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 Answers to Check Your Progress

1.8 Further Reading

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

This the introductory unit of this course. In this unit, you will learn essential programming concepts to design and develop accurate and efficient programs. We will cover how to write algorithms and pseudocode,

as well as how to use flowcharts to visually represent logical flows. These tools will help both programmers and users clearly understand problem solutions.

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 to 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 more 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 $\rightarrow$ 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  print A
         else if A < B
                then print B
         else
                print "The numbers are equal"
Step 4:  End
```

***Example 1.4:* Write the algorithm to convert the temperature on ^oF (Fahrenheit) to ^oC(Centigrade) using the formula $^oC = 5/9 (^oF – 32)$.**

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 $\rightarrow$ 5/9*(F - 32)

Step 3: Print 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: Print 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 statements 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 statements.



CHECK YOUR PROGRESS

Q.1: 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.

Q.2: 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 pseudocodes. 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

Let us see at some more examples of pseudocodes in terms of computer programming.

Example 1.7: Write a pseudocode to find area of a rectangle.

Solution:

READ height of rectangle

READ width of rectangle

COMPUTE area as height * width

Example 1.8: Write a pseudocode for finding the average of 5 numbers.

Solution: **Input** 5 numbers

sum = add numbers together

avg = sum / 5

Display avg

Example 1.9: 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

Q.3: Is there any standard rule when writing pseudocode?

Q.4: 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.

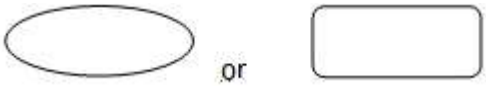
Normally, an algorithm is expressed as a flowchart and then the flowchart is converted into a program using some programming language. Flowcharts are independent of the programming language that are being used. Hence, one can fully concentrate on the logic of the problem solving at this stage.

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 ***ovals***, 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.

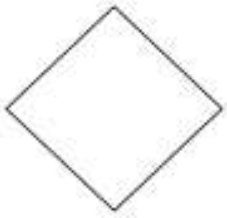
- **Process**: A process or computation represents arithmetic and data movement instructions in the flowchart. It is generally represented by using a ***rectangle***. All arithmetic processes of addition, subtraction, multiplication and division are indicated in the process symbol. If there are more 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.



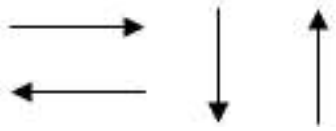
- **Decision :** The decision symbol is represented by using a ***diamond***. It is used in a flowchart to indicate the point where a

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.



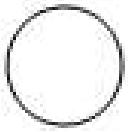
- **Flow lines/Arrows:** Flow lines (or, Arrows) are solid lines with arrowheads which indicate the flow of operation. They show the exact sequence in which the instructions are to be executed. The normal flow of the flowchart is depicted from top to bottom and from left to right.



- **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 together in Figure 1.1:

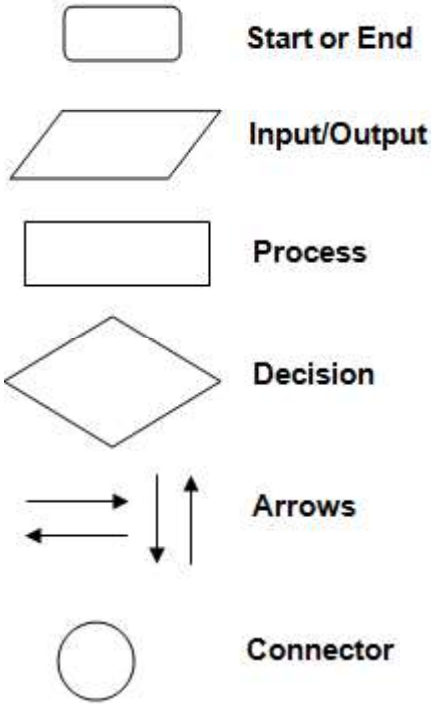


Figure 1.1: Flowchart Symbols

1.5.2 Advantages and Limitations of Flowchart

Advantages of 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

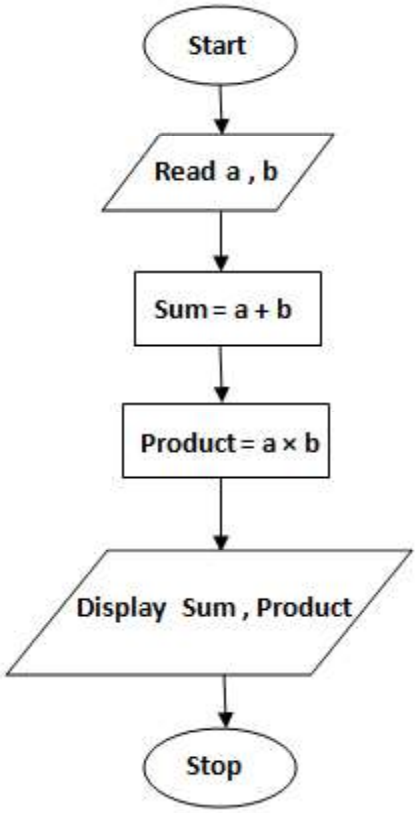
Q.5: What is a flowchart?

Q.6: Give any two advantages of flowcharts.

Let us look at few examples of flowcharts:

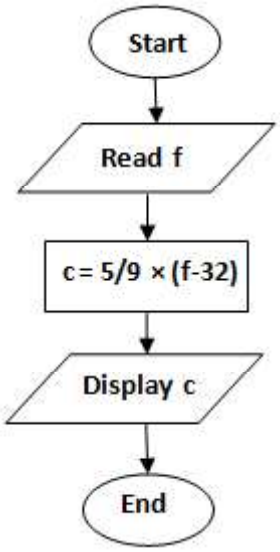
Example 1.10: Draw the flowchart to find the sum and product of two given numbers.

