string displays as they are in the string. The two symbols specified are as follow:

% Conversion specification

\ Escape sequence

Conversion specification indicates the data type used in the program. The following are the different control strings used to specify different data types, which are also known as **format specifier**.

Data type	Format specifier
int	%d or %I
short signed	%d or %I
long int	%ld
long signed	%ld
long unsigned	%lu
unsigned hexadecimal	%x
unsigned octal	%o
float	%f
double	%lf

signed character	%c
unsigned character	%c
string	%s

//Program 5.3: Program to show the use of control string:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr();
   printf("%d  %f  %c", 4,4.0,'4');
   getch();
}
```

Output: 4 4.000000 4

Here, %d represents an integer value 4, %f represents a float value 4.0 and finally %c represents a character value 4. The second value 4.0 is displayed as 4.000000 instead of 4.0. This is the normal tendency of compiler that float value is always displayed with upto 6 digits after decimal point. This can be reformatted by using **field-width specifier**. The field width specifier tells printf( ) how many columns on screens should be used while printing a value.

//Program 5.4: Program to show the use of **field-width specifier:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr();
   printf("%d%5.2f%c",  4,4.0,'4');
   getch();
}
```

Output: 4 4 . 0 0 4

Here %5.2f means float value for 5 characters, out of which 2 for after decimal point, one for the decimal point itself and the remaining two for before decimal.

```
//Program 5.5:
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr();
   printf("\n%d", 4);
   printf("\n%2d", 4);
   printf("\n%3d", 4);
   printf("\n%-3d", 4);
   getch();
}
```

Here, %d means display the value without any field-width specifier, %2d means display the value with two column from right side, %3d means display the value with three columns form right side and finally %-3d means display the value with 3 columns from left side. Here, sign indicate left justification. Program 5.5. will be more clear if we execute the program in our computer and see the output.

5.6 ESCAPE SEQUENCES

Escape sequences are useful to beautify the output in the display screen. The output of a program should be well formatted, so that nobody should be confused. The following are the commonly used escape sequences in C.

Escape Sequence	Purpose	Description
\n	New line	Takes the cursor to the beginning of the next line.
\b	Backspace	Moves the cursor one position left of its current position
\f	Form feed	Moves the page on the printer to the beginning of next page.

\"	Double quote	If the character " to be printed
\\	Backslash	If the character \ to be printed
\t	Tab	Advances the remaining output one tab on the screen
\r	Carriage return	Takes the cursor to the beginning of the line in which it is currently placed.
\a	Alert	Alerts the user by sounding the speaker inside the computer.

/*Program 5.6: C program showing the use of double quote (") in a display message.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr();
   printf("\"KKHSOU\"");
   getch();
}
```

Output: "KKHSOU"

/*Program 5.7: C program showing the use of tab (\t) in a display message.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr( );
   printf("\t Education\t is\t a \t backbone\t of \t a \t society.");
   getch();
}
```

Output:

Education is a backbone of a society.



CHECK YOUR PROGRESS

Q.4: Write a program to input two numbers from the keyboard and find the sum?

Q.5: The basic pay of an employee is input through the keyboard. His dearness allowance is 119% of basic pay, and house rent allowance is 15% of basic pay. Write a program to calculate his gross salary.

Q.6: Write a program to check whether an input integer is odd or even.

5.7 UNFORMATTED I/O FUNCTIONS

Unformatted functions are easier to use. They do not require any control string to use. There are several input and output functions through which we can accept data and display them on the monitor. We can accept a single character or a set of characters. Similarly, we can display a single character as well as a set of characters using output functions. While entering data using **scanf( )** function, a few problems arise such as:

- It requires to hit the enter key to accept the data.
- It cannot accept blank character. So multiple words are not possible to accept.

The above problems are eliminated in the unformatted functions. We will discuss here some input and output functions, such as -

Input functions:

getch( )

getche( )

getchar( )

gets( )

Output functions:

putch( )

putchar( )

puts( )

getch( ): This function accepts a character from the keyboard and does not echo the typed character on the screen. It will read a character just after it is typed without waiting for the enter key to be pressed. In most of our previous examples, we have used this function. One might think that C program always end with getch( ) - which is not true. getch( ) function can be used anywhere in a C program to accept a single character. Most often this function is used to hold the execution of a program, or to see intermediate results or to stop rolling the screen.

getche(): The function getche( ) is similar to getch( ) except that getche( ) will echo the typed character on the screen whereas getch( ) does not echo the typed character on the screen.

getchar( ): getchar( ) also reads a character from the keyboard and echoes it on the screen just after it is typed. It requires the enter key to be pressed.

gets( ): It accepts a string or a group of words from the keyboard at a time which is not possible by **scanf( )** or **getch( )** . It is used to read a sentence from the keyboard. The reading process will terminate when an enter key is hit.

/*Program 5.8: C program showing the use of some input/output functions.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char a, b, c;
    clrscr();
    printf("Press any key to continue : \n");
    a=getch();
    printf("Type any character \n");
    b=getche();
    printf("Press any key and press enter key \n");
    c=getchar();
    printf("Outputs are : ");
```

```
        putch(a);
        putch(b);
        putch(c);
        getch();
    }
```

putch( ) and **putchar( )** are just opposite to **getch( )**. They are used to display a single character on the screen. **puts( )** is also just opposite to **gets( )**. It is used to display a string or a sentence on the screen.

/*Program 5.9: C program which reads name and title from the keyboard and display it. */

```
    #include<stdio.h>
    #include<conio.h>
    void main()
    {
       char name[30];
       clrscr();
       printf("Your name please: ");
       gets(name);  //name will be entered through
the keyboard
       puts("Hello ");    //display Hello
       puts(name);  //display the entered name
       getch();
    }
```



5.8 LET US SUM UP

- A computer program basically consist of three sections i.e. input, processing and output.
- Library functions are used to perform some computations and I/O operations.
- Library functions are stored in a special file called *header file*.
- Common formatted I/O functions are printf() and scanf().
- Control strings or format specifiers are used to specify data type.
- Escape sequences are used to beautify output of programs.

- Unformatted I/O functions do not requires any control string.
- Common unformatted I/O functions are getch(), getche(), getchar(), gets(), putch(), puts(), putchar().



5.9 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
2) Gottfried Byron, S; *Programming with C*; Tata McGraw-Hill Education.



5.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: Different input devices like keyboard, mouse, secondary storage etc. are used to input data into a computer program.

Ans. to Q. No. 2: A header file is a special system file with extension .h which contains C library function declarations and **macro** definitions. It is used to use some library function into a program.

Ans. to Q. No. 3: To use **printf()** and **scanf()**functions in a program stdio.h header file is required.

Ans. to Q. No. 4:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a, b, c;
    clrscr();
    printf("Enter the first number :\n");
    scanf("%d", &a);
    printf("Enter the second number :\n");
    scanf("%d", &b);
    c=a+b;
    printf("The sum is %d", c);
    getch();
```

```
}
```

Ans. to Q. No. 5:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int bpay, hra, da, gross;
    clrscr();
    printf("Enter the basic pay :\n");
    da=bpay*1.19;
    hra=bpay*.15;
    gross=bpay+da+hra;
    printf("The gross salary is %d", gross);
    getch();
}
```

Ans. to Q. No. 6:

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



5.11 MODEL QUESTIONS

Q.1: What is control string?

Q.2: Explain the different format specifiers used in C language?

Q.3: What are the uses of escape sequence?

Q.4: Differentiate between formatted and unformatted function.

Q.5: The temperature of a city in Fahrenheit degrees is input through the keyboard. Write a program to convert this temperature into Centigrade degrees.

Q.6: If a four-digit number is input through the keyboard, write a program to reverse the number.

Q.7: Any year is input through the keyboard. Write a program to determine whether the year is a leap year or not.

Q.8: A number is entered through the keyboard. Write a program to find the reversed number and to determine whether the original and reversed numbers are equal or not.

Q.9: Any character is entered through the keyboard, write a program to determine whether the character entered is a capital letter, a small case letter, a digit or a special symbol.

Q.10: Write a program to calculate overtime pay of an employee. Over time is paid at the rate of Rs. 10.00 per hour for every hour worked above 8 hours. Assume that employees do not work for fractional part of an hour.

*** ***** ***

UNIT 6: CONDITIONAL STATEMENTS

UNIT STRUCTURE

6.1 Learning Objectives

6.2 Introduction

6.3 Decision Control Statements

6.4 Conditional Branching Statements

- 6.4.1 *if* Statement
- 6.4.2 *if-else* Statement
- 6.4.3 *Nested if-else* Statement
- 6.4.4 *switch* Statement

6.5 *break* Statement

6.6 *continue* Statement

6.7 *goto* Statement

6.8 Let Us Sum Up

6.9 Further Reading

6.10 Answers to Check Your Progress

6.11 Model Questions

6.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- learn about decision control statements
- use conditional and uncondional branching statements in programming
- learn about *if*, *if-else*, *switch* statements
- learn to use break, continue and goto statements.

6.2 INTRODUCTION

In the previous units, we have seen that the C language has a collection of library functions, which include a number of input/output functions. Moreover, the instructions were executed in the same order in which they appeared within a program. Each instruction was executed once

and only once. But C language provides the facilities of decision making 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 learn about decision control statements that can alter the flow of a sequence of instructions.

6.3 DECISION CONTROL STATEMENTS

We have already been acquainted with simple C programs in the previous units. We have seen that the code in the C program is executed sequentially from the beginning to end of the program. But in some situations, we may want only selected statements to be executed. In such cases, we use decision control statements.

C language supports two types of decision control statements. These includes conditional and unconditional branching.

6.4 CONDITIONAL BRANCHING STATEMENT

While writing computer programs we usually instruct the computer to check various kinds of situations and to act accordingly. The computer performs comparisons of various kinds of statements. The conditional branching statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not.

Conditional branching statements include *if*, *if-else,* nested *if-else, switch* statements.

6.4.1 *if* Statement

If statement is the most popular and simple 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 to a block of statements.

The **if statement** evaluates the test expression inside parenthesis. If the test expression is evaluated to true (nonzero), then the statements inside the body of if is executed. If test expression

is evaluated to false (0), statements inside the body of if is skipped.

In C any ***non-zero*** integer value is treated as ***true*** and ***zero*** is treated as ***false***. The syntax of simple ***if*** statement is as follows:

if(test condition)

{

/*statement(s) that will execute if the condition is true.*/

}

The parentheses { } used in if statements are not necessary if the statement block contains only one statement.

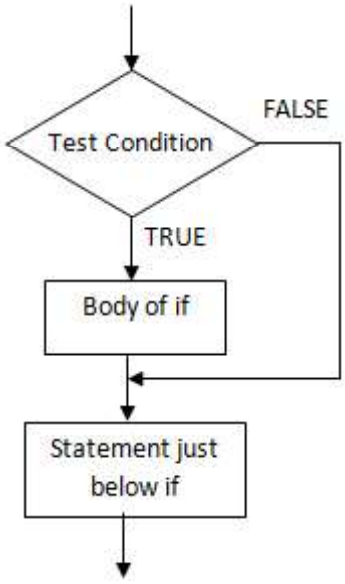


Fig. 6.1: Control transfer in if statements

Program 6.1: Write a C program to check whether the entered number is positive or negative

Solution:

```
#include<stdio.h>
#include<conio.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 " );
   getch();
}
```

Output:

Enter the number 3

The Number is Positive

6.4.2 *if-else* Statement

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

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

If the test condition is true, then the body of **if** will be executed and if the test condition is false, then the statements in the **else** part will be executed.

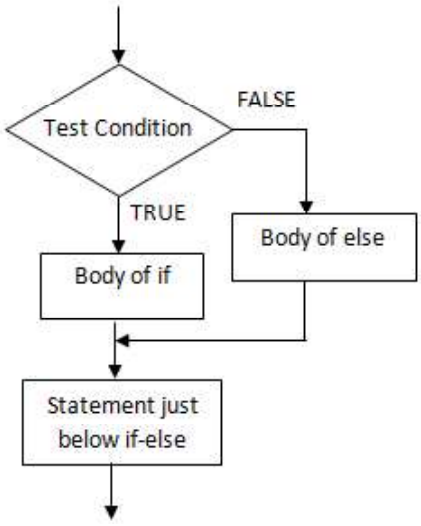


Fig 6.2: Control transfer in if-else statements

Program 6.2: Write a C program to print the larger of the two given numbers.

Solution:

```
#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 be 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 be 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

Let us consider a program for checking whether a year is a leap year or not. Leap year is a special year containing one extra day i.e., total number number of days in a leap year is 366 days. If a year is exactly divisible by 4 and not divisible by 100 then it is a leap year. Again, if the year is exactly divisible by 400 then it is a leap year, otherwise it is a common year.

Program 6.3: Write a C program to check whether a year is leap year or not.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int year;
   /* Read year from user */
   printf("Enter year : ");
   scanf("%d", &year);
   // Check for leap year
   if(((year%4 == 0) && (year%100 !=0)) ||
(year%400==0))
   {
      printf("LEAP YEAR");
   }
      else
   {
      printf("COMMON YEAR");
   }
      getch();
}
```

If we enter 2004, 2016 the output will be LEAP YEAR. And if we enter year like 2005, 2010, 2015, then the output will be COMMON YEAR.

6.4.3 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
}
```

Program 6.4: The program in *Program 6.1* can be rewritten in nested if-structure in following ways

Solution:

```
#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

6.4.4 *switch* Statement

The main disadvantage of ***if-else*** statements is that they are too complex to understand, read and debug when there are multiple conditions for testing. ***Switch case*** statements are a substitute for long **if** statements that compare a variable to several "integral" values. The basic format for using *switch-case* is outlined below.

switch (<variable/expression>)
{
case *label1*:
/* statements to be executed if <variable> = = *label1** **break;**
case *label2*:
/* statements to be executed if <variable> = = *label2 */*
break;
...
...
default:
/* statements to be executed if <variable> does not equal the value following any of the cases*/
}

The ***switch*** statement checks the value of ***variable/ expression*** against the list of ***case labels*** and when a match is found, the block of statements associated with that ***case*** is executed. The labels in the syntax are constants or expressions. 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 statements in the ***default*** section.

An important point to remember about the ***switch*** statement is that the ***case*** values may only be constant integral expressions. The ***default*** case is optional, but it is wise to include it as it handles any unexpected ***cases***. ***Switch*** statements serve as a simple way to write long ***if*** statements when the requirements are met. Let us consider an example for a better understanding of the switch statement.

There are some rules that should be followed while writing **switch** statement. These rules are:

- The **<variable/expression>** used in a switch statement must have an integral type.
- We can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression (labels in the syntax) for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being used is equal to a case, constant or literal, the statements following that case will execute until a break statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional ***default*** case, which mostly appears at the end of the switch. The default case can be used for performing a task when none of the cases is true.

Program 6.5: Write a program to convert a single digit number into words. [For example if you enter 5, the output should be "Five"].

```
#include<stdio.h>
#include<conio.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

6.5 *break* STATEMENT

The **break** statement generally appears inside switch body or a loop body. In C, the break statement is used to terminate the execution of the current enclosing switch or loop body in which it appears. The general format of the break statement is:

break;

The *break* is a keyword and a semicolon must be inserted after the word *break*. We have already seen its usage in the *switch* statement.

Break statement is widely used in loop. When the compiler encounters a *break* statement, the control passes to the statement that follows the loop.The following program Program 6.6 shows the use of *break* statement inside *for* loop. In a loop, the *break* statement is mostly associated with an *if* statement.

Program 6.6: Write a program to show use of break statements inside for loop.

Solution:

```
#include <stdio.h>
#include<conio.h>
void main()
{
   int i;
   for (i = 1; i <= 10; i++)
   {
      printf ("%d, ", i);
      if (i == 4)
      break;
   }
   getch();
}
```

When we run the program, the output will be 1, 2, 3, 4, although the program is written for display from 1 to 10. In the *for* loop, the counter **i** is from 1 to 10. As soon as i becomes equal to 4, the *break* statement is executed and the control jumps out of the loop.

Hence, the break statement is used to exit a loop from any point within its body, bypassing its normal terminating condition.

6.6 *continue* STATEMENT

Similar to the *break* statement, the ***continue*** statement can only appear in the body of a loop. 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;

When the compiler encounters a continue statement then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the next iteration.

The use of *continue* statement in loops is shown in the following syntax. Like *break* statement, *continue* statement is also associated with an *if* condition.

```
while (test-condition)
{
    ……
    if(……..)
    continue;
    ……
    ……
}
do
{
    ……
    if(……..)
    continue;
    ……
```

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

For example, if we want to display the numbers from 1 to 10 except 7, then we can use *continue* statement in the following way:

Program 6.7: Write a program to show use of continue statement.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i;
   for(i=1;i<=10;i++)
   {
      if(i==7)
      continue;
      printf("%d\t",i);
   }
   getch();
}
```

The output of the program will be 1 2 3 4 5 6 8 9 10.

As soon as **i** becomes equal to 7, the *continue* statement is encountered, so the rest of the statements in the for loop are skipped and the control passes to the next iteration that increments the value of i.

Thus, the ***continue*** statement forces the next iteration to take place, skipping any code in between itself and the test condition of the loop.

6.7 *goto* STATEMENT

C language supports an unconditional control statement that is ***goto***. The ***goto*** statement is used to tranfer control to a specified ***label***. It is used to transfer the control in a program from one point to another point unconditionally. Therefore, **goto** is called unconditional branching statement. The syntax of the goto statement is:



The following program shows the use of *goto* statement with the help of *for* loop.

Program 6.8: Write a program to show use of goto statement.

Solution:

```
#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

It is a good programming practice to use the *break* and *continue* statements in preference to *goto* whenever possible. Use of *goto* statement makes it difficult to trace the control flow of a program and thus the program becomes complicated and hard to modify.



