CHAPTER 1

INTRODUCTION TO FLOWCHARTING

1.0 OBJECTIVES

Friends,After studying this topic you will be able to -

- describe problem solving
- describe the meaning of flowcharts and flowcharts as a tool to represent program logic sequence.
- explain types of flowcharts and flowchart symbols.
- state uses of flowcharts and advantages of flowcharts
- describe develop flowcharts for problem solving.
- describe the advanced flowcharting techniques involved in flowcharts for computations, decision making, loops, predefined process etc.

1.1 INTRODUCTION

Computers are capable of handling various complex problems which are tedious and routine in nature. In order that a computer solve a problem, a method for the solution and a detailed procedure has to be prepared by the programmer. The problem solving Involves :

- Detailed study of the problem

- Problem redefinition

- Identification of input data, output requirements and conditions and limitations

- Alternative methods of solution

- Selection of the most suitable method

- Preparation of a list of procedures and steps to obtain the solution

- Generating the output

The preparation of lists of procedures and steps to obtain the result introduces the algorithmic approach to problem solving. The algorithm is a sequence of instructions designed in such a way that if the instructions are executed in a specific sequence the desired results will be obtained. The instructions should be precise and concise and the result should be obtained after a finite execution of steps. This means that the algorithm should not repeat one or more instructions infinitely. It should terminate at some point and result in the desired output.
An algorithm should possess the following characteristics :

- Each and every instruction should be precise and clear

- Each instruction should be performed a finite number of times

- The algorithm should ultimately terminate

- When the algorithm terminates the desired result should be obtained.

1.2 FLOWCHARTS

Before you start coding a program it is necessary to plan the step by step solution to the task your program will carry out. Such a plan can be symbolically developed using a diagram. This diagram is then called a flowchart. Hence a flowchart is a symbolic representation of a solution to a given task. A flowchart can be developed for practically any job. Flowcharting is a tool that can help us to develop and represent graphically program logic sequence. It also enables us to trace and detect any logical or other errors before the programs are written.

1.3 TYPES OF FLOWCHARTS

Computer professionals use two types of flowcharts viz :

- Program Flowcharts.

- System Flowcharts

1.3.1 Program Flowcharts :

These are used by programmers. A program flowchart shows the program structure, logic flow and operations performed. It also forms an important part of the documentation of the system. It broadly includes the following:

- Program Structure.

- Program Logic.

- Data Inputs at various stages.

- Data Processing

- Computations and Calculations.

- Conditions on which decisions are based.

- Branching & Looping Sequences.

- Results.

- Various Outputs.

The emphasis in a program flowchart is on the logic.

1.3.2 System Flowcharts :

System flowcharts are used by system analyst to show various processes, sub systems, outputs and operations on data in a system.

In this course material we will be discussing program flowcharts only.

1.2 & 1.3 Check Your Progress.

Answer in 1-2 sentences :

a) What is an algorithm?

.....

.....

b) What is a flowchart?

.....

.....

c) What are the types of flowcharts?

.....

.....

d) List any two steps involved in problem solving.

.....

.....

1.4 FLOWCHART SYMBOLS

Normally, an algorithm is expressed as a flowchart and then the flowchart is converted into a program with the programming language. Flowcharts are independent of the programming language being used. Hence one can fully concentrate on the logic of the problem solving at this stage. A large number of programmers use flowcharts to assist them in the development of computer programs. Once the flowchart is fully ready, the programmer then write it in the programming language. At this stage he need not concentrate on the logic but can give more attention to coding each instruction in the box of the flowchart in terms of the statements of the programming language selected.

A flowchart can thus be described as the picture of the logic to be included in the computer program. It is always recommended for a beginner, to draw flowcharts prior to writing programs in the selected language. Flowcharts are very helpful during the testing of the program as well as incorporating further modifications.

Flowcharting has many standard symbols. Flowcharts use boxes of different shapes to denote different types of instructions. The actual instruction is written in the box. These boxes are connected with solid lines which have arrowheads to indicate the direction of flow of the flowchart. The boxes which are used in flowcharts are standardised to have specific meanings. These flowchart symbols have been standardised by the American National Standards Institute. (ANSI).

While using the flowchart symbols following points have to be kept in mind:

- The shape of the symbol is important and must not be changed.
- The size can be changed as required.
- The symbol must be immediately recognizable.
- The details inside the symbol must be clearly legible.
- The flow lines, as far as possible, must not cross.