Solution:



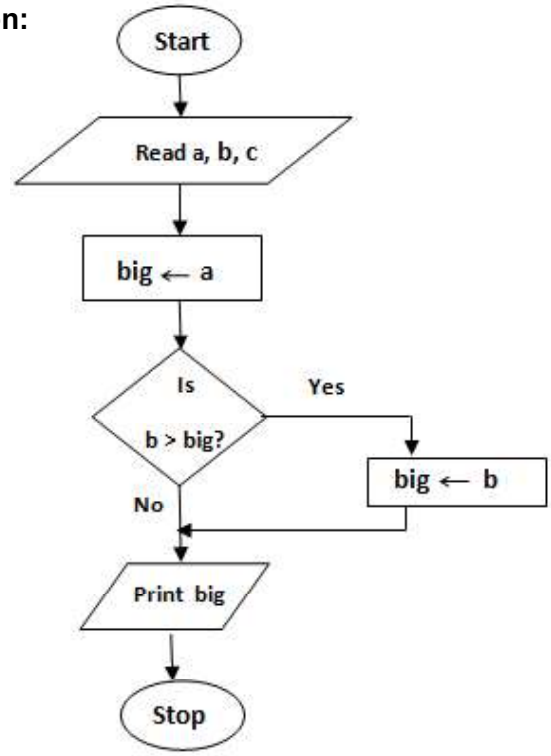
Example 1.11: Draw the flowchart to convert the temperature in Fahrenheit (^{0}f) to Centigrade (^{0}c).

Solution:



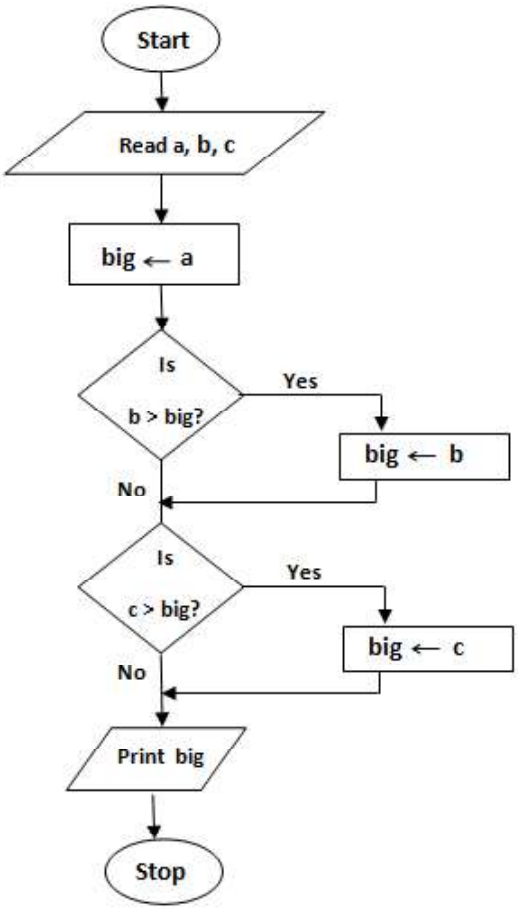
Example 1.12: Draw the flowchart to find the biggest of the two given numbers.

Solution:



Example 1.13: Draw a flowchart to find the biggest of three given numbers.

Solution:





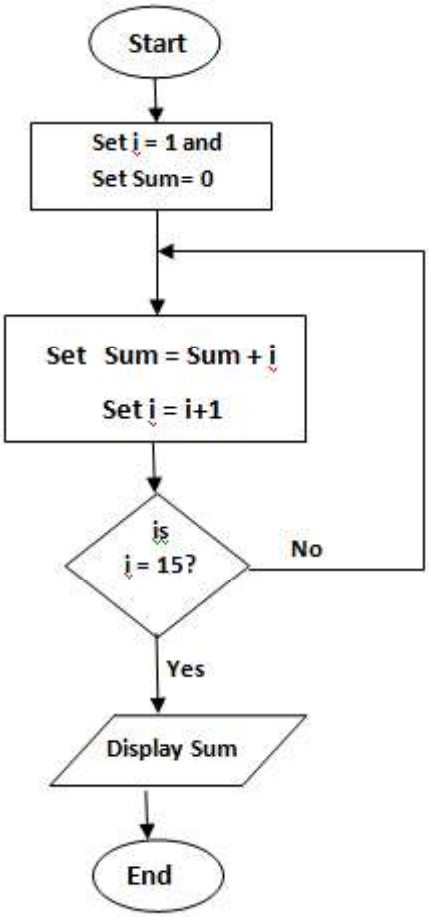
ACTIVITY

Q.1: 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]

Q.2: Draw a flowchart to subtract two numbers.

Example 1.14: Draw a flowchart to calculate the sum of first 15 natural numbers.

Solution:





CHECK YOUR PROGRESS

Q.7: In a flowchart, which symbol is represented using a rectangle?

a) Decision
b) Connector
c) Input/Output
d) Process

Q.8: Which one of the following is a graphical or symbolic representation of a process?

a) Algorithm b) Pseudocode

c) Flowchart d) Program

Q.9: 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 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



1.8 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.



1.9 MODEL QUESTIONS

Q.1: Define algorithm. How is it useful in the context of software development?

Q.2: What do you understand by the term pseudocode?

Q.3: Differentiate between algorithm and pseudocode.

Q.4: Write an algorithm to find the smallest of three given numbers.

Q.5: Write the algorithm to find the average of three given numbers.

Q.6: Write an algorithm to find the sum of natural numbers upto N.

Q.7: Write an algorithm to find the area of a triangle.

Q.8: Write an algorithm to find the sum of a set of numbers.

Q.9: Write an algorithm to find the factorial of a given number.

Q.10: Write a pseudocode to compute the average of *n* numbers.

Q.11: What is a flowchart? What are the significance of flowchart?

Q.12: With the help of an example explain the use of a flowchart.

Q.13: How is a flowchart different from an algorithm? Do we need to have both of them for developing a program?

Q.14: Draw the flowchart to find the area and circumference of a circle of radius r.

Q.15: Draw the flowchart to find the smallest of three given numbers.

Q.16: Discuss the advantages and limitations of flowchart.

Q.17: Describe the symbols used in flowchart.

*** ***** ***

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

2.6 Reserved Words

2.7 Constants

2.8 Variables

2.9 Let Us Sum Up

2.10 Answers to Check Your Progress

2.11 Further Reading

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 etc.
- define variables in C language.

2.2 INTRODUCTION

We have already learned about basic programming concepts including pseudo code, algorithm and flowcharts in the introductory unit. In this unit, we will introduce some basic concepts of C language like identifiers, keywords and constants etc. The C language was developed by Dennis Ritchie at Bell Laboratories during the 1970's.

2.3 C CHARACTER SET

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

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

2.4 TOKENS

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

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

The following names are valid identifiers:

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

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

x”	illegal characters (“).
order-no	illegal character (–)
total sum	illegal character (blank space)

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. So you canot use these reserved words as variable names. Since these reserved words have some special meaning in C language, therefore these words are often known as “keyword”. The following list shows all the reserved words available in C language:

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*	*igned*	*++ while*

2.7 CONSTANTS

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

There are two broad categories of constant in C, ***literal constant*** and ***symbolic constant***. A literal constant is just a value. For example, 10 is a literal constant. It does not have a name; it is just a literal value.