CHECK YOUR PROGRESS

Q.1: Fill in the blanks:

i) In C any _____ integer value is treated as true and _____ is treated as false.

ii) When a _____ statement is reached, the *switch* statement terminates, and the flow of control jumps to the next line following the switch statement.

iii) We can have _____ number of case statements within a switch.

iv) If none of the cases is matched with the variable in the switch statement, then the statement under _____ will be executed.

v) The switch case control expression must be of _____ type.

vi) The _____ statement is used to transfer control to a specified label.

Q.2: Choose the appropriate option:

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

Q.3: What will be the value of x in the following code:

```
#include <stdio.hi>
#include<conio.h>
void main ()
{
    int x, y = 10;
    x = (y < 10) ? 30 : 40;
    printf( "Value of x: %d", x);
    getch();
}
```



6.8 LET US SUM UP

- Conditional statements are used to execute a statement or a group of statements based on certain conditions. *if*, *if-else*, *switch* statements are conditional statements.
- In an *if* statement, if the condition is *true* the statements inside the parenthesis { }, will be executed, else the control will be transferred to the next statement after *if.*
- In an *if-else* statement, if the condition is *true* the statements between *if* and *else* is executed. If it is *false* the statement after *else* is executed.
- The *switch case* statements are often used as an alternative to long *if* statements that compare a variable to several integral values. It matches the value in variable with any of the cases inside, the statements under the case that matches will be executed.
- *Default* is a case that is executed when the value of the variable does not match with any of the values of the *case* statement.
- The *break* statement causes an immediate *exit* from the innermost loop structure.

- The *continue* statement causes the loop to be continued with the next iteration after skipping any statement in between.
- The *goto* statement is used to transfer the control in a program from one point to another point unconditionally.



6.9 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.
2) Thareja, R. (2012); *Computer Fundamentals & Programming in C*; OXFORD University Press.



6.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) non-zero, zero, ii) break, iii) any, iv) default, v) integral, vi) goto

Ans. to Q. No. 2: c) to choose from multiple possibilities which may arise due to different values of a single variable.

Ans. to Q. No. 3: Value of x: 40



6.11 MODEL QUESTIONS

Q.1: What is the role of conditional statements in programming?

Q.2: Explain the importance of switch case statement. In which situations is a switch case desirable?

Q.3: Explain the usefulness of default statement in switch case statement.

Q.4: Write a program to print the prime factor of a number.

Q.5: Write a program to test if a given number is a power of 2.

Q.6: Write a program using switch case to display a menu that offers five options: read three numbers, calculate total, calculate average, display the smallest and display the largest value.

Q.7: Write a C program to check whether a number is even or odd.

Q.8: Write a C program to check whether a character is alphabet or not.

Q.9: Write a C program to input any alphabet and check whether it is vowel or consonant.

Q.10: Write a C program to input any character and check whether it is alphabet, digit or special character.

Q.11: Write a C program to check whether a character is uppercase or lowercase alphabet.

Q.12: Write a C program to input marks of five subjects Physics, Chemistry, Biology, Mathematics and Computer. Calculate percentage and grade according to following:

Percentage > 90% : Grade A

Percentage > 80% : Grade B

Percentage > 70% : Grade C

Percentage > 60% : Grade D

Percentage > 40% : Grade E

Percentage < 40% : Grade F

Q.13: What is the similarity and difference between break and continue statements?

Q.14: What are decision control statements? Explain in detail.

*** ***** ***

UNIT 7: LOOP CONTROL STRUCTURES

UNIT STRUCTURE

7.1 Learning Objectives

7.2 Introduction

7.3 Loop Control Statements

- 7.3.1 *while* Loop
- 7.3.2 *do-while* Loop
- 7.3.3 *for* Loop

7.4 Let Us Sum Up

7.5 Further Reading

7.6 Answers to Check Your Progress

7.7 Model Questions

7.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- learn about the concept of loop
- learn to write three different types of loops
- use while, do-while and for loop in programming
- learn to write loop within a loop.

7.2 INTRODUCTION

In the previous unit we have learned about conditional statements. We have seen that the C language is accompanied by a collection of statements and functions. Moreover the instructions were executed in the same order in which they appeared within a program. Each instruction was executed once and only once. But 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 various control statements available in C language.

7.3 LOOP CONTROL STATEMENTS

The repetation process within a computer program is known as ***iteration***. C language supports three types of iterative statements, also known as ***loop control statements***.

Loop control statements are used to execute and repeat a block of statements depending on the value of a condition. There are three types of loop control statements in C language. They are:

- **while loop**
- **do-while loop**
- **for loop**

7.3.1 *While* loop

The while loop provides a mechanism to repeat one or more statements while a particular condition (i.e., test expression) is true. The syntax of *while* loop is as follows:

```
statement x;
while(condition)
{
        statement block;
}
statement y;
```

In case of *while* loop, the condition is tested first. If the condition is *true*, only then the statement block will be executed otherwise if the condition is *false*, the control will jump outside the *while* loop block.

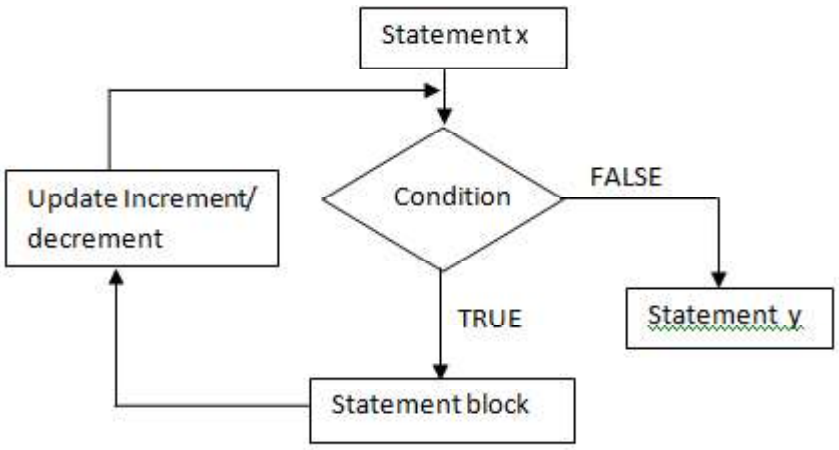


Fig. 7.1: Flow chart of while loop

From the diagram it is clear that the *while* loop will execute as long as the condition is *true*. The updation of *increment/decrement* statement is necessary.

Again if the condition never becomes *false* then it will become an infinite loop which is never desirable. If the condition evaluates to false, then the statements enclosed in the loop are never executed.

- The braces { } can be omitted when there only one statement available in the statement block.
- The initialization of variable takes place only once to initialize the value of the variable.
- The condition and increment/decrement are executed on each iteration.

For example, in the following program 7.1, while loop is used while calculating the factorial of a number.

```
//Program 7.1: Program for finding the factorial
of a number.
#include <stdio.h>
#include<conio.h>
void main()
{
   int number;
   long factorial;
   printf("Enter an integer: ");
```

```
   scanf("%d",&number);
   factorial = 1;
   // loop terminates when number is less than or
equal to 0
   while (number > 0)
   {
   factorial = factorial*number;  // factorial =
factorial*number;
   number--;
   }
   printf("Factorial= %ld", factorial);
getch();
}
```

7.3.2 *do…while* STATEMENT

The *do-while* loop is similar to *while* loop except that in a *do-while* loop, the test condition is tested at the end of the loop. Since the test condition is tested at the end, this implies that the body of the loop gets executed at least once even if the condition is false. The general form of writing *do-while* loop is as follows:

```
statement x;
do
{
      statement block;
} while(condition);
statement y;
```

In case of *do-while* loop it should be remembered that test condition is followed by a semicolon. The curly bracket is optional if there is only one statement in the body of the loop. The *do-while* loop continues to execute while the condition is true. When the condition becomes false, the control will jump to statement following the *do-while* loop. The major difference of *do-while* loop with *while* loop is that it always executes at least once.

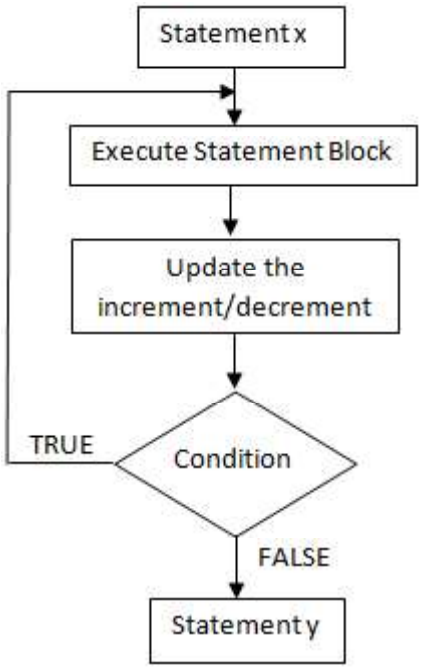


Fig. 7.2: Flow chart of *do-while* loop

```
//Program 7.2: Program to display all the even numbers less than or equal to10 using do-while loop.
#include<stdio.h>
#include<conio.h>
void main()
{
   int c=2;
   do
   {
      printf( "%d  " , c) ;
      c=c+2;
   } while( c < = 10 );
   getch();
}
```

Output: 2 4 6 8 10

```
//Program 7.3: Program to compute the average of first n numbers using do-while loop.
#include<stdio.h>
```

```
#include<conio.h>
void main()
{
   int n,i=0,sum=0;
   clrscr();
   float avg=0.0;
   printf("\nEnter the value of n:");
   scanf("%d", &n);
   do
   {
      sum=sum+i;
      i=i+1;
   }while(i<=n);
   avg=sum/n;
   printf("\nThe sum of first n numbers=%d",sum);
   printf("\nThe average of first %d
numbers=%f",n,avg);
getch();
}
```

Output: Enter the value of n: 5

The sum of first n numbers= 15

The average of first 5 numbers = 3.000000

7.3.3 *for* loop

The ***for*** loop consists of three different statements separated by semicolon: ***Initialization***, ***Test condition*** and ***Increment/ Decrement statement*** for updation of loop variable. Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting it to some other values according to our requirement.

When a *for* loop is executed, the loop variable is initialized only once. With every iteration of the loop, the loop variable is updated and the condition is checked. If the test condition is *True*, the statement block of the loop is executed, else the statements

comprising the statement block of the loop are skipped and the control jumps to the immediate statement following the *for* loop body. The general syntax and flow chart of *for* loop is given here:

```
for (initialization; test condition; increment/decrement statement)
{
    statement block ;  //codes to be executed;
}
statement y;
```

Some important point regarding for loop:

- Every section of the for loop is separated from the other with a semicolon.
- It is possible that one of the sections may be empty, though the semicolon still have to be there.
- In case all the expressions are omitted, then there must be two semicolons in the for loop. This is as follows:

```
for( ; ; )
```

- In case of for loop, the condition is tested before the statements contained in the body are executed. So if the condition does not hold *True*, then the body of the loop may never get executed.
- Multiple conditions in the test expression can be tested by using the logical AND (&&) and logical OR (||) operators.
- Multiple initializations must be separated with a comma operator as shown in *Program 7.5*:

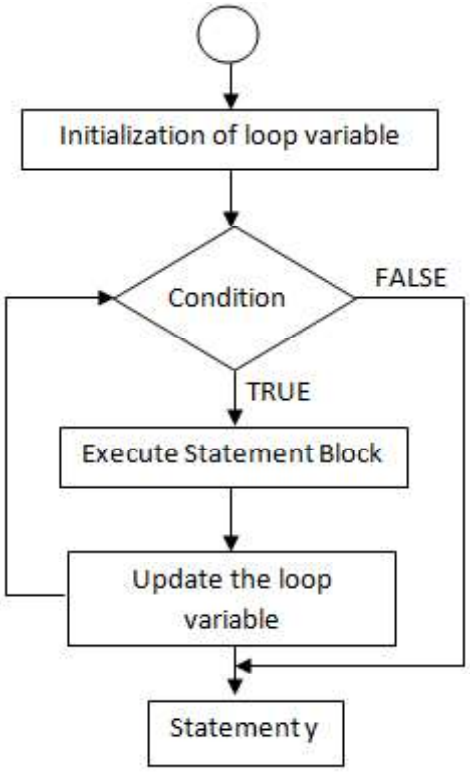


Fig.7.3: Flow chart of *for* loop

In a for loop *initialization* statement allows the programmer to give a value. *Test condion* is tested before each iteration. The *condition* is a relational expression. If *condition* evaluates to *True* (i.e., nonzero) the next iteration occurs, otherwise the loop is terminated. *Increment* or *decrementor* statement is executed at the end of each iteration. A simple example of displaying natural numbers upto 10 is shown using for loop.

```
/*Program 7.4: Write a C program displaying 10
natural numbers using for loop*/
#include<stdio.h>
#include<conio.h>
void main()
{
   int n, i;
   printf ( " The first 10 natural numbers are: "
) ;
   for ( i = 0 ; i < 10 ; i++ )
   printf ( " %d\t ", i );
```

```
getch();
}
```

Output: Enter the Limit : 10

0 1 2 3 4 5 6 7 8 9

In the above example the first loop variable **i** is initialized as **0**, then check the condition, is the value of **i** is less then **10**. For the first execution, the condition results are ***True***. The control will execute the *printf* statement which will display the value **0**. Next the value of loop variable **i** is incremented by **1** with the increment statement i++. Thus, the current value of loop variable **i** is now **1**. Again, check the condition whether **1<10**. Since the condition is *True*, therefore the *printf* statement will be executed and display **1**. Again increment the loop variable by **1** and repeat the same procedure until the condition evalutes to ***False***.

```
/*Program 7.5: Program showing multiple
initializations separated by comma operator */
#include<stdio.h>
#include<conio.h>
void main()
{
   int i, sum;
   clrscr();
   for( i=1, sum = 0; i<=10; i++ )
      sum=sum+i;
   printf("\nSummation of 1 to 10 is %d",sum);
getch();
}
```

In the Program 7.5, i is initialized to 1 and sum is initialized to 0 separating it with a comma. The output will give the sum of 1 to 10 which is 55.

- If the loop control variable is updated within the block of statement, then the third part can be skipped.

- In some situation,we are required to write for loop with two semicolons only i.e., no initialization, test condition and updating of loop control variable. Then the for loop become an infinite loop if no stopping condition is specified in the body of the loop. For example, the following program segment will infinitely display the word "KKHSOU" on the computer screen.

//Program 7.6:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr();
   for( ; ; )
   {
      printf("KKHSOU\n");
   }
getch();
}
```

- Floating point variable should not be used as loop control variable.
- While writing a loop, the following points are to be added :
  - Declaration of a loop control variable (i.e.,a counter)
  - Statements inside loop body
  - Evaluating the test expression (i.e., conditions)
  - Incrementing or decrementing the loop control (counter) variable.



CHECK YOUR PROGRESS

Q.1: Consider the following code fragment:

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

How many times will the loop be executed–

a) Infinite b) 9 c) 4 d) 0

Q.2: How many times will the following loop be executed-

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

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

Q.3: 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);
}
```

Q.4: Write a C program to check whether a given number is a palindrome or not.



7.4 LET US SUM UP

- Loop directs a program to perform a set of operations again and again until a specified condition is achieved.
- There are three types of loops: while, do-while and for loop.
- Every loop has a start value, a step value and a stop value., i.e., there are three sections of a loop: *inialization*, *condition* and *updation of loop control variable*.
- The *while* and *do-while* loop constructs are more suitable in situations where prior knowledge of the terminating condition is not known.
- *While* loop evaluates a test expression before allowing entry into the loop, whereas *do-while* loop is executed at least once before it evaluates the test expression which is available at the end of the loop.
- The *for* loop construct is appropriate when in advance it is known as to how many times the loop will be executed. It is suitable for a finite loop.



7.5 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.
2) Thareja, R. (2012); *Computer Fundamentals & Programming in C*; OXFORD University Press.



7.6 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) Infinite

Ans. to Q. No. 2: b) 25

Ans. to Q. No. 3: 0 5 15 30
sum = 30

Ans. to Q. No. 4:

```
#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”);
}
```



7.7 MODEL QUESTIONS

Q.1: How many times does the loop iterate?

```
for(i=0; i=10;i = i+2)
    printf(“Guwahati\n”);
```

a) 5 b) 10 c) 2 d) none of the above

Q.2: How many times does the loop occur?

```
i=0;
while(i<5)
        printf("%d\n",i++);
```

a) 5 b) infinite c) 4 d) 6

Q.3: Write a C program using do while loop, to calculate the sum of every third integer, beginning with i=2, for all values of i that are less than 50.