Terminal Symbol:

Every flowchart has a unique starting point and an ending point. The 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. *There can be only one*

START and one STOP terminator in you entire flowchart. In case a program logic involves a pause, it is also indicated with the terminal symbol.

Input/Output Symbol :

This symbol is used to denote any input/output function in the program. Thus if there is any input to the program via an input device, like a keyboard, tape, card reader 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, plotters, magnetic tapes, disk, monitors etc. are indicated in the Input/Output symbol.

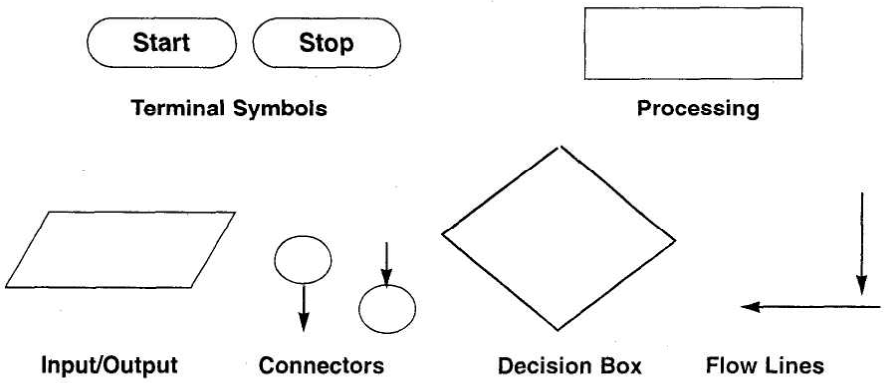


Fig. 1 : Flowchart Symbols

Process Symbol :

A process symbol is used to represent arithmetic and data movement instructions in the flowchart. All arithmetic processes of addition, subtraction, multiplication and division are indicated in the process symbol. The logical process of data movement form one memory location to another is also represented in the process box. If there are more than one process instructions to be executed sequentially, they can be placed in the same process box, one below the other in the sequence in which they are to be executed.

Decision Symbol :

The decision symbol 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 in the decision box. All the possible paths should be accounted for. During execution, the appropriate path will be followed depending upon the result of the decision.

Flowlines :

Flowlines 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 left to right.

Connectors :

In situations, where the flowcharts becomes big, it may so happen that the flowlines 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 in the circle. This letter or digit indicates a link. A pair of such identically labelled connectors are used to indicate a continued flow in situations where flowcharts are complex or spread over more than one page. Thus a connector indicates an exit from some section in the flowchart and an entry into another section

of the flowchart. If an arrow enters a flowchart but does not leave it, it means that it is an exit point in the flowchart and program control is transferred to an identically labelled connector which has an outlet. This connector will be connected to the further program flow from the point where it has exited. Connectors do not represent any operation in the flowchart. Their use is only for the purpose of increased convenience and clarity.

1.4 Check Your Progress.

1. Answer in 1- 2 sentences

a) What is the use of decision box in flowcharts?

.....

.....

b) What do flowlines show?

.....

.....

2. Match the following :

Column A	Column B
a) Connector	(i) show the sequence of instructions execution
b) Input/Output	(ii) represent arithmetic and data movement instructions
c) Process	(iii) represents entry or exit to another part of flowchart.
d) Flow Lines	(iv) denote any input/output function

1.5 ADVANTAGES OF FLOWCHARTS

There are a number of advantages when using flowcharts in problem solving. They provide a very powerful tool to programmers to first represent their program logic graphically and independent of the programming language.

- Developing the program logic and sequence. A macro flowchart can first be designed to depict the main line of logic of the software. This model can then be broken down into smaller detailed parts for further study and analysis.

- A 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 a program

- It shows the execution of logical steps without the syntax and language complexities of a 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 carried out. Flowcharts can thus be used as working models in design of new software systems.

- Flowcharts provide a strong documentation in the overall documentation of the software system.

- Once the flowchart is complete, it becomes very easy for programmers to write the program from the starting point to the ending point. Since the flowchart is a detailed representation of the program logic no step is missed during the actual program writing resulting in error free programs. Such programs can also be developed faster.

- A flowchart is very helpful in the process of debugging a program. The bugs can be detected and corrected with the help of a flowchart in a systematic manner.- A flowchart proves to be a very effective tool for testing. Different sets of data are fed as input to program for the purpose

1.5 Check Your Progress.

1. **Give any two advantages of flowcharts.**

   .....