Depending on the type of data, literal constant is of different types. They include *integer*, *character* and *floating point* constant. Integer constant can again be subdivided into *decimal* (base-10), *octal* (base-8), and *hexadecimal* (base-16) integer constant. One important variation of character constant is *string constant*. Remember that a character constant is always enclosed with single quotation mark, whereas a string constant is always enclosed with a double quotation mark. Another point to remember is that an octal integer constant always starts with 0 and a hexadecimal constant with 0x.

EXAMPLE	TYPES
153	Decimal integer constant
015	Octal integer constant
0xA1	Hexadecimal integer constant
153.371	Floating point constant
‘a’	Character constant
‘1’	Character constant
“a”	String constant
“153”	String Constant

2.8 VARIABLES

A variable is an identifier to store value. You can identify a variable with a container which takes different values at different times 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 character 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. All together 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 variable in C language is an entity whose value may vary during program execution.



2.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: Identifiers are the 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.

Ans. to Q. No. 2: 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.

Ans. to Q. No. 3: There are two broad categories of constant in C, ***literal constant*** and ***symbolic constant.***

Ans. to Q. No. 4: 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.



2.11 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.



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: What are variables and tokens in C language?

Q.5: Give four different types of token with suitable example.

*** ***** ***

UNIT 3: VARIABLES AND DATA TYPES

UNIT STRUCTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Basic Data Types in C

3.4 C Variables and their Declarations

3.5 Symbolic Constants

3.6 Let Us Sum Up

3.7 Answers to Check Your Progress

3.8 Further Reading

3.9 Model Questions

3.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

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

3.2 INTRODUCTION

We have already learned the elements of C programming in the previous unit.

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

3.3 BASIC DATA TYPES IN C

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 explains the different categories of C data type.

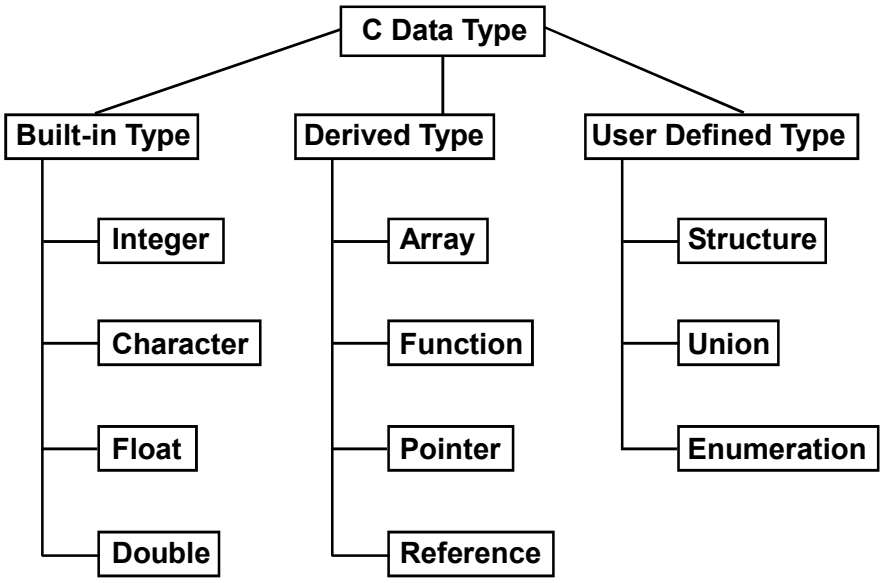


Fig 3.1: Data Types in C language

The first category of data type is the ***built-in*** data type, which is also known as elementary or basic type. Sometime these are called the "*primitive*" type. These basic data types 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:

Type	Size (in Bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed sort int	4	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647

float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E-308
long float	10	3.4E-4932 to 1.1E+4932

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

Moreover, besides these basic data types, another special data type is ***void***. The void type specifies the return type of a function, when it is not returning any value. Sometime void is used to indicate an empty parameter to a function. After all, this data type holds the literal meaning of void.

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.

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

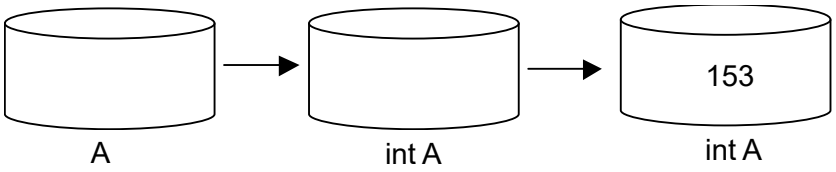
Table 3.2: Format specifier for usual variable type

3.4 C VARIABLES AND THEIR DECLARATIONS

We have already introduced variables in the previous unit (Unit 2). Now, in this unit, let us try to understand how varibales are initialized and declared.