Q.4: Write a C program to reverse a given integer (for example, 17536 is reversed as 63571).

Q.5: Write a C program to check whether the given number *n* is a prime number.

Q.6: Write a C program to generate the fibonacci series: 0 1 1 2 3 5 8... upto n.

Q.7: Write a C program to display the following pattern:

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

Q.8: Write a C program to display multiplication table from 1 to 5 upto 10 numbers.

*** ***** ***

UNIT 8: STORAGE CLASS

UNIT STRUCTURE

8.1 Learning Objectives

8.2 Introduction

8.3 Storage Class

8.4 Automatic Variable

8.5 External Variable

8.6 Static Variable

8.7 Register Variable

8.8 Let Us Sum Up

8.9 Further Reading

8.10 Answers to Check Your Progress

8.11 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- learn about Storage Class
- describe Automatic, External, Static and Register Variable
- describe the Scope of Variables
- define lifetime of a Variable

8.2 INTRODUCTION

We already have some basic idea about variables. Variables can be defined as 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.

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.The storage class also 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. In this unit, we will discuss the various storage classes.

8.3 STORAGE CLASS

Storage class is related to the declaration of variable, function, and parameters that we use in C programs. The storage class in the function is used when returning a value of a particular data type from the function. 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 parameters in which storage class is going to be used. Depending upon the above, storage class can be classified into four categories:

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

Now, let us try to understand each storage class in the next section using examples.

8.4 AUTOMATIC VARIABLE

We already know how variables are used in C program. Now, we are going to use storage class in the variable declarations. Actually, we have already used automatic storage class in the programs in the previous units. Let us take an example as shown below:

Program 8.1: Program to illustrate the use of automatic variable.

```
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);
    }
```

By default all the variables declared are automatic.We can also explicitly define the variables using **auto** keyword.

Let us look at the special properties of automatic storage that make it different from other storage class. The automatic variable has the following characteristics:

a) Storage: The value of the variable is stored in the memory of the computer (not in register).

b) Default intial value: The default initial value for automatic variables is garbage value i.e. any unpredictable value. It means that if the variable is not initialized then the variable contains some useless value.

c) Scope: The scope means the availabilty of the variable. Automatic variable is local to the block in which the variable is declared. Outside this block the variable can not be accesed.

d) Life time: The life time of this variable is within the block it is declared.

Program 8.2: Program to illustrate the use of automatic variables.

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

Output: i=10

j=8214 (or some other garbage value)

Since here we initialized **i** to 10 explicitly, therefore the value of **i** is

displayed as 10; But in case of **j,** as we do not assign any value so garbage value is displayed at output. This example describes about the default intial value of auto variables. Now, let us understand about the life time and scope of automatic variables using another example.

Program 8.3: Program to show lifetime and scope of automatic variables.

```
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. The first inner block is executed and therefore 3 is displayed as the first output as in this block **i=3.** After that the second inner block is executed and 2 is displayed as in the second inner block the value of **i** is 2**(i=2)**. Next, we come to the outer block where the value of **i** is 1**(i=1)** so 1 is displayed as the last output. We have seen that the value of **i** is different in each block.Whatever value we have initialized **i** with, it remains valid only within the block where we have declared it. This is known as **scope** of the variable.

8.5 EXTERNAL VARIABLE

We have now 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 fits this need. The properties of the external variable are:

a) **Storage:** The external storage variables are stored in memory.
b) **Default initial value:** The default initial value for external variables is zero.
c) **Scope:** The scope for external variables is global i.e., the variable is available to all funcitons and blocks within the program.
d) **Life time:** The lifetime of the variables is until the program's execution stops.

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 functions. Let us understand the scope and life time of external variables using examples.

Program 8.4: Program to illustrate the concept of external variables.

```
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

In the above pogram, the 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, here 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

Q.1: Write true or false:

a) The default initial value of external variable is same as with automatic variable.

b) By default the variables declared are automatic.

c) The storage for the automatic variable is in the registers.

Q.2: Write down one similarity between automatic and external variables.

8.6 STATIC VARIABLE

Static variable is 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 for static variables the value does not disappear when the function is no longer active.The last updated value for static variable always persists. That is, when the control comes back to the same function again, the static variables have the same value as they left at the last time. Properties of static variables are:

a) **Storage:** The static variables are stored in memory.

b) **Default initial value:** The default initial value for static variables is zero.

c) **Scope:** The scope of static variables is global.

d) **Life time:** The life time of static variables is untill the program's execution does not stop.

Program 8.5: Program to illustrate the concept of static variable.

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

Output: 4

5

It can be observed 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 the 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() function, 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 output of the program after the modifications should be: 4

4

The 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.

8.7 REGISTER VARIABLE

We have discussed about the automatic, static, external storage classes in earlier sections. Each class stores the variables in the memory of the computer. We know that there are mainly two areas for storing data in the computer– Memory and CPU registers. Accessing data from the register is faster than from memory. This makes the program to run faster. Register storage class makes it possible for the program to run faster. We use the register class with variables which accessed frequently like loops (such as for, do-while etc.).

The characteristics of register variables in terms of scope, life time etc are:

a) **Storage:** The register variables are stroed in CPU registers.
b) **Default initial value:** The register variables take garbage values as the default values.
c) **Scope:** The scope of register variables is local to the block in which it is declared.
d) **Life time:** The life time of register variables is till the control remains within the particular block where it is declared.

Program 8.6: Program to illustrate the concept of register storage class variables.

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

```
}
```

A question that may arise in your mind here, is that if we declare most of the variables as register then every program should run faster. But then why do we not always use this concept. This is because the number of registers is limited in a computer system and so we can not use register class for all variables. If in a program the total number of register variables exceeds the system register quantity; then all the variables that exceed are by default declared as automatic.

For example, if we write a program with a loop that uses 20 register variables (assume) and the computer has only 16 CPU registers, then the rest of the 4 variables are automatically transformed to automatic storage class variables. A important point to be noted here, is that register storage can not be used for float and double data type since CPU registers usual capacity is 16 bit and both float and double data types require 4 byte (4 x 8 = 32 bit) and 8 byte (8 x 8 = 64 bit) storage respectively.



CHECK YOUR PROGRESS

Q.3: Identify from the following which statements are 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.

Q.4: Write one difference between register and auotmatic storage class variable.



8.8 LET US SUM UP

- Storage class of a variable determines the variable location, default intial value,scope and life time of the variable.
- By default all the variables declared are aotomatic in nature.
- Except for the register storage class, all other storage class variables are located in the memory.

- Static storage variables retain their previous value during program executions. Static class variable are usually used when the program needs to share a variable.
- Since locations of register storage class variables are in the CPU registers, so those programs with register variables run faster than the programs where other class variables are used.
- The number of CPU registers is limited in a computer system so we can not use register class for all variables.



8.9 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
2) Gottfried Byron, S; *Programming with C*; Tata McGraw-Hill Education.



8.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) False, b) True, c) False

Ans. to Q. No. 2: One similarity between automatic and external variables is that the storage location for both class variables are in the memory.

Ans. to Q. No. 3: a) False, b) True, c) False.

Ans. to Q. No. 4: Register class variable stores value in CPU registers whereas automatic class variable use memory for storing data.



8.11 MODEL QUESTIONS

Q.1: Define storage class of a variable. What are the different types of storage class?

Q.2: Compare external and automatic storage class variables.

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

Q.4: Why are register storage class variables used with loop counter variables?

Q.5: Mention the common factor among automatic, static and external variables.

*** ***** ***

UNIT 9: ARRAY

UNIT STRUCTURE

9.1 Learning Objectives

9.2 Introduction

9.3 Array

9.3.1 Terminologies Associated with Array

9.4 Array Declaration and Initialization

9.5 One-Dimensional Array (1-D Array)

9.5.1 Entering Data Values in 1-D Array

9.5.2 Accessing Values from 1-D Array

9.6 Two-Dimensional Array (2-D Array)

9.6.1 Storage Representation of 2-D Arrays

9.6.2 Entering Data Values in 2-D Array

9.6.3 Accessing Values From 2-D Array

9.7 Let Us Sum Up

9.8 Further Reading

9.9 Answers to Check Your Progress

9.10 Model Questions

9.1 LEARNING OBJECTIVES

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

- define an array
- declare and initialize array
- create and access one dimensional array
- create and access two dimensional array.

9.2 INTRODUCTION

In the previous unit, we have learnt about the different storage class. We have also learnt about conditional statements and loop control structures in the earlier units.

You must have come across the term ***array*** many times and wondered what it is and where it can be applied.

In this unit we will learn to define an array. We will also learn to declare an array and to initialize an array. In addition to these, different types of arrays like one dimensional and two dimensional arrays will also be covered in this unit.

9.3 ARRAY

Array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations. In this definition by 'finite', we mean that the array contains a fixed number of elements. By 'ordered' we mean that all the elements are stored in contiguous locations of the computer memory in a linear way. By 'homogeneous' we mean that all the elements in the array must belong to the same data type.

9.3.1 Terminology

Let us look at some of the terminologies associated with array.

- **Size:** The number of elements in the array.
- **Type:** The type indicates the data type of the array. It can be integer, floating point or character.
- **Base:** The base of the array is the address of the first element of the array.
- **Index:** Elements in an array are referred using a subscript or index value. The index is an integer value which gives the position of the element in an array. It is denoted by $\mathbf{A_i}$ or **A [i]** where **A** is the name of the array and **"i"** is the subscript or index. Since array elements are identified by using index or subscripts, the array is also called an indexed or subscripted variable.

9.4 ARRAY DECLARATION AND INITIALIZATION

Declaration of Array: An array can be declared just like we declare any other variable. We give the data type of the variable followed with the name of the variable. In case of an array also we give the data type followed with the name of the array but we also include an additional component that

is the size of the array. The number of elements that the array can contain needs to be declared at the beginning of the array. This is because according to the definition of an array, it contains a fixed number of elements.

We can declare an array in the following way:

data_type name_of_array [size_of_array];

In the above declaration syntax, data type can be any of the valid data types for variables. The data type can be integer, floating point or a character. It is followed by the name of the array variable and the size of the array that is given inside square brackets. For example:

int array [5] ;

char arr [10] ;

float balance [3] ;

Need for arrays: To stress the need for arrays let us look at the following example.

A cricket match has been organized between two teams A and B. Suppose we do not have the knowledge of arrays and we need to keep the batting scores of all the players for the team batting first. If we want to keep the batting score for one player we can declare a variable like the one below:

int bat_score1;

where we store the score of player 1. To store the score of rest of the 10 players we will need to declare 10 more variables of data type integer as given below:

int bat_score2, bat_score3, …. bat_score11;

Now if, instead of keeping the scores of just 11 players for one match suppose we are required to keep score of 3 teams. How many variables do we create using the above approach? Do we create 33 integer variables?

An easier way to keep the scores of the team would be to use an array. Instead of declaring 11 variables for keeping the score of 11 players in a team wouldn't it be easier to store all the batting values of one team for one match in one array variable.

```
int     team_a [ 11 ];
int     team_b [ 11 ];
int     team_c [ 11 ];
```

Definition of array: When we declare an array, a contiguous memory location is allocated to that array. Let us look at an example of the physical representation of array of size 10 in computer's memory.

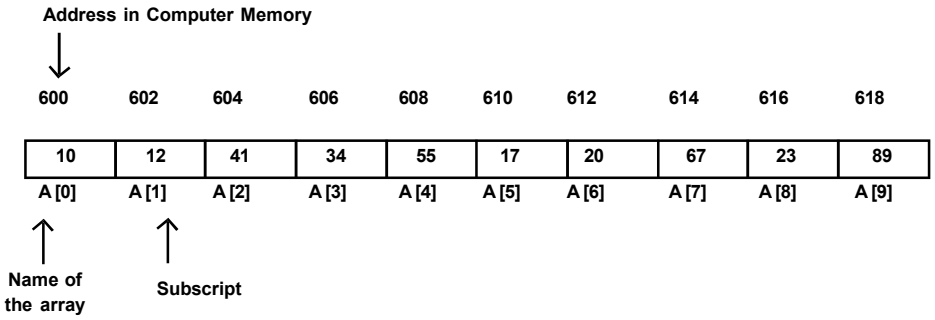


Fig. 9.1 Physical representation of array in memory

In the above figure we can see how array is actually stored in the memory. Let us consider an integer array of ten elements. Here, the name of the array is A. The smallest index of an array is called a lower bound and the highest index is called an upper bound. In case of C, the lower bound of an array is 0 and the number of elements can be calculated as the difference between upper and lower bound plus 1.

No of elements = Upper – Lower + 1

The number of elements in the above case will be (9-0+1=10) ten elements. The base address of the array in this case is 600, since that is the address of the first element in the array. If we assume that the compiler to store an integer value needs two bytes of storage then the address of the second element is 602. Similarly, the rest of the elements are also stored continuously taking two bytes of storage per integer number.

Array initialization: We can initialize the elements of the array when we declare the array. Initialization is generally done when we already know the values of the elements of an array. Just as in declaration, in initialization also we give the data type of the array, followed by the name and size of the array. But in addition to these we also provide the values of the data elements of the array within braces **{ }** separated by commas. Let us look at how initialization is done using the following example:

int A [5] = { 1, 2, 3, 4, 5 };

In the above example, the array A is declared and initialized. The array A has a size of 5 elements and the values of the five elements have been initialized as follows:

A [0] = 1

A [1] = 2

A [2] = 3

A [3] = 4

A [4] = 5

The first value that is given in braces is put into the first array location with subscript value 0. So 1 is kept at location A[0]. Similarly, the second value inside braces is put into the second array location with subscript value A[1] and so on till all the given values has been put in the array.

When initialization of values is done, a possibility of not mentioning the array size in the square brackets [ ] beforehand is also provided. That is, we can leave the square brackets empty. For this case the compiler assumes the array size equal to the number of values provided inside braces { }. For example, we can write the following statement:

int A [ ] = { 1, 2, 3, 4, 5, 6 };

In the above statement, the compiler will assume that the size of the array is 6 since six data values have been given inside the braces.



CHECK YOUR PROGRESS

Q.1: What is an array?

Q.2: Give the syntax for declaring an array.

Q.3: Arrays cannot store elements of _____ data types.

Q.4: Can we declare an array without mentioning the size of an array?

9.5 ONE DIMENSIONAL ARRAY (1-D ARRAY)

One dimensional array is a collection of homogeneous data elements with only one row. It is the simplest form of array. The declaration and definition that we have learnt so far has been for a single dimensional array.

Address calculation: The elements of an array are stored in contiguous memory locations. This means that if we know the base address then we can calculate the address of the other array elements.

Let 'b' be the memory location of the first element of the array and each element requires 'w' words of memory space. Then, the address or location of element A[i] will be the summation of the base address and the product value of 'i' and 'w'.

Address of element A[i] = b + i X w

For example if we consider the figure 9.1, then the base address, b = 600. Now if we consider an integer array that requires word size 2, the address of the 10^{th} element should be 618. If we calculate the address according to the formula given above we get the same answer.

Address of element A[9] = 600 + (9 X 2) = 600 + 18 = 618

9.5.1 Entering Data Values in 1-D Array

We can insert values into an array at the time of initialization. But apart from that, we can also insert values into array elements by accessing them individually. For example, if we have an integer array A of 5 elements, we can insert the values for each of the five array positions as follows:

```
A [0] = 11;
A [1] = 22;
A [2] = 33;
A [3] = 44;
A [4] = 55;
```

In the above statements, the value for each array position has been filled individually instead of giving the values at the time of initialization and declaration.

Let us now suppose that we have an array of 50 elements. Also we can write statements like above only when we have the knowledge of the element values in advance. Otherwise when we have to take the values from the user at run time these types of statements do not provide a suitable solution.

An easier solution to the above problem is to use loop control structure statements like ***while***, ***do_while*** and ***for*** loop. Let us look at a code where with the help of for loop we can easily enter any number of values at the run time. This can be implemented as:

```
for ( i = 0; i < 5;  i++ )
{
   scanf("%d", &A[ i ]);
}
```

In the code above a 'for loop' is used to traverse from the first element of the array to the last element. The above 'scanf' statement accepts an integer value and stores it in the address of the given array location. The array location is moved from the first element position to the last element position using for loop. "&A[i]" gives the address of the 'ith' element of the array where the current data value needs to be stored.

9.5.2 Accessing Values from 1-D Array

We have learnt in the previous part about entering values in a one dimensional array. Once the values have been inserted we may need to access the array for various purposes. The procedure for accessing the values in a one dimensional array is similar to entering the values in one. This can be implemented as:

```
for ( i = 0; i < size;  i++ )
{
printf("%d", A[ i ]);
}
```

In the above code, a 'for loop' is used to traverse from the first element of the array to its last element. The 'printf' statement prints the integer value which is stored in the address of the given array location. The array location is moved from the first element position to the last element position using for loop. "**A[i]"** gives the value of the element at the **ith position** of the array.