   .....

1.6 DEVELOPING FLOWCHARTS

In developing the flowcharts following points have to be considered:

- Defining the problem.
- Identify the various steps required to form a solution.
- Determine the required input and output parameters.
- Get expected input data values and output result.
- Determine the various computations and decisions involved.

With this background of flowcharts and flowchart symbols let us now draw some sample flowcharts. First we shall write the steps to prepare the flowchart for a particular task and then draw the flowchart.

Example : To prepare a flowchart to add two numbers. (Fig. 2a.)

The steps are :

1. Start.
2. Get two numbers N1 and N2.
3. Add them.
4. Print the result.
5. Stop.

Example : To prepare a flowchart to determine the greatest of two numbers. Here we use the decision symbol. We also combine the two reads for numbers A and B in one box.

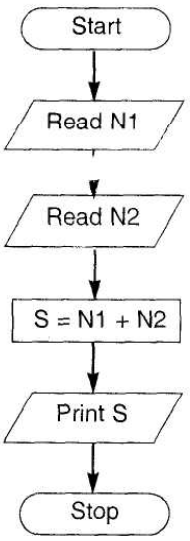


Fig. 2a) Flowchart to add two numbers N1 and N2

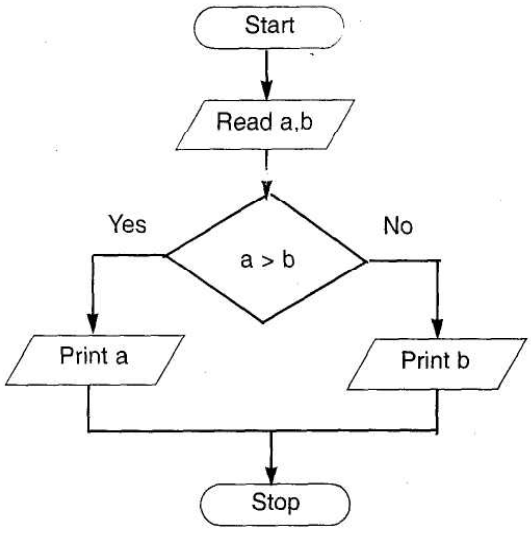


Fig. 2b) Flowchart to determine the greater of two numbers a and b

The steps are :

1. Start

2. Get two number A and B.

3. If A > B then print A else print B.

4. Stop.

Note that in the first example, we have used two separate input/output boxes to read the numbers N1 and N2. In the second example, both the numbers a and b are read in the same box. Thus if more than one instructions of the same kind follow one another then they can be combined in the same box.

1.6 Check Your Progress.

1. Write the steps and draw the flowcharts for the following :

a) Find the average of three numbers a, b and c.

b) Find the area of a rectangle whose length and breadth are given.

2. Answer the following :

a) What are the points to be considered in developing flowcharts?

.....

.....

1.7 TECHNIQUES

In this section we shall cover the various flowcharting techniques viz.

- flowcharts for computations
- flowcharts for decision making
- flowcharts for loops
- Predefined Process

1.7.1 Flowcharts for Computations :

Computers are used to perform many calculations at high speed. When you develop a program it also involves several calculations.

The general format of the flowcharting steps for computations is :

- Create memvars used in calculations and read operation.

- Get required data input using memvars.

- Perform the necessary calculations.

- Print the result.

Programming considerations while using computation techniques : Most languages have provision for creating memvars. The exact syntax depends on the language used. In most cases (but not all) your programs have to create and initialize the memvars before you can use them.

The following examples show the usage of flowcharts in computations. The flowcharts are shown in Fig.3a and Fig. 3b.

Example : Flowchart for a program that converts temperature in degrees Celsius to degrees Fahrenheit.

First let us write the steps involved in this computation technique.

1. Start.

2. Create memvars F and C (for temperature in Fahrenheit and Celsius).

2. Read degrees Celsius into C.

3. Compute the degrees Fahrenheit into F.

4. Print result (F).

5. Stop.

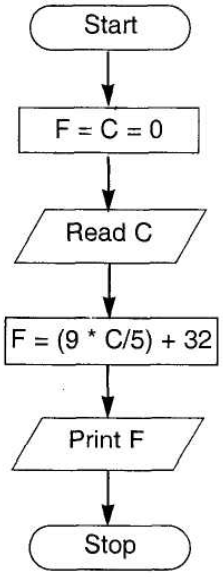


Fig. 3a) Flowchart to convert temperature from Celsius to Fahrenheit

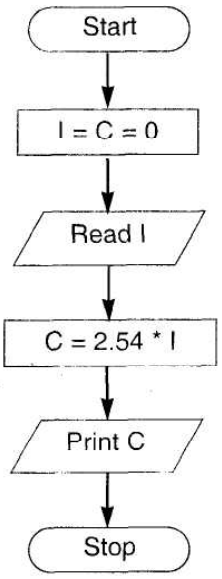


Fig. 3b) Flowchart to convert inches to centimetres

Example : Flowchart for a program that converts inches to centimeters First let us write the steps involved in this computation technique.