We already know that a variable is an identifier to store value. 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. Again we are taking the same example of storing 153 in a variable. Suppose you want to store value 153 to a variable. What you will do is – first create the name of the variable, suppose A. Since 153 is integer, so declare the variable A as integer, and then assign 153 to that variable.



Now, in C programming language, this can be done using the following statement:

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

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

data_type variable1, variable2, ... , variableN;

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

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

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

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

The first method is already shown in the example. In second approach you can make a call of C standard input function (that is *scanf,*

getch, getc, gets etc.) to store value to a variable. For example, the previous 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.

We can also declare and initialize a variable in a single statement as follows:

int A = 153;

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

int j, k;

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

int j;

int k;

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

int j, k;

float x,y,z;

A few examples of variable declarations are shown below :

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

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. which 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?



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 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: The built in data types are *integer, character, float, double*.

Ans. to Q. No. 2: 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.

Ans. to Q. No. 3: 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.



3.8 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.



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 initialized? Explain with example.

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

Q.5: How are the variables declared in C? Give a suitable example.

Q.6: Give examples of built-in and derived data types.

Q.7: Give the difference between signed and unsigned integers.

Q.8: Write the size in bytes of the following data types:
char, float, int, long int, double.

Q.9: Write the format specifiers for: array of char, int, unsigned int, char.

*** ***** ***

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 Answers to Check Your Progress

4.9 Further Reading

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 you 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.



Operands: The data items that operators act upon are called operands.

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.

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

Table 4.1: Arithmetic Operators

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:

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

Table 4.2 Relational operators

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:

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

Table 4.3: Logical Operators

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.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive
<<	Shift left
>>	Shift right

Table 4.4: Bitwise operators

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

Bitwise AND (&) Operator: The bitwise AND operator performs logical operations on a bit-by-bit level using the following truth table:

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

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

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

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

The output will be 12.

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

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

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

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

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