Program 9.2: **Program to enter and access data elements in a 1-D array**

```
# include<stdio.h>
# include<conio.h>
int main()
{
   // declaring integer array of size 10
   int array[10];
   int i;
   // Entering data values in array
   printf("Enter any 10 integer values:\n");
   for(i=0;i<10;i++)
   {
      scanf("%d",&array[i]);
   }
   // Traversing and printing data values from
array
   for(i=0;i<10;i++)
   {
      printf("array[%d]     element     is
%d\n",i,array[i]);
   }
getch();
return 0;
}
```

9.6 TWO DIMENSIONAL ARRAY (2-D ARRAY)

Two- dimensional arrays are collection of homogenous data elements where the elements are ordered in number of rows and columns. A two dimensional array can be declared just as we declare a one dimensional array variable. We give the data type of the array variable followed with the name of the array variable and the size of the array in square brackets. In case of a two dimensional array also we give the data

type followed with name and size of the array but we add another value in brackets to give the second size of the array. The first size represents the row size and the second size represents the column size of the array. This is because according to the definition of a two dimensional array the array is organized in rows and columns.

Declaration of 2-D Array: We can declare a two dimensional array in the following way:

data_type name_of_array [row_size] [column_size];

In the above declaration syntax, data type can be any of the valid data types for 1-D arrays. The data type can be integer, floating point or a character. It is followed by the name of the array variable and the row size and column size of the array that are given inside different square brackets. For example:

```
int     array [ 5 ] [ 5 ];
char    arr [ 10 ] [ 3] ;
float   balance [ 2 ] [ 4 ] ;
```

When we declare an array, a contiguous memory location is allocated to that array. Let us look at an example of the physical representation of a two dimensional array of size [3][4] in computer's memory.

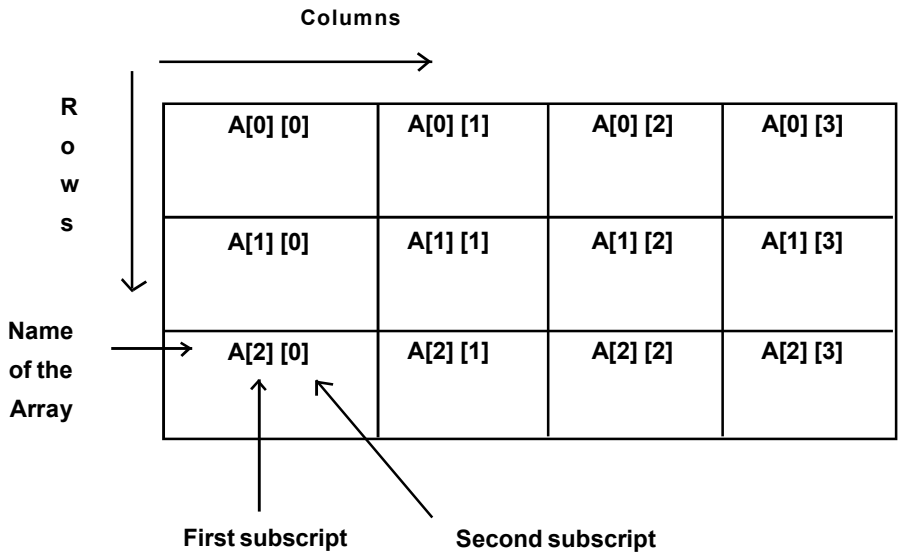


Figure 9.2: Representation of two-dimensional array A of size [3][4]

In the above figure, we can see that array A is a two dimensional matrix with three rows and four columns. Two subscripts are needed to

represent an element in a two dimensional array. The first subscript represents the row value while the second subscript represents the column value of the array. In C language, similar to single dimensional arrays, the subscripts value for both rows and column values starts from 0. If we want to represent the first element in the array we denote it by **A[0][0],** where the first subscript value 0 means that the element is located in the first row and the second subscript value 0 means that the element is located in the first column. Since we need two subscripts to uniquely represent any element in this type of array, hence it is known as two dimensional arrays.

9.6.1 Storage Representation of 2-D Arrays

Two dimensional arrays can be stored in two different ways. They can be stored in either row-major order or in column-major order.

Row-major order: In row-major order, the elements are stored on a row-by-row basis. Here, the first row is filled first followed by second row and so on. This is the common way of storing elements generally for 2-D array. Let us try to understand this concept using an example. Suppose we need to fill the following 9 element values { 1, 2, 3 4, 5, 6, 7, 8, 9 } and in an array B[3][3].

Then, we will first fill up the first row of the array using the first three element values 1, 2, and 3. Once the first row is filled, we then proceed to fill up the rest of the rows one at a time using the rest of the data values. The filled array will have the following structure:

B =

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

Column-major order: In column-major order, the elements are stored in a column-by-column basis. Here, the first column is filled first followed by second column and so on. Let us use the earlier example itself to understand this concept. We need to fill the following 9 element values {1, 2, 3 4, 5, 6, 7, 8, 9} and in an array B[3][3].

We will first fill up the first column of the array using the first three element values 1, 2, and 3. Once the first column is filled, we will then proceed to fill up the rest of the columns one at a time using the rest of the data values. The filled array will have the following structure:

B =

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

Address calculation: The address calculation for elements in two-dimensional arrays will depend on whether the elements are stored in row-major order or in column-major order. Let '**b'** be the base address or the address of the first element in the array and **'w'** be the word size. Then the address of an element **B[i] [j]** when the array has a maximum row and column size **B [m] [n]** for both the cases can be calculated using the formulas below.

- For row-major, the address of element

  B [i] [j] = b + (i × n + j) × w

  where, i represents the row value,
  j represents the column value and
  n represents the column size.

- For column-major, the address of element

  B [i] [j] = b + (j × m + i) × w

  where, i represents the row value,
  j represents the column value and
  n represents the column size.

9.6.2 Entering Data Values in 2-D Array

We can insert values into a 2-D array at the time of initialization. Insertion of data elements in a 2-D array is similar to insertion of element in 1-D arrays. Let us look at the example below.

```
int  B [3] [4] =     {    { 1, 2, 3, 4 },
                     { 5, 6, 7, 8 },
```

```
        { 9, 10, 11, 12 }
    };
```

In the above initialization, the data elements of the first row are put together inside braces, followed by data elements of second row inside second braces and so on till the last row. All these individual braces representing each row are then put inside one common brace to indicate that all rows belong to the same array.

We can also insert values into array elements by accessing them individually. For example, if we have an integer array B [2][3] of 6 elements, we can insert the values for each of the 6 data elements using their array positions as follows:

B [0] [0] = 11;
B [0] [1] = 22;
B [0] [2] = 33;
B [1] [0] = 44;
B [1] [1] = 55;
B [1] [2] = 66;

In the above statements, the value for each array position in the 2-D array has been filled individually instead of giving the values at the time of initialization.

Another easier way to insert data values into elements is to use loop control structure statements like *while*, *do while* and *for* loop. Let us look at a code where with the help of nested for loop we can easily enter the data values for the array B [2] [3] without initialization. This can be implemented as:

```
for ( i = 0; i < 2;  i++ )
{
   for ( j = 0; j < 3; j++ )
   {
      scanf("%d", &B[ i ] [ j ]);
   }
}
```

In the above code the first '*for loop'* is used to traverse through the rows and the second for loop is used to traverse through the

columns. For each iteration of the first *for loop*, the second *for loop* traverses from the first to the last column in the array. "**B[i][j]"** gives the address of the element at the "**i**th" **row** and **"j**th" **column** of the array where the current data value needs to be stored.

9.6.3 Accessing Values from 2-D Array

We have learnt the different ways of inserting values in a two dimensional array. Once the values have been inserted we may need to access the array for various purposes. The procedure for accessing the values in a two dimensional array is similar to entering the values. This can be implemented as:

```
for( i = 0; i < row_size;  i++ )
{
   for ( j=0; j< column_size; j++ )
   {
      printf("%d", B[ i ][ j ]);
   }
}
```

In the above code we have used a nested '*for loop*' to traverse the two dimensional array. The first '*for loop*' is used to traverse through the rows and the second *for loop* is used to traverse through the columns. For each iteration of the first *for loop*, the second for loop traverses from the first to the last column in the array. The 'printf' statement prints the integer value which is stored in the address of the given array location. "**B[i][j]"** gives the address of the element at the "**i**th" **row** and **"j**th" **column** of the array where the current data value needs is stored.

Program 9.2: **Program to enter and access data elements in a 2-D array**

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

```
   int A[10][10], row_size, col_size, i, j;
   clrscr();
   printf("Give the number of rows you want : ");
   scanf("%d",&row_size);
   printf("Give the number of columns you want :
");
   scanf("%d",&col_size);
   // Entering elements in 2D array
   printf("\n Enter elemnts in an array\n");
   for(i=0;i<row_size;i++)
   {
      for(j=0;j<col_size;j++)
      {
         scanf("%d",&A[i][j]);
      }
   }
   // Accessing and printing elements in 2D array
   printf("\n The elements in the array A are:\n");
   for(i=0;i<row_size;i++)
   {
      for(j=0;j<col_size;j++)
      {
         printf("A[%d][%d]  =  %d\n",i,j,A[i][j]);
      }
      printf("\n");
   }
   getch();
}
```



CHECK YOUR PROGRESS

Q.5: The base address of a char array C[15] is 210. What is the memory address of the C[14]?

Q.6: Find the memory address of D[4] [8] for an array D[10][15] and when the word size is 4 and base address is 100.



9.7 LET US SUM UP

- Array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations. In this definition, by 'finite' we mean that the array contains a fixed number of elements.
- By 'ordered' we mean that all the elements are stored in contiguous locations of the computer memory in linear way. By 'homogeneous' we mean that all the elements in the array must belong to the same data type.
- Elements in an array are referred to as using a subscript or index value.
- An array is declared in the following way:

  data_type name_of_array [size_of_array];
- One dimensional array is a collection of homogeneous data elements with only one row.
- The address or location of element A[i] will be the summation of the base address and the product value of 'i' and 'w'.

  Address of element A[i] = b + i X w
- Two-dimensional arrays are collection of homogenous data elements where the elements are ordered in number of rows and columns.
- Two dimensional array is declared in the following way:

  data_type name_of_array[row_size][column_size];
- In row-major order, the elements are stored on a row-by-row basis.
- In column-major order, the elements are stored in a column-by-column basis.



9.8 FURTHER READING

1) Kanetkar, Y. P. (2008); *Let us C*; Jones and Bartlett Publishers, Inc.
2) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
3) Thareja, R. (2015); *Introduction to C Programming*.



9.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: An array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations.

Ans. to Q. No. 2: The syntax for declaring an array is:

data_type name_of_array [size_of_array];

Ans. to Q. No. 3: Arrays cannot store elements of different data types.

Ans. to Q. No. 4: Yes, arrays can be declared without mentioning the size provided we initialize the array with the value of the elements.

Ans. to Q. No. 5: In this case, b=210, i=14 and w=1(since character data types takes a byte size of 1 for storage in C).

So, the address of C [14] = b + (i × w)
= 210 + (14×1)
= 224

Ans. to Q. No. 6: Let us assume that row-major storage representation in used in this case. Then in this case, b =100, i= 4, j=8, n= 15, w=4

So the address for D [4] [8] = b + (i × n + j)
= 100 + (4×15 + 8) × 4
= 100 + 68 × 4
= 100 + 272
= 372



9.10 MODEL QUESTIONS

Q.1: Define array.

Q.2: What is row major order and column major order of representation?

Q.3: Write a program to input a one dimensional array of 10 elements. Also write a function to print the elements on computer screen.

Q.4: How is address translation done in case of 1-D arrays?

Q.5: Write a program to input a two dimensional array **A[5][3]**. Also write a function to print the elements on computer screen.

Q.6: How is address translation done in case of 2-D arrays?

Q.7: What do you mean by multi-dimensional arrays? Can there be arrays of more than two dimensions?

Q.8: Why do we need arrays?

*** ***** ***

UNIT 10: STRINGS

UNIT STRUCTURE

10.1 Learning Objectives

10.2 Introduction

10.3 String

- 10.3.1 String Declaration and Initialization
- 10.3.2 Entering Values in String
- 10.3.3 Accessing Values in String

10.4 Array of Strings

10.5 String Handling Functions

- 10.5.1 *strlen()* Function
- 10.5.2 *strcpy()* Function
- 10.5.3 *strcmp()* Function
- 10.5.4 *strrev()* Function
- 10.5.5 *strcat()* Function

10.6 Let Us Sum Up

10.7 Further Reading

10.8 Answers to Check Your Progress

10.9 Model Questions

10.1 LEARNING OBJECTIVES

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

- define a string
- declare and initialize a string
- use string handling functions like *strlen()*, *strcmp()*, *strcpy()*
- use string handling functions like *strrev()* and *strcat()*

10.2 INTRODUCTION

In the previous unit, we have learnt about arrays and its types. In this unit we will learn about array of characters i.e., string. We will learn to define, declare and initialize string. In addition to this different string handling functions will also be discussed in this unit.

10.3 STRINGS

An array is a collection of homogeneous elements that are finite and ordered. Strings are also arrays but of character data type. Let us now discuss the concept of strings.

An array of characters is called a ***string***. In other words, strings are arrays where the data type is character. Arrays can be one dimensional or multidimensional. But strings are one-dimensional array of characters which are terminated by a **null character** represented by **'\0'**. Every string contains one or more characters that comprise the string followed by a null character '\0' that indicates the end of the string.

10.3.1 String Declaration and Initialization

Declaration of String: A string can be declared just as we declare any other array variable. We give the data type of the variable followed with the name and size of the variable. In case of a string, we give the data type as character followed with the name of the string and the size of the string. The size of the string should be the sum of the size of a number of elements that the string can contain plus the size for the null character that designates the end of the string. So, to hold the null character at the end of the string, the size of the string should be one more than the number of characters intended to enter in the string.

We can declare a string in the following way:

char name_of_string [size_of_string];

In the above declaration syntax, data type has to be a character. It is followed by the name of the string variable and the size of the string that is given inside square brackets. For example:

```
char    array [ 5 ] ;
char    name [ 10 ] ;
char    book [ 30 ] ;
```

Need for Strings: To stress the need for strings let us look at the following example:

A cricket match has been organized between two teams A and B. Suppose we do not have the knowledge of string arrays and we need to keep the name of the best player. If we use a character variable then we will be able to store only one character of the player's name.

char name1;

Let us assume that name of the player is "Rahul". To store this name do we declare five char variables to store the five characters in the name? This as shown below does not solve our problem.

char name1, name2, name3, name4, name5;

An easier way to keep the scores of the team would be to use a character array or a string. Instead of declaring many variables for keeping the name of one individual we can store the name in one string variable.

char best_player [6];

Here, we declare a character array of size six since the first five locations will store the name "Rahul" and the 6th location of the array will store the null character.

Definition of Strings: When we declare a string, a contiguous memory location is allocated to that string. Let us look at an example of the physical representation of string of size 6 which contains the string "Hello" in computers memory.

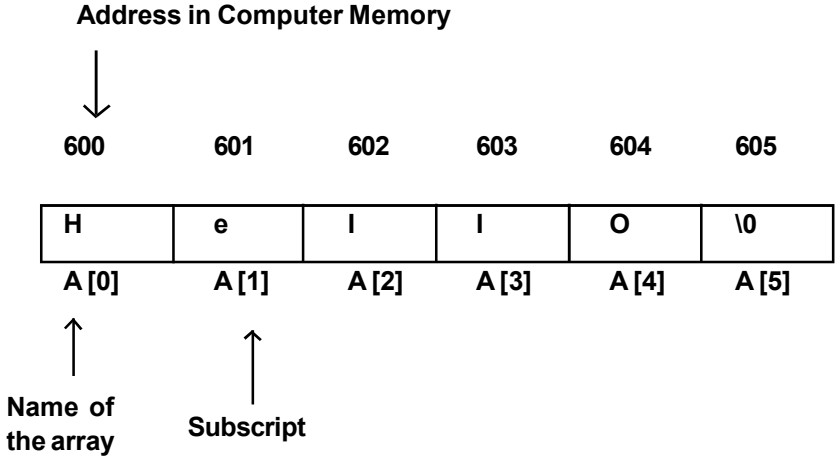


Fig. 10.1: Physical representation of string in memory

In Figure 10.1, we can see how array is actually stored in the memory. Each string element is identified with the help of the index or subscript value. The starting index value for string in C is always 0. In the above example, the first element of the string is stored in the memory address 600. For most C compilers char variable requires one byte of storage. Since strings are stored in contagious memory, so the second element will be stored next to the first element in location 601.

String Initialization: We can initialize the elements of the string when we declare the string. Initialization is generally done when we already know the character values of the string. Just like in declaration, in initialization also we give the data type of the string, followed by the name and size of the array. But in addition to these, we also provide the values of the character elements of the string within braces **{}** separated by commas where the values of the elements need to be put inside single quotes. Let us look at how initialization is done using the following example.

char A [6] = { 'H', 'e', 'l', 'l', 'o', '\0' };

In the above example, the array A is declared and initialized. The character array A has a size of 6 and the values of the elements in the string have been initialized as follows:

A [0] = H
A [1] = e
A [2] = l
A [3] = l
A [4] = o
A [5] = '\0'

The first value that is given in braces is put into the first string location with subscript value 0. So 'H' is kept at location A [0]. Similarly the second value 'e' inside braces is put into the second string location with subscript value A [1] and so on till all the given values have been put in the string.

We can also initialize the values of string without mentioning the string size in the square brackets. That is, we can leave the square brackets empty. For this case the compiler assumes the string size to be equal to the number of char values provided inside braces { }. For example, we can write the following statement:

char A [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };

In the above statement, the compiler will assume that the size of the array is 6 since six data values have been given inside the braces. There is another much simpler way to initialize the string. We can also give the values of the character elements together in double quotes **""** instead to providing them individually. For example, we can write the following statement:

char A [ ] = "Hello";

For this case the compiler assumes the string size to be equal to the number of char values provided inside braces {} plus one more for the null character. In this case, we do not need to explicitly provide the null character at the end of the string. The compiler itself will add the null character at the end of the string.

10.3.2 Entering Values in String

We can insert values into a string at the time of initialization. But apart from that, we can also insert values into strings by accessing them individually. For example, if we have a string A of 6 elements, we can insert the values for each of the six string positions as follows:

A [0] = 'H';
A [1] = 'e';
A [2] = 'l';
A [3] = 'l';
A [4] = 'o';
A [5] = '\0';

In the above statements, the value for each string position has been filled individually instead of giving the values at the time of

initialization and declaration. Let us now suppose that we have a string of 50 elements which consists in the name of a book.

A solution to the above problem is to use loop control structure statements like *while*, *do while* and *for* loop. Let us look at a code where with the help of *for* loop we can easily enter any number of character values. This can be implemented as:

```
for ( i = 0; i < 50;  i++ )
{
    scanf("%c", &A[ i ]);
}
```

In the code above, a 'for loop' is used to traverse from the first element of the string to the last element. The above 'scanf' statement accepts a character value and stores it in the address of the given string location. The location in the string is moved from the first element position to the last element position using for loop. "&A[i]" gives the address of the 'i^{th}' element of the string where the current character data value needs to be stored.

An easier solution to this is that instead of providing the data values individually we can provide them as together as a string. Let us look at the code below which has implemented this solution.

```
scanf("%s", &A);
```

The above 'scanf' statement accepts a string instead of a single character and stores it in the address of the given string location. For our case, the above statement accepts the string and stores it the string variable A.

10.3.3 Accessing Values from Strings

We have learnt in the previous part about entering values in a string. Once the values have been inserted we may need to access the string for various purposes. The procedure for accessing the values of a string is similar to entering the values. This can be implemented in two ways. In the first way we access the values of the string individually as follows:

```
for ( i = 0; i < 6;  i ++ )
{
    printf("c", A[ i ]);
}
```

In the above code a 'for loop' is used to traverse from the first element of the string to its last element. The 'printf' statement prints the character value which is stored in the address of the given string location. The array location is moved from the first element position to the last element position using " *for loop"*. "**A[i]"** gives the value of the element at the **i^{th} position** of the string. In the second way we access the values of the string together. This can be implemented as:

```
printf("%s", A);
```

The above 'printf' statement prints a string instead of a single character from the address of the given string location.

```
//Program 10.1: Program to enter and access elements in strings
# include<stdio.h>
# include<conio.h>
void main()
{
   // Program to enter and access elements in strings
   char str1[10], str2[10];
   int i;
   clrscr();
   // Entering values individually
   printf("Enter string str1:\n");
   for(i=0;i<9;i++)
   {
      scanf("%c",&str1[i]);
   }
   // Displaying entered values individually
```

```
   printf("The values entered in string1 are:\n");
   for(i=0;i<9;i++)
   {
      printf("Value in string 1 str1[%d] are
%c\n",i,str1[i]);
   }
   // Entering values together as a string
   printf("Enter string str2:\n");
   scanf("%s",&str2);
   // Displaying entered values together as a
string
   printf("The values entered in string2 are:\n");
   printf("%s\n",str2);
   getch();
}
```

10.4 ARRAY OF STRINGS

Array of strings is a two dimensional character array. Similar to two dimensional arrays, strings can also be of two or more dimensions.

We can declare a two dimensional string in the following way:

char name_of_string [row_size] [column_size] ;

In the above declaration syntax, data type of the string has to be of character type. It is followed by the name of the string variable and the row size and column size of the string are given inside different square brackets. For example:

char array [5] [5];
char student_name [3] [10] ;
char book_name [2] [4] ;

Inserting and accessing of values in a two dimensional string is similar to insertion and access of data elements in 2-D arrays. For example, suppose we need to store the name of 5 best players in the team. We can declare a string **name [5][30]** to keep the names of the persons and this can be implemented as:

char name [5] [30] ;

In the above declaration, the string variable **name** has memory to store five names where each name can have a maximum of 29 characters. A *for loop* can be used for traversing from the first row to the 5th row of the string variable to insert and access the names of the five individuals. This can be implemented as:

```
for ( i=0 ; i < 5; i++ )
{
   scanf("%s",name[i]);
}
for ( i=0 ; i < 5; i++ )
{
   printf("%s",name[i]);
}
```

```
/*Program 10.2: Program to enter and access elements in two dimensional strings */
# include<stdio.h>
# include<conio.h>
void main()
{
   // Program to enter and access elements in 2-D strings
   char name[5][30];
   int i;
   clrscr();
   // Entering values in 2D string
   printf("Enter first name of five persons:");
   for(i=0;i<5;i++)
   {
      printf("\nEnter name of %d person:",i+1);
      scanf("%s",&name[i]);
   }
   // Displaying values in 2D string
```

```
    for(i=0;i<5;i++)
    {
       printf("\nName of %d person is :",i+1);
       printf("%s",name[i]);
    }
getch();
}
```



CHECK YOUR PROGRESS

Q.1: What is a string?

Q.2: Give the syntax for declaring a string.

Q.3: Can we declare a string without mentioning the size of a string?

10.5 STRING HANDLING FUNCTIONS

String handling contains functions that are used for manipulating and performing special operations on strings. The file **"*string.h*"** is available in the C library and contains many string manipulation functions. We can use the functions in this file by including the header file "string.h" in our C program. Some of the common functions are:

- strlen() function
- strcpy() function
- strcmp() function
- strrev() function
- strcat() function

Let us look at these functions in detail in the following section:

10.5.1 *strlen()* Function

The function *strlen()* is used for finding the number of characters present in a given string. It gives back the length of the specified string variable. The syntax is as follows:

len = strlen(str1);

where 'len' is an integer variable which keeps the length of the string variable 'str1'. The following program determines the length of a string using *strlen()* function.

```
//Program 10.3: Program to find the length of a
string
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   // Program to find the length of a string
   char str1[10];
   int len;
   clrscr();
   printf("Enter string :\n");
   scanf("%s",&str1);
   len = strlen(str1);
   printf("The length of string %s is
%d",str1,len);
   getch();
}
```

10.5.2 *strcpy()* Function

The function *strcpy()* is used for coping the contents of one string to another string. It copies the contents of the source string to a destination string. The syntax is as follows:

strcpy(str1,str2);

where 'str1' is the destination string and 'str2' is the source string. The following program copies contents of one string to another using *strcpy()* function.

```
/*Program 10.4: Program to copy the contents of
one string to another string*/
```

```
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   // Program to copy contents of one strin to
another string
   char str1[20], str2[10];
   clrscr();
   printf("\nEnter string str1:");
   scanf("%s",&str1);
   printf("\nEnter string str2:");
   scanf("%s",&str2);
   printf("\nValue of string str1 before copy is
:%s",str1);
   strcpy(str1,str2);
   printf("\nValue of string str1 after copy is
:%s",str1);
   getch();
}
```

10.5.3 *strcmp()* Function

The function *strcmp()* is used for comparing the contents of two strings. It compares the contents of the strings character by character and then returns an integer as the output. The syntax is as follows:

s = strcmp(str1,str2);

where 's' is an integer and 'str1', 'str2' are the two strings whose contents are compared. The value of the s has the following meanings:

If s = 0, then str1 and str2 are equal

If s = 1, then str2 > str1

If s = -1, then str1 > str2

The following program compares the values of two strings using strcmp() function.

```
/*Program 10.5: Program to compare the contents of two strings*/
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   //Program to compare contents of one string to another string
   char str1[10], str2[10];
   int result;
   clrscr();
   printf("\nEnter string str1:");
   scanf("%s",&str1);
   printf("\nEnter string str2:");
   scanf("%s",&str2);
   result = strcmp(str1,str2);
   if(result == 0)
      printf("\nString str1 is equal to String str2 ");
   else
      printf("\nString str1 not equal String str2 ");
   getch();
}
```

10.5.4 *strrev()* Function

The function *strrev()* is used to reverse the contents of any given string. It reverses all the contents of the string except the null character which is used to indicate the end of a string. The syntax is as follows:

strrev(str1);

where 'str1' is the string whose contents are to be reversed. The following program reverses the contents of a string using *strrev()* function.

```
//Program 10.6: Program to reverse the contents
of a string
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   // Program to reverse the contents of a string
   char str1[10];
   clrscr();
   printf("\nEnter string :");
   scanf("%s",&str1);
   printf("\nOriginal string: %s",str1);
   strrev(str1);
   printf("\nAfter string reversal. String is
%s",str1);
   getch();
}
```

10.5.5 *strcat()* Function

The function *strcat()* is used to combine the contents of one string with another string. It concatenates the contents of the source string to a destination string. The syntax is as follows:

strcat(str1,str2);

where 'str1' is the destination string and 'str2' is the source string. The following program concatenates the contents of one string to another string using strcat() function.

```
Program 10.7: Program to concatenate two strings
# include<stdio.h>
```

```
# include<conio.h>
# include<string.h>
void main()
{
   // Program to concatenate two strings
   char str1[20], str2[10];
   clrscr();
   printf("\nEnter string str1:");
   scanf("%s",&str1);
   printf("\nEnter string str2:");
   scanf("%s",&str2);
   strcat(str1,str2);
   printf("Conactenated string is %s\n",str1);
   getch();
}
```



CHECK YOUR PROGRESS

Q.4: What is string handling functions?

Q.5: Can we perform the operations done on strings by string handling functions without using these library functions?



10.6 LET US SUM UP

- An array of characters is called strings. In other words, strings are arrays where the data type is character.
- Strings are one-dimensional array of characters which are terminated by a **null character** represented by **'\0'**.
- We can declare a string in the following way:
  char name_of_string [size_of_string];
- The data type has to be a character. It is followed by the name of the string variable and the size of the string that is given inside square brackets.

- Array of strings is two dimensional character arrays. Similar to two dimensional arrays, strings can also be of two or more dimensions.
- We can declare a two dimensional string in the following way:
  char name_of_string [row_size] [column_size] ;
- String handling contains functions that are used for manipulating and performing special operations on strings. The file **"string.h"** is available in the C library and contains many string manipulation functions.
- The function *strlen()* is used to find the number of characters present in a given string. It gives back the length of the specified string variable.
- The function *strcpy()* is used to copy the contents of one string to another string. It copies the contents of the source string to a destination string.
- The function *strcmp()* is used to compare the contents of two strings. It compares the contents of the strings character by character and then returns an integer as the output.
- The function *strrev()* is used to reverse the contents of any given string. It reverses all the contents of the string except the null character which is used to indicate the end of a string.
- The function *strcat()* is used to combine the contents of one string with another string. It concatenates the contents of the source string to a destination string.



10.7 FURTHER READING

1) Kanetkar, Y. P. (2008); *Let us C*; Jones and Bartlett Publishers, Inc.
2) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
3) Thareja, R. (2015); *Introduction to C Programming*.



10.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: An array of characters is called strings. In other words, strings are arrays where the data type is character.

Ans. to Q. No. 2: We can declare a string in the following way:

char name_of_string [size_of_string];

The data type has to be a character. It is followed by the name of the string variable and the size of the string that is given inside square brackets.

Ans. to Q. No. 3: Yes, we can declare a string without mentioning its size provided we initialize it with the data elements.

Ans. to Q. No. 4: String handling contains functions that are used for manipulating and performing special operations on strings. The file **"string.h"** is available in the C library and contains many string manipulation functions.

Ans. to Q. No. 5: Yes, the operations performed by string handling functions can be performed without using them.



10.9 MODEL QUESTIONS

Q.1: Define strings. How are strings represented in memory?

Q.2: Write a program to find the length of a string without using library functions.

Q.3: Write a program to copy one string to another without using library functions.

Q.4: Write a program to combine two strings without using library functions.

Q.5: Write a program to reverse a string without using library functions.

Q.6: Write a program to compare between two strings without using library functions.

Q.7: Write a program to take input a number and a string and then display the string that many numbers of times.

*** ***** ***

UNIT 11: FUNCTIONS

UNIT STRUCTURE

11.1 Learning Objectives

11.2 Introduction

11.3 Use of Functions

11.4 Function Declaration

11.5 Function Definition

11.6 Function Call

- 11.6.1 Call by Value
- 11.6.2 Call by Reference

11.7 Nesting of Functions

11.8 Function Parameters

11.9 Categories of Function

11.10 Recursive Function

11.11 Let Us Sum Up

11.12 Further Reading

11.13 Answers To Check Your Progress

11.14 Model Questions

11.1 LEARNING OBJECTIVES

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

- learn about function and its use in programs
- declare a function
- define a function
- describe function call
- learn about nesting of function
- learn about function parameters
- describe function categories
- illustrate recurrsive function

11.2 INTRODUCTION

In the earlier units, we have already used functions like *scanf( )*, *printf( )*, *clrscr( )*, *sqrt( )* etc. We have seen that C language 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 language also allows the users to define their own functions for carrying out various tasks. 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. This unit will help you in writing user-defined functions.

11.3 USE OF FUNCTIONS

A function is a set of statements that carries out some specific task in a program and it can be processed independently. Every C program consists of one or more functions. One of these functions must be called ***main***. 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.
- If the whole large 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.

11.4 FUNCTION DECLARATION



Compiler: Program that decodes instructions written in a higher order language and produces an assembly language program.

A function declaration is also known as ***function prototype.*** Function prototypes are usually written at the beginning of the program, ahead of any user defined functions including *main*. It hints to the *compiler* that the ***main()*** 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. For example:

int add(int, int);

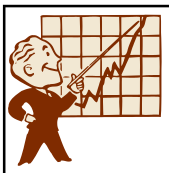
In the above prototype decleration, *int* is the data type of the item returned by the function, *add* is the name of the function and *int* within the brackets '(' and ')' are the data types of the arguments.

Program 11.1: Program to find the sum of two numbers.

Solution:

```
#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

Q.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 semicolon.

iii) The function prototype ends with a semicolon.

Q.2: Declare a function with function name "multiplication" to multiply between two floating point numbers. The function must have two fractional data type as argument which returns nothing to the calling function.

Q.3: What is the meaning of the statement *int substract(int,int);*?

11.5 FUNCTION DEFINITION

Functions can be defined anywhere in the program 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

General format of function definition is as follows:

```
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 program. 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.

Program 11.1: 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 integers is also an integer, so the return type is *int* .

Example 11.2: 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.

11.6 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. We can call a function by simply using the function name followed by a list of parameters(or arguments) 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:

Program 11.2: 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
    }
```

Parameter passing is a method for communication of data between the *calling function* and *called function*. These can be achieved by two ways:

- Call-by-value
- Call-by-reference

11.6.1 Call-by-value

In case of *call-by-value*, the compiler copies the value of an argument in a *calling function* to a corresponding parameter in the *called function* definition. The parameter in the called function is initialized with the value of the passed argument. As long as the parameter has not been declared as constant, the value of the parameter can be changed, but the changes are performed only within the scope of the *called function*; they have no effect on the value of the argument in the *calling function*.

In the following example, the *calling function* **main()** passes two values 5 and 10 to the *called function* **func()**. The function **func()** receives 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.

Program 11.3: Program to illustrate calling a function by value.

```
#include<stdio.h>
void func(int, int);
void main(void)
{
   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)
   {
   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

11.6.2 Call-by-reference

Call-by-reference refers to a method of calling a function by passing the address of an argument in the *calling function* to a corresponding parameter in the *called function*.

We have used an example below to illustrate the concept of *call-by-value* and *call-by-reference*. The aim of the two programs listed below is to perform swapping (interchange) of two values.