```
        c = a | b;              // 61 = 0011 1101
}
```

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

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

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

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

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

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

bit x of op2	result
0	0
0	1

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

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

Bitwise Shift Operators: C provides two bitwise shift operators, bitwise left shift (**<<**) and bitwise right shift (**>>**), for shifting bits left

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

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 supports some special operators such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->). The size of 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 helps 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 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.

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

Table.4.9: Precedence and Associativity

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

```
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

C Expressions 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 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



4.9 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.



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: PREPROCESSOR DIRECTIVES AND 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 Answers to Check Your Progress

5.10 Further Reading

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<steio.in>
#include<conio.in
void main()
{
    clrscr();
    printf("\n%d", 4);
    printf("\n%2d", 4);
    printf("\n%3d", 4);
    printf("\n%-3d", 4);
    getch();
}
```

%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 S.S. 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 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();
}
```



5.10 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.



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 Conditional Operator Statement

6.9 Let Us Sum Up

6.10 Answers to Check Your Progress

6.11 Further Reading

6.12 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.

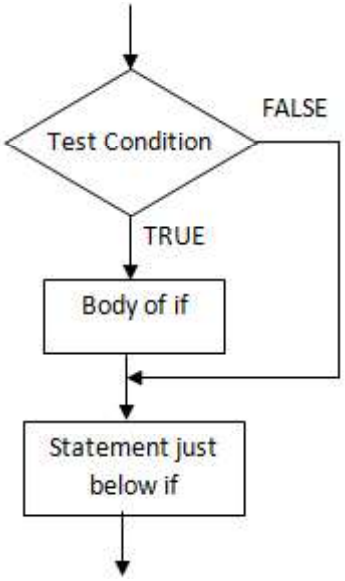


Fig. 6.1: Control transfer in if statements

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.