Program 11.4: 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("a and b before interchange: %d %d", a,
b);
   swap(a,b);    //function call
   printf("\na and b after interchange: %d %d",
a, b);
   getch();
}
void swap(int i, int j)    //function definition
{
   int t;
   t = i;
   i = j;
   j = t;
}
```

Here, the value to function *swap( )* is passed by value. When we execute this program, we will find that no swapping takes place. The values of **a** and **b** are passed to swap, and the swap function does swap them, but when the function returns to main() nothing happens.The values of **a** and **b** are still the same. The output will be:

a and b before interchange: 5 10

a and b after interchange: 5 10

In the next program we use pointers to perform call-by-reference for swapping of two values. Our next unit will help you in understanding ***pointers***.



Pointer: A pointer is a variable that holds the address of another variable.

Program 11.5: Example to illustrate calling a function 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
   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 function uses called-by-reference. In other words, address is passed by using the symbol **"&"** and the value is accessed by using the symbol **"*"**. When **swap( )** function is called, the addresses of **a** and **b** are passed to the function. 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**. So, when we interchange the values of **i** and **j** in function **swap()**, we interchange the values of **a** and **b**. Hence, when the function is complete, **a** and **b** have been interchanged. The output will be:

a and b before interchange: 5 10

a and b after interchange: 10 5



CHECK YOUR PROGRESS

Q.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.

Q.5: Write down the syntax of function definition.

Q.6: Write the first line of the function definition, including the formal argument declarations, for each of the situations described below:

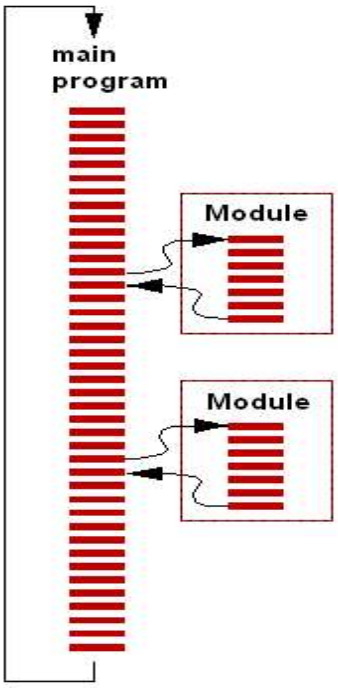
i) A function called ***average*** accepts two integer arguments and returns a floating-point result.

ii) A function called ***convert*** accepts a character and returns another character.



LET US KNOW

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called **modules** that are separately named and can be individually called. It is basically *"a divide-and-conquer"* approach to problem solving. In C, each module refers to a function that is responsible for a single task. The module performs a function and then returns control back to the program or instruction that called it.



11.7 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.

Suppose 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.

Program 11.6: Program to illustrate the concept of nested 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( )

The **main( )** function 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.

11.8 FUNCTION PARAMETERS

C functions 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.

Formal Parameters: The parameters which appear in the first line of the function definition are referred to as *formal parameter*. 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*. 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. Therefore, parameters of the same name in different functions are unrelated.

Let us consider the following example to illustrate the concept of formal and actual parameters:

Program 11.7: Program to illustrate the concept of formal and actual parameters

```
#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)**.

11.9 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

We have illustrated the above categories of functions by using different programs. The concept of different categories of functions is explained using an example to "Multiply of two integer numbers".

- **Functions with no arguments and no return values**

Program 11.8: Program to illustrate the concept of a function with no arguments and no return values

```
    #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**

Program 11.9: Program to illustrate the concept of a function with arguments and no return values

```
#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**

Program 11.10: Program to illustrate the concept of a function with arguments and one return values

```
#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**

Program 11.11: Program to illustrate the concept of a function with no arguments but a return value

```
#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**

Program 11.12: Program to illustrate the concept of a functions returning multiple values

```
#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. 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** pointer variables. Since they are declared as **int**, they can point to locations of **int** type data.

11.10 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:

Program 11.13: Program to illustrate the concept of recursive function

```
    #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

.....

.....

In the above case, we will have to 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

Program 11.14: Program to find 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.

Program 11.15: Program to 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("\n Enter a positive integer:");
        scanf("%d",&n);
        s=sum(n);
        printf("\n Sum 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



EXERCISE

1) Write a C program to find the GCD (Greatest Common Divisor) of two positive integers using recursion.



CHECK YOUR PROGRESS

Q.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.

Q.8: Write a C program to generate first n fibonacci terms using recursion.



11.11 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 is called a recursive function.



11.12 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.

2) Gottfried Byron, S. Programming with C.; Tata McGraw Hill Education.



11.13 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) True, ii) False, iii) True

Ans. to Q. No. 2: void multiplication(float, float);

Ans. to Q. No. 3: The statement **int substract(int, int);** is a function declaration where the function name is "substract" and the function has two integer type arguments and its return type is also integer.

Ans. to Q. No. 4: i) void, ii) main( ), iii) actual parameters, iv) unchanged, v) call by value, call by reference

Ans. to Q. No. 5: General format of function definition is given below:

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

Ans. to Q. No. 6: i) float average(int a, int b)

ii) char convert(char a)

Ans. to Q. No. 7: i) True, ii) False, iii) False, iv) True, v) False, vi) True

Ans. to Q. No. 1: 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
   }
}
```



11.14 MODEL QUESTIONS

Q.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( );

Q.2: State three advantages to the use of functions.

Q.3: What is meant by a function call? From what part of a program can a function be called?

Q.4: What are formal and actual arguments? What is the relationship between formal and actual argument?

Q.5: What is the purpose of *return* statement?

Q.6: Can a function be called from more than one place within a program?

Q.7: What is recursion? Explain it with example.

Q.8: Write a complete C program that will calculate the real roots of the quadratic equation $ax^2+bx+c=0$.

Q.9: Write a C program using function to find the square of an integer number without using the library function sqrt( ).

Q.10: What is a *main()* function?

*** ***** ***

UNIT 12: POINTERS

UNIT STRUCTURE

12.1 Learning Objectives

12.2 Introduction

12.3 Declaring Pointer Variable

12.4 Pointer Arithmetic

12.5 Pointers and One Dimensional Arrays

12.6 Pointers and Character Arrays

12.7 Passing Pointers to Functions as Arguments

12.8 Dynamic Memory Allocation

12.8.1 Library Function for Dynamic Memory Allocation

12.9 Let Us Sum Up

12.10 Further Reading

12.11 Answers to Check Your Progress

12.12 Model Questions

12.1 LEARNING OBJECTIVES

After going through this unit you will be able to:

- learn the basic concept of pointer
- declaring the pointer variable
- access array elements using pointer
- relate pointers to functions

12.2 INTRODUCTION

A pointer is a variable that represents the location of a data item or memory area. In other words, pointer variable holds address of other memory location rather than a value. Pointers make C and C++ more reliable and flexible. One can access the memory location directly using pointer. Within the computer memory, every data item occupies one or more contiguous memory locations. The number of required memory location depends on the data type used. For example, a character type variable occupies 1 byte (8 bits) of memory, an integer usually requires two contiguous bytes (16 bits), a floating-point number requires 4 contiguous bytes(32 bits) and so

on. Each and every memory location has a unique memory address or location number.

12.3 DECLARING POINTER VARIABLE

Declaration of pointer variable is just like normal variable declaration. Here, only the variable is followed by an *(asterisk). The general form is:

<data type> *<variable name>;

Example 1: int *p, *q;

char *name;

float *p;

Example 2:

int *p, x=10;

p=&x;

	p	x	Variable name
	2001	10	Value at address
	2001		Address of memory or variable

Here, p is a pointer variable of type integer i.e., p can store address of some other integer variable. Integer pointer can point only other integer variable. The symbol '**&**' is an **address of** operator. The statement p=&x assigns the address of variable x into p. We assume the address of x is 2001, it also may be some other value. Here is an example to look at the address of a pointer variable and value.

```
/*Program 12.1: Program to display address and value of a variable using pointer.*/
#include<stdio.h>
#include<conio.h>
void main()
{
   int *p, x=10;
   clrscr();
   p=&x;
   printf("Address of x %d\n", &x);
   printf("Value of p %d\n",p);
   printf("Value at address %d ", *p);
   getch();
}
```

12.4 POINTER ARITHMETIC

The following arithmetic operations can be performed with pointer variables in C and C++:

Subtraction	–
Incrementation	++
Decrementation	– –

Pointer arithmetic follows data type size i.e., it causes the pointer to be incremented or decremented by the number of bytes occupied by a particular data type. For example, an integer pointer variable when incremented by 1, will increase by 2, as the size of an integer variable in windows environment is 2 bytes. This could be explained with the help of the following example.

```
//Program 12.2: Program showing pointer arithmetic.
#include<stdio.h>
#include<conio.h>
void main()
{
   int *p, x=10;
   clrscr();
   p=&x;
   printf("p= %d\n", p);
   p=p+1;
   printf("p= %d\n",p);
   p=p+1;
   printf("p=%d ", p);
   getch();
}
```

Output: p= –12
p= –10
p= –8

Here, the value of p increases by 2 rather than by 1. It is because of the data type integer. One can change the data type from integer to float. In such situation p will increase by 4.

12.5 POINTERS AND ONE DIMENSIONAL ARRAYS

As discussed earlier, an array is a variable to represent multiple memory locations with the same name. Each and every element of an array has their unique memory addresses. These memory addresses can store into pointer variables. Therefore, if x is a one dimensional array, then the address of the first array element can be expressed as either &x[0] or simply as x. Moreover, the address of second array element can be written as either &x[1] or (x+1), and so on. This can be explained with one example.

Let int x[5] = {10,20,30,40,50};

Here,	0	1	2	3	4	index
x	10	20	30	40	50	Data
	2000	2002	2004	2006	2008	Address

Here, we assume that address of the first array element i.e., x[0] is 2000, so the address of second array element will be 2002, since one integer variable occupies 2 bytes in memory.

//Program 12.3: Display the content of an array using pointer

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int *p, x[5]={10,20,30,40,50};
   clrscr();
   p=&x[0];
   printf("First element=%d\t",*p);
   p=p+1;
   printf("Second element= %d\t",*p);
   p=p+1;
   printf("Third element= %d\t",*p);
   p=p+1;
   printf("Fourth element= %d\t",*p);
   p=p+1;
```

```
        printf("Fifth element= %d\t",*p);
        getch();
    }
```

The above program can also be written using loop construct, which reduces the number of statements.

```
    /*Program 12.4: Program to display the array content using pointer and loop*/
    #include<stdio.h>
    #include<conio.h>
    void main()
    {
        int *p,i, x[5]={10,20,30,40,50};
        clrscr();
        p=&x[0];
        printf("Elements are : ");
        for(i=0;i<=4;i=i+1)
        {
            printf("%d",*p);
            p=p+1;
        }
        getch();
    }
```

Here, p is a pointer variable of type integer and initially assigned the address of the first array element x[0]. In subsequent iteration or repetition the value of p is increased by 1. i.e., by 2 bytes, since p is an integer variable and displays the value of that address.

12.6 POINTERS AND CHARACTER ARRAYS

A character array forms a string. In other words, a string may be considered as a string of characters. So, string can be processed using pointer variables. Strings are always terminated with null character i.e. '\0'.

```
    //Program 12.5: Program to display characters of
```

```
a string using pointer.
     #include<stdio.h>
     #include<conio.h>
     void main()
     {
        char name[ ]=”KKHO UNIVERSITY”;
        char *p;
        clrscr();
        p=&name[0];
        while(*p!=’\0')
        {
           printf(“%d”,*p);
           p++;
        }
        getch();
     }
```

Here, the base address of the character array “name” is stored into the pointer variable p. In each iteration, value at address of p is displayed, then p is incremented by 1 until ’\0’ is found. One can increment the pointer by using “++” operator and can decrement the pointer by using “- -” operator.

```
     /*Program 12.6: Program to display characters of
a string in reverse order using pointer.*/
     #include<stdio.h>
     #include<conio.h>
     void main()
     {
        char name[]=”KKHSO UNIVERSITY”;
        char *p,*q;
        clrscr();
        p=&name[0];
        q=&name[0];
        while(*p!=’\0')
        {
```

```
            p++;
        }
            p–;
        while(p!=q)
        {
            Printf("%d\t",*p);
            p–;
        }
        getch();
    }
```

12.7 PASSING POINTERS TO FUNCTIONS AS ARGUMENTS

Pointers can pass to a function as arguments. Arguments can be passed to a function in two ways i.e. **by value** and **by reference**. When an argument is passed by value, the data item is copied to the function i.e. it makes a duplicate copy of the original variable. In such situation, any alteration or modification made in the variables of the function does not affect the value of the original variable. Some situation may arise where we want to change the value of variable in the original program with the help of a function. When calling a function using pointers, it copies the address of a variable to the function, not the value of the variable. The contents of that address can be accessed directly either within the called function or calling function. In other words, pointer variables are the direct means of communication among the function variables. Let us explain the use of a pointer variable with the help of the following example.

```
    /*Program 12.7: Program to show passing address
of a variable to a function.*/
    #include<stdio.h>
    #include<conio.h>
    void main()
    {
```

```
        void passdata(int *p);
        int x=10;
        clrscr();
        passdata(&x);
        getch();
    }
    void passdata(int *p)
    {
        printf("Address of x : %d\n",p);
        printf("Value of x : \n", *p)";
    }
```

Here, there are two functions viz. main() and passdata(). The main() function calls the passdata() function and passed the address of variable x to the function. In passdata(), p is a pointer variable to accept the address of variable x. Here, p is a formal parameter and x is actual parameter.

```
    /*Program 12.8: Program to show call by value and
call by reference*/
    #include<stdio.h>
    #include<conio.h>
    void main()
    {
        void passdata1(int *p);
        void passdata2(int p);
        int x=10;
        clrscr();
        printf("Value of x before passdata1 call %d\n",
x);
        passdata2(x);
        printf("Value of x after passdata1 call
%d\n",x|);
        passdata1(&x);
        printf("Value of x after passdata2 call %d\n",
x);
```

```
   getch();
}
   void passdata1(int *p)
{
   *p=20;
}
void passdata2(int p)
{
   p=20;
}
```

This program contains two functions, called passdata1() and passdata2(). The function passdata1() receives one pointer to integer variable as its argument and passdata2() receives one integer variable as argument. This variable originally assigned a value 10. The value is then changed to 20 within passdata2() function. The new value is not reflected within main, because the argument x was passed by value. As we have discussed earlier, when a function is called by value, then a duplicate copy of the original variable is copied to the called function. Any changes to the variables of called function are local to that function only. In passdata1(), the statement *p=20 indicates that the value 20 is assigned to the contents of the pointer address, which is address of x. Since the address is recognised in both functions i.e. passdata1() and main(), the reassigned value will be recognised within main after call to passdata1().

Note : *Function declaration, call and definition need not to be in order*.

```
/*Program 12.9: Program to exchange value of two
variable using function*/
#include<stdio.h>
#include<conio.h>
void main()
{
   void exchange(int *p, int *q);
   int x=10, y=20;
```

```
        clrscr();
        printf("Value of x & y before function call");
        printf("x=%d  y=%d\n",x,y);
        exchange(&x,  &y);  //passing address of x and
y
        printf("Value of x & y after function call");
        printf("x=%d  y=%d\n",x,y);
        getch();
    }
    void exchange(int *p, int *q)
    {
        int temp;
        temp=*p;
        *p=*q;
        *q=temp;
    }
```

12.8 DYNAMIC MEMORY ALLOCATION

Memory allocation refers to the reservation of memory for storing data. Memory allocation is done in C language in two ways (i) Static allocation and (ii) Dynamic allocation. Static allocation is done by using array. The main disadvantage of static allocation is that the programmer must know the size of the array or data while writing the program. Generally, it is not possible to know the required memory in advance. To overcome this problem dynamic memory allocation is done. Dynamic memory allocation refers to the allocation of memory during program execution. The basic difference between static and dynamic memory allocation is that in static allocation memory is allocated during the compile time, whereas in dynamic allocation memory is allocated during the program execution time.

12.8.1 Library Function for Dynamic Memory Allocation

C language provides a set of library functions for dynamic memory allocation and de-allocation. There are basically four functions used for this purpose. They are:

malloc( ), ***calloc( )***, ***realloc( )*** and ***free( )***

- **malloc():** This function is used to allocate a block of memory. After allocating the memory it returns a pointer of type *void*. This means that one can assign it to any type of pointer. The syntax for using *malloc()* is as follows:

  ptr =(cast-type *)malloc(no. of bytes);

  Here, ptr is a pointer of type cast-type. For example,

  ```
  int x;
  x=(int *)malloc(100);
  ```

  If it is unable to find the requested amount of memory, malloc() function returns NULL.

- **calloc():** It is a library function to allocate memory. The difference between *calloc()* and *malloc()* is that *calloc()* initialize the allocated memory to zero where as *malloc()* does not initialize allocated memory to zero.

  Declaration Syntax: Following is the declaration for *calloc()* function.

void (cast-type*)calloc(no. of elements to be allocated,
size of each element)

For example:

```
ptr = (int*) calloc(10, sizeof(int));
```

This statement allocates contiguous space in memory for an array of 10 elements each of size of int, i.e., 2 bytes.

```
/*Program 12.10: Program to show the usage of
   calloc() function*/
#include <stdio.h>
#include <stdlib.h>
int main()
```

```
{
   int i,no;
   int *p;
   printf("Enter number of elements :");
   scanf("%d",&no);
   p = (int*)calloc(no, sizeof(int));
   printf("Enter %d numbers:\n",no);
   for( i=0 ; i < no ; i++ )
   {
      scanf("%d",&p[i]);
   }
   printf("The numbers are: ");
   for( i=0 ; i < n ; i++ )
   {
      printf("%d ",p[i]);
   }
      free( p );
      return(0);
}
```

- **free( ):** A memory area that is dynamically allocated using either calloc() or malloc() doesn't get freed automatically when the execution terminates. You must explicitly use free() library function to release the memory space.

  Syntax: free(ptr);

  This statement frees the space allocated in the memory pointed by ptr.

- **realloc( ):** The size of dynamically allocated memory can be changed by using realloc() library function.

  Syntax: void *realloc(void *ptr, size_t size);

  Example: int *ptr = (int *)malloc(sizeof(int)*2);//dynamic allocation for two integer value

  int *ptr_new;

  ptr_new = (int *)realloc(ptr, sizeof(int)*3);// dynamic reallocation for three integer value



CHECK YOUR PROGRESS

Q.1: Choose the appropriate option:

i) Which one of the following is valid pointer declaration:

a) int a*; b) int *a, c) int *a d) int *a;

ii) What will be the output of the following statements if the variable 'a' located at address 50 [Assume no syntax error and working platform is MS Windows]

```
int *p, a=50;
p=&a
a++, p++;
a++, p++;
printf("%d%d", a,p);
```

a) 52, 54 b) 54, 52 c) 52, 52 d) 54, 54

iii) In a string '\0' is known as:

a) Back zero b) Slash zero

c) Null character d) End character

iv) The meaning of the following statements is:

void sum(int *p, int *q);

a) Function declaration

b) Call by address

c) Function does not return value

d) All of the above

v) malloc() is:

a) used to allocate memory statically

b) used to allocate memory dynamically

c) a user defined function

d) does not return value



12.9 LET US SUM UP

- In this unit we have discussed pointers and their relation to array and function.
- Pointers are variables that hold the address of other variables or memory locations.
- Pointer variables behave different way depending on the data type. For example a pointer to *char* is different from a pointer to *int*.
- There are two special operators used in pointer operation i.e. "address of" operator, **&** and "content of" or "value at address" operator ***.



12.10 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.
2) Gottfried Byron, S. Programming with C.; Tata McGraw Hill Education.



12.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) (d) int *a, ii) (a) 52, 54, iii) (c) Null character, iv) (d) None of the above, v) (b) used to allocate memory dynamically



12.12 MODEL QUESTIONS

Q.1: What is pointer? How pointer variables are declared?

Q.2: Explain the mechanism to access one dimensional array using pointer.

Q.3: What is pointer arithmetic? What are the operators used in pointer arithmetic.

Q.4: Write a program to input your name and display it in reverse order using pointer.

Q.5: Write a program to input your name and display it using pointer

Q.6: Explain how pointers are passed to a function.

Q.7: Write a program to input a string and count the number of vowels

Q.8: What is dynamic memory allocation? What are the functions used to allocate memory dynamically.

Q.9: What are the difference between 'pass by value' and 'pass by reference'.

*** ***** ***

UNIT 13: STRUCTURE AND UNION

UNIT STRUCTURE

13.1 Learning Objectives

13.2 Introduction

13.3 Structure

- 13.3.1 Structure Declaration
- 13.3.2 Initialization of Structure
- 13.3.3 Accessing the Members of a Structure

13.4 Array of Structures

13.5 Structure within a Structure

13.6 Passing Structures to Functions

13.7 Pointer to Structure

13.8 Union

13.9 Enumerated Data Types

13.10 Defining Your Own Types (Typedef)

13.11 Let Us Sum Up

13.12 Further Reading

13.13 Answers To Check Your Progress

13.14 Model Questions

13.1 LEARNING OBJECTIVES

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

- write program using a structure rather than several arrays
- 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 types
- learn about typedef

13.2 INTRODUCTION

We have already been acquainted with array which is a linear data structure. Array takes basic data types like *int*, *char, float* 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*.

In this unit we will learn about structure and union. Here we will see how a structure and union are defined, declared and accessed in C programming language.

13.3 STRUCTURE

A structure is similar to records. It stores related information about an entity. With the use of structures, programmers can conveniently handle a group of related data items of different data types.

In C language, structure is basically a user defined heterogeneous data type. The main difference between a structure and an array is that an array contains related information of the same data type.

13.3.1 Structure Declaration

A structure is declared using the keyword ***struct*** followed by a structure name. All the variables of the structure are declared within the structure. The data types of all these variables within a structure can be of different types. A structure is generally declared with the following syntax:

```
struct struct_name
{
    data_type variable_name;
    data_type variable_name;
    ....................................
```

.....

};

For example, to keep the details of a book, we have to declare a structure for the book containing variables like title, author, pages, price, publisher etc. This book structure can be declared as:

```
struct book
{
    char title[20];
    char author[15];
    char publisher[25];
    int pages;
    float price;
};
```

In the above declaration, the book name (i.e., title), author name and publisher name would have to be stored as **string**, and the page and price could be **int** and **float** respectively. The keyword *struct* declares a structure to hold the details of five fields namely title, author, publisher, pages and price. These are members of the structures. Each member may belong to different or same data type. It is not always necessary to define the structure within the *main()* function.

Structure declaration acts as a template which conveys structure information and member names to the compiler. Structure is a user-defined data type. Now, let us discuss how to declare structure variables.

We can declare structure variables using the structure name (tag name) any where in the program. For example, the statement,

struct book book1, book2, book3;

declares *book1, book2, book3* as variables of type *struct book*. Each declaration has five elements of the structure *book*. The complete structure declaration might look like this:

```
struct book
{
    char title[20], author[15], publisher[25];
    int pages;
    float price;
};
struct book
{
    char title[20], author[15], publisher[25];
    int pages;
    float price;
}   book1,book2,book3;
```

The use of tag name is optional. In the declaration, ***book1***, ***book2***, ***book3*** are structure variables representing three books. Tag_name is not included in this declation. A structure is usually defined before ***main()***. In such cases, the structure assumes global status and all the functions can access the structure.

Again, let us consider another structure declaration "employee".

```
struct employee
{
    char fname[15];
    char lname[15];
    int id_no;
    int month;
    int day;
    int year;
}   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 month, 2 bytes for day, 2 bytes for year.

13.3.2 Initialization of Structures

Initialization of structure means assigning some constants to the members of the structure. A structure can be initialised in the same way as other data types are initialized. The general syntax to initialize a structure variable is as follows:

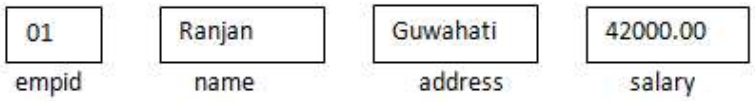
```
struct struct_name
{
    data_type member_name1;
    data_type member_name1;
    data_type member_name1;
    .....
    .....
}   struct_var = {constant1, constant2, ....};
    or,
    struct struct_name
{
    data_type member_name1;
    data_type member_name1;
    data_type member_name1;
    .....
    .....
};  struct struct_name struct_var = {constant1, constant2, ....};
```

For example, let us initialize an employee structure as follows:

```
struct employee
{
    int empid;
    char name[20];
    char address[30];
    float salary;
}   emp1 = {01,”Ranjan”,”Guwahati”, 42000.00};
    or by writing
    struct employee emp1 = {01,”Ranjan”,”Guwahati”, 42000.00};
```

C language automatically initializes the structure members if the user does not explicitly initialize all the members. This is known as *partial initialization*. ***Integer*** and ***float*** members are initialized to ***zero*** and ***character arrays*** are initialized to '**\0**' (null value) by default. Pictorially we can represent it as follows:

struct employee emp1 = {01,"Ranjan","Guwahati", 42000.00};



struct employee emp2 = {02,"Kaveri"};

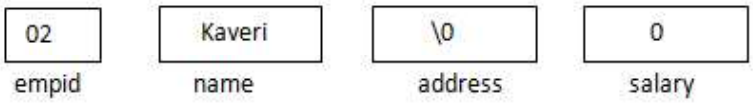


Fig. 13.1: Assigning valuesto a structure element

13.3.3 Accessing the Members of a Structure

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***. A structure member variable is generally accessed using the '.' dot operator. The syntax is:

struct_var . member_name;

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.

To assign value to the individual data members of the structure variable *book1*, we may write,

book1.price = 520.00;
book1.name = "Java";
book1.author="Kumar";

We can use *scanf()* function to input values for data members of the structure variable *book1* like this :

```
scanf("%f",&book1.price);
scanf("%d",&book1.pages);
```

For displaying the values of structure variable book1, we can use *printf()* function like this:

```
printf("%f", & book1.price);
printf("%s", &book1.author);
```

```
/*Program 13.1: Program to enter information of one student and to display that 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();
    }
```



EXERCISE

1) Modify the previous program(*Program 13.1*) for storing and displaying information of 2 students.

13.4 ARRAY OF STRUCTURES

In programming we may often need to handle many records. For example, in a class, there may be many numbers of students, say 50 students. So, to keep the records of 50 students we need an array of structures. This can be written as

```
struct student
    {
        int rollno;
        char name[20];
        char course[15];
        float fees;
    };
struct student s[50];      //s is an array of structure
```

Here, s is an array of structure of 50 students, each of which is of type *struct student*.

```
/*Program 13.2: Program for storing records of 50
students and displaying those records*/
    #include<stdio.h>
    #include<conio.h>
    void main()
    {  struct student
       {
```

```
        int roll;
        char name[20];
        char address[30];
        int age;
    };
        struct student s[50];
        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].rollno);
        printf("\nEnter the name of the student:");
        scanf("%s",&s[i].name);
        printf("\nEnter the course of the student:");
        scanf("%s",&s[i].course);
        printf("\nEnter the dues of the student:");
        scanf("%f",&s[i].fees);
    }
        printf("\n\nInformation of all students:");
        for(i=1;i<=n;i++)
    {
        printf("\nRoll no.:%d",s[i].rollno);
        printf("\nName:%s",s[i].name);
        printf("\nCourse:%s",s[i].course);
        printf("\nSchool dues:%f\n\n",s[i].fees);
    }
    getch();
}
```



CHECK YOUR PROGRESS

Q.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*.

Q.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

Q.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.

13.5 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;
    struct date dob;        //structure within structure
}   student1,student2;
```

the structure ***student*** contains another structure ***date*** as one of its members.

13.6 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.

Passing individual members of structure to a function: To pass any individual member of the structure to a function as argument, we have to use the dot operator to refer to the particular member of the structure.

For example, a program to display the contents of a structure passing the individual elements to a function is shown below:

Program 13.3: Program to illustrate passing individual structure elements to a function.

```
#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 function*/
   display(e1.emp_id,e1.salary);  //function call
   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, passing individual elements would become more tedious as the number of structure elements increases. A better way would be to pass the entire structure variable at a time.

Passing entire structure to a function: There may be numerous structure members (elements) in a structure. Passing these individual elements as argument to a function would be a tedious task. Just like any other variable, we can pass an entire structure as a function argument. A

structure is passed as an argument using the call by value method. This means a copy of each member of the structure is made. This method is very inefficient, especially when the structure is very big or the function is called frequently. Use of pointers in such sitiation is more suitable.

In the following program, we are passing a whole structure to a function.

```
     //Program 13.4: Program to illustrate passing a
whole structure to a function.
     #include<stdio.h>
     #include<conio.h>
     struct employee
     {
        int emp_id;
        char name[25];
        char department[10];
        float salary;
     };
     static struct employee
e1={10,"Palash","Sales",26000.00};
     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: 10 Palash Sales 26000.00

```
//Program 13.5: Program to illustrate 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 item
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

13.7 POINTER TO STRUCTURE

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 program shows the relationship between a structure and a pointer.

```
//Program 13.6: 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

Q.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.

Q.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 _____.

13.8 UNION

In some situations we may wish to store information about a person. 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 the information for a person.

C provides a data structure which fits our needs for the above scenario, 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 data type 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 data type. 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 data structure that can hold different types of data.

```
      //Program 13.7: Program demonstrating initializing
union members 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 integer values are displayed wrongly as the union only holds one value for one data type.

13.9 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 identifier that names

the type being defined and ***enumeration 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 13.8: Program to illustrate the
concept of enumerated data type
#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.

13.10 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, we can use the newly defined type as:

ulong distance;

We can use the typedef keyword to define a structure as folllows:

```
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

Q.6: Mentions the features of union data type.

Q.7: Mention the advantages of structure type over the union type.



13.11 LET US SUM UP

- Structure is basically a user-defined data type that can store related information together.
- The main difference between a structure and an array is that, an array contains related information of the same data type.
- A structure is declared using the keyword ***struct*** followed by a structure name. Elements of the structure are declared within the { and } brackets with their respective data types. For example, the following is a structure declaration for storing customer information.

```
struct customer
{
    int cid;
    char name[20];
    char address[30];
    long int mobile_no;
};
```

- Memory is allocated only when we declare variables of the structure. In other words, memory is allocated only when we instantiate the structure. For example, for the above structure customer, memory will be reserved when we declare structure variable.

```
struct customer c;
```

- C permits the use of arrays as structure members.
- A member of the structure cannot be accessed directly using its name. We must use the structure name followed by the dot '.' operator before specifying the member name.
- In case of partial inialization, first few members of the structure are initialized and the uninitialized members are assigned default values.
- A structure can be placed within another structure.
- 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.



13.12 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.
2) Gottfried Byron, S. Programming with C.; Tata McGraw Hill Education.



13.13 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1:
```
struct complex
{
    float real, imaginary;
};
struct complex c1,c2,c3;
```

Ans. to Q. No. 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.)

Ans. to Q. No. 3: i) True, ii) True, iii) False, iv) False, v) False, vi) False

Ans. to Q. No. 4: i) True, ii) True, iii) False, iv) True, v) True

Ans. to Q. No. 5: i) Pointers, ii) tag name

Ans. to Q. No. 6: The main characteristics of Union data type are:

a) The size of union is equal to the size of number of bytes occupied by the largest data member in it.

b) only one data member in union is active at a time.

Ans. to Q. No. 7: The structure data type can hold many data related to an entity like person, book, student etc., but a union type holds only one data active at a time.



13.14 MODEL QUESTIONS

Q.1: What is a structure? How is a structure different from an array?

Q.2: How is structure declared? Define a structure to represent a date.

Q.3: What is meant by array of structure?

Q.4: How are the data elements of a structure accessed and processed?

Q.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.

Q.6: What is meant by union? Differentiate between structure and union.

Q.7: What is the purpose of typedef feature? How is this feature used with structure?

Q.8: Write short notes on:

a) Enumerated data type

b) Type definition.

Q.9: Write a C program to read and display the information of all the students in a class.

*** ***** ***

UNIT 14: FILE HANDLING

UNIT STRUCTURE

14.1 Learning Objectives

14.2 Introduction

14.3 File System

14.4 Opening and Closing File

14.5 Input/Output Operations on Files

14.5.1 getc() and putc()

14.5.2 fputs() and fgets()

14.5.3 fprintf() and fscanf()

14.6 Seeking Forward and Backward

14.7 Let Us Sum Up

14.8 Further Reading

14.9 Answers To Check Your Progress

14.10 Model Questions

14.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- learn about various file operations
- read data from a file
- write data to a file
- describe various input/output functions for file operations

14.2 INTRODUCTION

In the previous units, we have 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 have to be read and written and the console oriented I/O function cannot handle it efficiently.

This is because the entire data is lost when the program is terminated or the computer is terminated or the computer is turned off. Therefore, it is necessary to have a more flexible approach where large amount of data can be stored permanently in disks and we can read them whenever required. The concept of file helps us to store and handle this type of data easily. In this unit we are going to discuss handling of files in C.

14.3 FILE SYSTEM

A ***file*** is a storage place 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 like:

- 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 operations 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.

The 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 going to discuss high level I/O functions in this unit. Let us take a brief overview of Disk I/O functions using the figure given below.

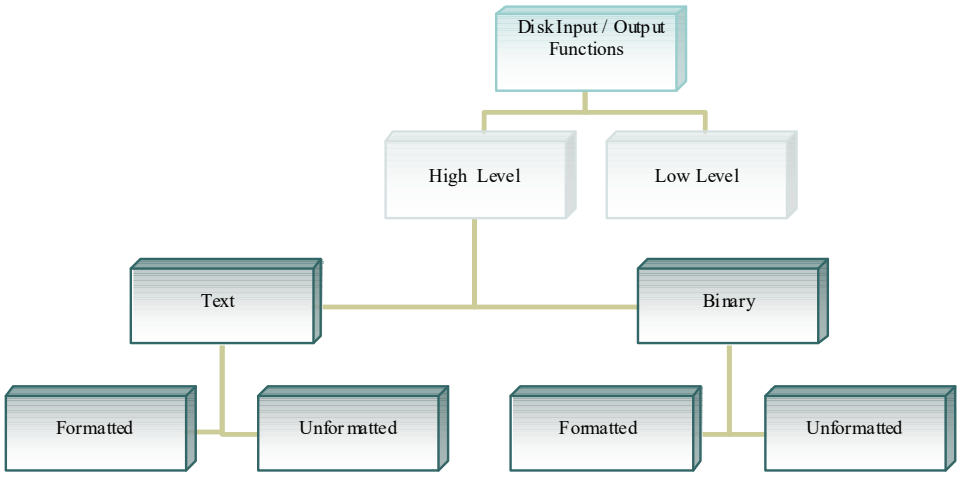


Fig 14.1: Disk I/O Functions

The High level input/output functions includes the following functions which will be discussed broadly in the next section.

i) **fopen():** fopen() function is used for opening an existing file and also used to create a new file.

ii) **fclose():** fclose() function is used to close a file which has been already opened.

iii) **getc():** getc() function is used to read single characters from a file.

iv) **putc():** putc()function is used to write single characters to the file.

v) **fputs():** fputs() function is used to writes strings to the file.

vi) **fgets():** fgets() function is used to read strings from file

vii) **fprintf():** fprintf() function is used to write a set of values to a file.

viii) **fscanf():** fscanf() funcyion is used to read a set of values from a file.