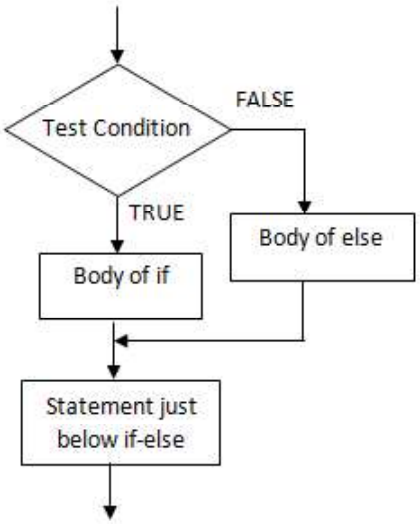


Fig 6.2: Control transfer in if-else statements

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.

6.8 *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();
}
```



6.9 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.



6.10 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 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.



6.12 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 Answers to Check Your Progress

7.6 Further Reading

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.

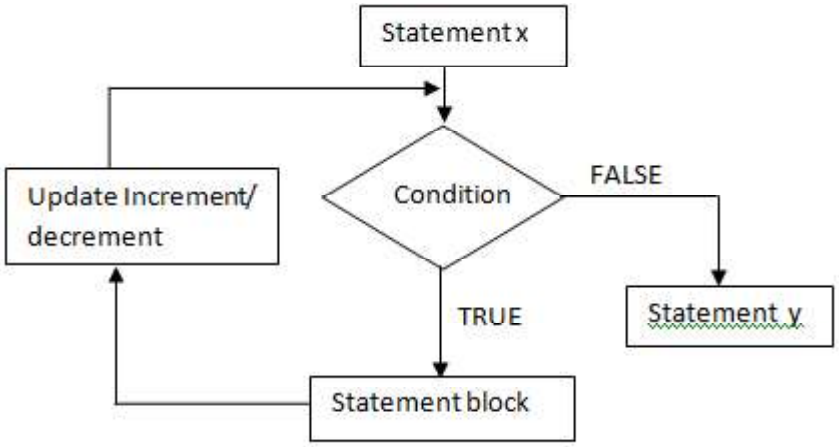


Fig. 7.1: Flow chart of while loop

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.

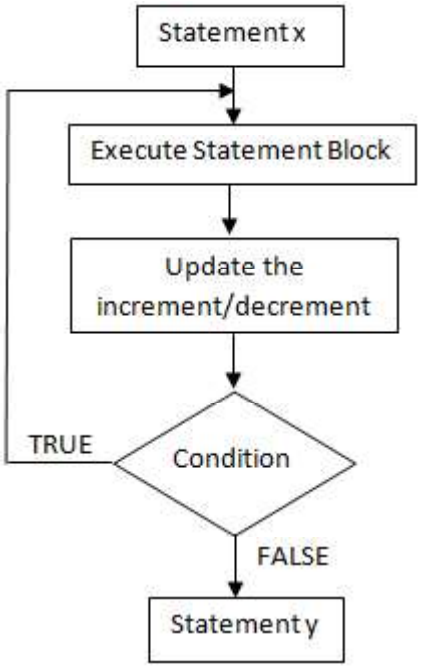


Fig. 7.2: Flow chart of *do-while* loop

//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 /***ncrement/ 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*:

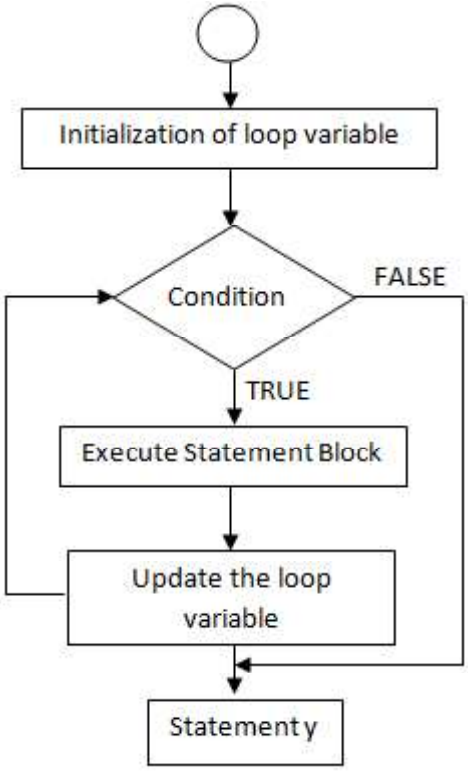


Fig.7.3: Flow chart of *for* loop

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 fort 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 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");
    }
```



7.6 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.



7.7 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 programm using d--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.

*** ***** ***