Files are used to store data in the secondary memory permenantly. Let us look at some of the basic file information associated with files like:

a) naming the file (i.e., file name)

b) data structure that links the file. and

c) purpose of opening the file.

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 examples of file names are:

i) add.c (Here **add** is primary part and **.c** is the extension)

ii) kkshou.c

iii) output.txt

b) The data structure of a file is defined as **FILE** where **FILE** is a kind of data type.

c) The purpose of opening a file can be to perform read or write operation on the files.

Note: All the files should be declared as type **FILE** before they are used.

14.4 OPENING AND CLOSING FILE

Opening a File: The pupose of opening a file was mentioned in the previous section. In addition to reading and writing a new file, we can also add data to a already existing file. Let us now try to understand how we can open and close any given file. The general syntax for opening a file is illustrated in the following program segment:

```
FILE *fptr;
fptr=fopen("filename","mode");
```

In the above program segment, 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 one of the specific purposes as listed below.

Mode	Meaning
"r"	Opens an existing file for reading purposes
"w"	Opens 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"	Opens an existing file for adding new data at the end of a file. New file is created if there is no existing file with that filename.

"r+"	Opens a file for both reading and writing purposes.
"w+"	Open a file for both reading and writing. If file name already exists, it will be destroyed and a new 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 above, the return value is assigned to **fptr**. If the specified file could not be opened successfully, then it returns **NULL**; sometimes fopen() may fail in opening a file with a specified mode because of some unwanted reason. This can be easily checked by comparing value of **fptr** with **NULL**. 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 ");
```



EOF: EOF is a constant defined in the header file stdio.h.

In the above program segment, if ***"data.txt"*** does not exist, then the value of ***fptr*** will be NULL and the output will be:

Error in opening the file

Closing File: It is always a good programming practice to close the file using fclose() as soon as all operations on the file have been completed. This 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().

The general syntax of fclose() is:

```
int fclose(file_pointer)
```

Here, fclose() function closes the file and returns zero on success or **EOF** if there is an error in closing the file. EOF stands for end of file.

14.5 INPUT/OUTPUT OPERATION ON FILES

The function **fopen()** can be used for opening a file for reading or writting purposes. To read data from a file, we need a mechanisim to read the contents of the file.

14.5.1 getc() and putc()

The function **getc()** is used to read the contents 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, increments 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**.

A question that may arise in our mind now is that while reading from the file continously, how can we determine whether that the file has been completly read or if we have reached the end of the file. Generally, we 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 reached. So we can check if we have reached the end of the file 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 using *fopen()* before writing data into the file. For example,

putc(ch,fptr)

In the above statement, **ch** is a character to be written to the file referrenced by the file pointer **fptr**.

Program 14.1: Write a program to read the contents of a already existing file which

```
contains some data. Display the data in the
file.
#include<stdio.h>
#inlcude<conio.h>
void main()
{
   FILE *fptr ;          // File pointer
declaration
   char ch;
   fptr=fopen("kkhsou.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();
}
```

In this example, we have considered a already existing file named "kkhsou.org". We want to read and display the contents of this file. 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 display the contents simultaneously to the monitor. We need to close that file using *fclose()*.

Program 14.2: Write a program to read your name, roll number from the keyboard and then write it to a file named "address.txt ".

```
#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]

In the above program, we are asked to read our name, roll number directly from the keyboard and then to write it to the file

named "address.txt". We can read the name and roll number using *scanf()* or *getchar()* functions or can use other input functions.Once we finish reading we need to open the file *"address.txt"* in write mode to write the data to the file.

In the output, we have entered "Nayan 4", which is then written to the file *"address.txt"* and we can open the file directly to view the output. In C language '**\n**' is used to denote a new line character. In the above program when we encounter '\n' in gecthar() then we stop the loop, thinking that the user has finished entering his data. However, there is another way of reading more than one line.

Program 14.3: Write a program to copy a file named "first.txt " to another file named "second.txt". Assume that file *first.txt* already exists.

```
#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();
}
```

In this example, since the file *first.txt* aready exists and we simply need to write a program that will copy the content of *"first.txt"* to *"second.txt".* We need to open the *"first.txt"* file in *read* mode and then *second.txt* file in *write* mode.

14.5.2 fputs() and fgets()

In the previous section, we have discussed how to read from the files and also how to write to the file. But we have been 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()*. Functions like *gets()* and *fputs()* are related to only keyboards input/ output, whereas *fgets()*,*fputs()* are related to files. The ***fputs()*** **f**unction writes a string to a file. For example, say if **fptr** is a file pointer which opens a file for writting, then :

fputs("Welcome to KKHSOU", fptr) ;

The sentence "Welcome to KKHSOU" is written to a file pointed by fptr. We can also 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. The general way of representing fgets() is:

fgets(strings,no_bytes,file_pointer);

The above statement reads a strings having (no_bytes-1) from a file pointed by file_pointer. "no_bytes" indicates how many bytes of strings are to be selected. We give -1 here because the file

always read from starting 0 for the first character.

Program 14.4: Write a program to read a line from the keyboard and then write it to a file named *"output.txt".*

```
#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)

In this example, we are asked to read a few lines using keyboard, so we will need a character array to store the lines read. Then we have to open a file *"output.txt"* in write mode and write the lines (character array) to it. The text *"Hello learners"* in this example is given by the user from the keyboard is written to the file output.txt.

Program 14.5: Write a program to read from a file then display it to the monitor screen.

```
#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 displayed it to the output screen i.e., monitor. Firstly, we open an existing file *"output.txt"* assuming that the file exists with some content. Then we use **fgets()** function to read the file. We have used a character array of size 79 in fgets(), since there can be a maximum of 80 characters in a line of a file. After reading from the file we store the variable **strngs** array using fgets() and then we display it to the output screen using puts().



CHECK YOUR PROGRESS

Q.1: State whether the following statements are True or False:

a) fopen() is used to open and close a file.

b) File concept is used mainly to store data permanently and to use them later.

c) fopen("file.c","w") opens file.c for reading purposes only.

Q.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.

14.5.3 fprintf() and fscanf()

We have till now discussed how to read or write a single character from a file using getc(), putc(). We have also discussed about reading and writing the strings using fgets() and fputs() to the files. All these functions are related to a particular data type i.e. characters. For writing and reading data to the files having different data type variables, we can use **fprintf()** and **fscanf()** respectively. These functions are mostly same as printf() and scanf() function that we have used in earlier programs except that fprintf() and fscanf() are used for writing and reading from the **files**. The general form of fprintf() is:

fprintf(fptr,"control string",list);

where fptr is a file pointer associated with the file opened for writing. The control string contains output specification for the items in the list. The list may include variables, constants and strings. We will get a clear idea of control string when we explain the examples. 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 a float variable.These different data type variables are 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);

where name, roll and per are read from the file specified by **fptr**.

Program 14.6: Write a program to read name,roll number from the keyboard for a student and then write it to a file using fprintf(). Then

```
use fscanf() to read the from the file written
by the code in the above program segment.
#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);
}
```

In the above program the name and roll number given by the user is written to the file **"out.txt"**.

In the next program, we open the file in read mode and then read the data using fscanf(). After that we may display it using printf() function in order to view the result.

```
#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);
}
```

14.6 SEEKING FORWARD AND BACKWARD

We have till now, discussed about reading/writing files only from the starting position. But sometimes we may encounter situations where we have to read or write files at random positions. Some basic functons that can be used for reading or writing files at random positions are explained briefly below:

a) ftell(): The **ftell()** function gives the current positions of a particular file. The general syntax for **ftell()** function is given below.

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 reset position of the file pointer to the start of the file. For example,

rewind(fptr);

n=ftell(fp)

Here, 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 **fseek()** 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 of the following:

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 is to be moved forwards or negative meaning that it is to be moved backwards. Let us try to understand the meanings using the statements given below.

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.

The function **fseek()** returns zero after successful operations and-1 otherwise.



CHECK YOUR PROGRESS

Q.3: Give the syntax of ftell().

Q.4: What file functions can be used to read/write different data type variables from file?



14.7 LET US SUM UP

- File concept is generally used to store data permanantly and to use it later. Opening, Reading, Writing Closing etc. are some of the operations that can be performed on files.
- 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() function returns NULL.
- putc(), fputs() and fprintf() are used mainly for reading data from a file pointed by a file pointer. Similarly, getc(), fscanf(), fgets() are used for reading information from the file.



14.8 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.

2) Gottfried Byron, S. *Programming with C*; Tata McGraw Hill Education.



14.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) False, b) True, c) False.

Ans. to Q. No. 2: a) write a character, b) reading, c) -1

Ans. to Q. No. 3: The general syntax for **ftell()** function is:

n = ftell (fptr),

where, the return value of **n** is in bytes which indicates that **n** bytes have already been read or written.

Ans. to Q. No. 4: **fprintf()** and **fscanf()**



14.10 MODEL QUESTIONS

Q.1: Explain the functions of fopen() and fclose() with one example each.

Q.2: Write the name of two file functions for input operations.

Q.3: Give the main difference between getc() and fgets() functions.

Q.4: Mention the various modes used with the fopen() functions.

Q.5: Explain briefly: ftell(), rewind(), fseek().

Q.6: Write C programs to perform the following:

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.

*** ***** ***

UNIT 15: PREPROCESSOR DIRECTIVES

UNIT STRUCTURE

15.1 Learning Objectives
15.2 Introduction
15.3 Preprocessor
15.4 Macro Substitution
15.5 Let Us Sum Up
15.6 Further Reading
15.7 Answers to Check Your Progress
15.8 Model Questions

15.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- define preprocessor directives
- learn about macro substitution
- define macro, parameterised macro etc.

15.2 INTRODUCTION

In this unit we will learn about preprocessor directives available in C language. We will also learn to define macros and invoke them from C program. The preprocessor is a program that processes the source code before it passes through the compiler. It operates under the control of the preprocessor directive which is placed in the source program before the *main( )* function. A C preprocessor instruct compiler to do required pre-processing before actual compilation.

Besides macros, we will also learn about file inclusion and conditional directives in this unit.

15.3 PREPROCESSOR

A unique feature of C language is the preprocessor. The prepocessor is a collection of special statements, called *directives,* which are executed

at the begining of program compilation. Before the source code is passed through the compiler, it is examined by the preprocessor for any preprocessor directives. In case the program has some preprocessor directives, appropriate actions are taken depending on the directives and the source program is handed over to the compiler. The general rules for defining a preprocessor are as follows:

a) Preprocessor directives are always preceded by a hash sign (#).
b) They must start in the first column.
c) Preprocessor directive should not be terminated by semicolon (;).
d) There should be only one preprocessor directive in one line.

The advantage of using preprocessor directives in a C program are as follows:

- The program becomes readable and easy to understand.
- The program becomes portable.
- The program can be easily modified.
- As a result, the program becomes more efficient to use.

Some examples of preprocessor directives in C language 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	Tests a compile time condition
#else	Specifies alternatives when #if test fails

a) #define: C language allows defining an identifier having constant value using #define directive. The #define statement is also called *macro definition* or simply a *macro*. 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 general syntax for defining a macro is as follows:

```
#define identifier string
```

The preprocessor replaces every occurence of the identifier in the source code by the string. For example, few macro directives are:

```
#define PI
#define ROWS
#define COLS
```

Program 15.1 shows the use of macro directive. In the program, the statement `#define PI  3.14` defines a macro PI as an abbreviation for the token 3.14.

/*Program 15.1: Program to illustrate the use of #define directive*/

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

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

b) #include (File Directive): The *#include* directive causes one file to be included in another file. With *#include* directive, an external file containing functions, variables, or macro definitions can be included as a part of our program. This avoids the effort to rewrite the code that is already written. The *#include* directive is used to inform the preprocessor to treat the contents of a specified file as if those contents had already appeared in the source program at the point where directive appears. The general format is :

```
#include <filename>
```

When we include a file using angular brackets, a search for thefile named "filename" in a standard list of system directories.

For example, *#include<stdio.h>,* which appears at the start of a program. This #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 *stdio.h* is essential for the proper functioning of the library functions such as *getchar()*, *printf()*, *scanf()* and *putchar()* etc.

c) #undef : Sometimes it is essential to undefine a macro that is already defined by #define. This can be accomplished by #undef directive. The #undef removes a macro definition from the macro symbol table. Once a macro name is undefined, the name the macro ceases to exist from the point of undefinitionand the preprocessor directive behaves as if it had never been a macro name. Undefining a macro means to cancel its definition.

In the following example the #undef directives removes whatever is defined with #define.

```
#define MAX 100
.
.
.
#undef MAX
```

d) #Conditional Directive: The preprocessor conditional compilation command allow 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, #endif, #else, #elif** etc.

- The **#if** directive is used to test whether an expression evaluates to a nonzero value or not. While using #if directive in a program, we should make sure that each #if directive must be matched by a closing #rndif directive. Let us look at the statements below:

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

- The **#elif** directive is used when there are more than two possible alternatives. The **#elif** directive is embedded within the #if directive and has the following syntax:

```
#if condition
       Statement1;
 #elif new_condition
        Statement2;
 #else
        Statement2;
 #endif
```

- The **#else** directive can be used within the controlled text of a #if directive to provide alternative text to be used if the condition is false.

```
#if condition
       Statement1;
 #else
       Statement2;
 #endif
```

- The **#endif** directive ends the scope of the #if, #else, or #elif directives. The number of #endif directives required depends on whether the #elif or #else directive is used.

15.4 MACRO SUBSTITUTION

Constant values or expressions can be identified using symbolic names. At the time of preprocessing, all these symbolic names are replaced by values/ expressions. This process is referred to macro substitution. The general form for a simple macro definition is:

```
#define  macro-name  value or string
```

It should be remembered that there is no white space(blank) between **#** and **define**, but single white space are used to separate the identifier i.e., **macro-name** and the **value** or **string**.

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 name that are within comments or string constants. It is not appropriate to end a macro with a semicolon because the macros are used within expressions in the body of the program. Macros that represent single numeric, string or character values can also be referred to constants. Some examples of simple macro definitions are given below:

```
#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 */
#define  N 50
#define  TITLE "ABC & Co"
```

In the last two example, the constant value 50 is identified by N and “ABC & Co” is identified by TITLE which are actually considered as strings. At the time of preprocessing, the value 50 and “ABC & Co” are substituted in the place N and Title. The identifier N and Title are considered as macro.

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

```
#define macro-name(p1, p2,....)body-of-macro
```

Here, p1,p2, ..etc. are parameters. 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 )
```

Let us consider a simple C program to illustrate the use of parameterized macro.

```
/*Program 15.2: Program to illustrate the use of macro
    #include<stdio.h>
    #define SQR(x)  x*x
    void main()
    {
       int result;
       int a=5, b=6;
       result = SQR(4);
       result = SQR(a+b);
       printf("Result: %d",&result);
    }
```

Output: 41

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

result = a+b * a+b;

which is evaluated as **a + (b * a) + b** which will not give the expected answer. The expected answer was **121** while output we recieved was **41**.

The problem, ofcourse, is with the evaluation of the operators involved in with the expanded macro. The multiplication operator is evaluated before the addition operator which makes the result 41. Even by adding parentheses as shown below the same result is produced.

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

This is because when a+b is passed it expands to **(a + (b*a) + b)** which again does not give us the desired result. Therefore, more parenthesies are required to get the desired result.

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

The above macro definition will finally give the desired result to the macro expansion.



CHECK YOUR PROGRESS

Q.1: Write down the rules for defining a preprocessor.

Q.2: Give the syntax for defining macro.



15.5 LET US SUM UP

- Preprocessor is a program that processes the source code before it passes through the compiler.
- A programmer can use preprocessor to make the program easy to read, portable and more efficient.
- The preprocessor directives are always preceded by a hash sign (#).
- To define processor macros, we use *#define* directive.
- A macro is a simple identifier which will be replaced by a code fragment.
- The *#include* directive causes one file to be included in another file.
- The *#include* directive is used to inform the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.

- The *#undef* directive undefines or removes a macro name previously creadted with *#define*.
- The *#if* directive is used to control the compilation of a source file.



15.6 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
2) Gottfried Byron, S; *Programming with C*; Tata McGraw-Hill Education.



15.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: The 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 in one line.

Ans. to Q. No. 2: The general form for a simple macro definition is:

#define macro-name value



15.8 MODEL QUESTIONS

Q.1: Define a macro. Summarize the similarities and differences between macros and functions.

Q.2: Give the syntax and use of preprocessor directives *#include* and *#define*.

Q.3: What is the scope of a preprocessor directive within a program file?

Q.4: Can we have a C program that does not use any preprocessor directive?

Q.5: Why should we incorporate preprocessor directives in our programs? Give at least one example to support your answer.

Q.6: Explain the importance of the #define preprocessor directive.

Q.7: How can #include directive be used in your program?

Q.8: Can the #undef directive be applied to a macro name that has not been previously defined? If yes, why?

*** ***** ***