1. Start.
2. Create memvars C and I (for Centimeters and Inches respectively).
2. Read value of Inches into I
3. Compute the Centimeters into C.
4. Print result (C).
5. Stop.

1.7.2 Flowcharts for decision making :

Computers are used extensively for performing various types of analysis. The decision symbol is used in flowcharts to indicate it.

The general format of steps for flowcharting is as follows:

- Perform the test of the condition.
- If condition evaluates true branch to Yes steps.
- If condition evaluates false branch to No steps.

Programming Considerations :

Most programming languages have commands for performing test and branching. The exact commands and syntax depends on the language used. Some of the conditional constructs available in programming languages for implementing decision making in programs are as follows:

- If
- If - else - endif
- If - elseif - endif

- Do case - endcase.
- Switch.

All languages do not support all of the above constructs.
The operators available for implementing the decision test are as follows:
- Relational Operators (which determine equality or inequality)
- Logical Operators, (useful for combining expressions)

The branching to another set of commands can be implemented by using functions, procedures etc.

Example: Flowchart to get marks for 3 subjects and declare the result. If the marks >= 35 in all the subjects the student passes else fails.

The steps involved in this process are :

1. Start.
2. Create memvars m1, m2, m3.
3. Read marks of three subjects m1, m2, m3.
4. If m1 >= 35 goto step 5 else goto step 7
5. If m2 >= 35 goto step 6 else goto step 7
6. If m3 >= 35 print Pass. Goto step 8
7. Print fail
8. Stop

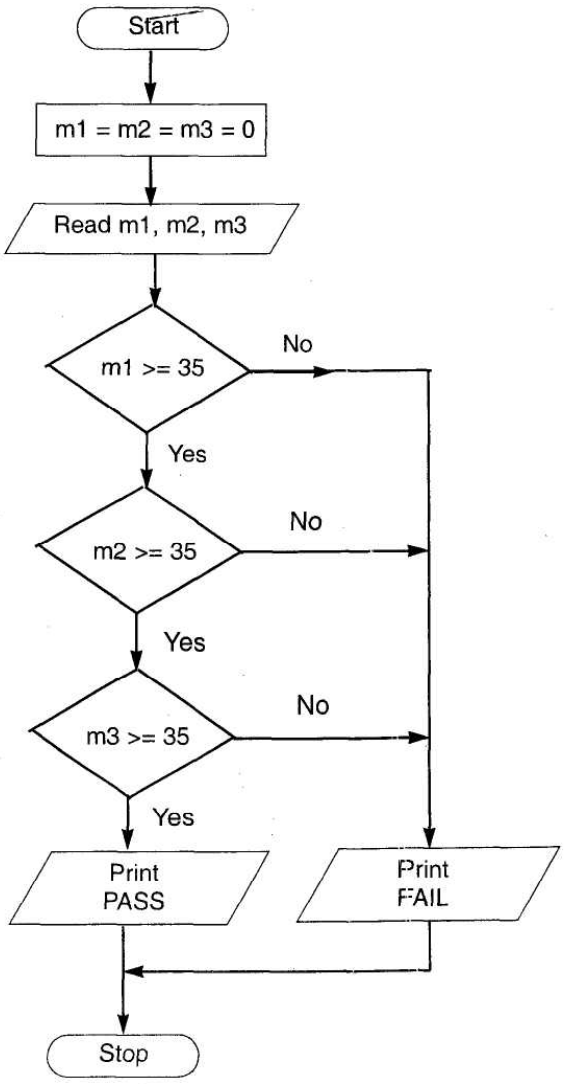


Fig. 4 Flowchart to determine whether pass or fail making using of decision box

The flowchart is shown in Fig. 4.

An alternative method is the one in which you can combine all the conditions

with the AND operator. The steps then would be :

1. Start
2. Create memvars m1, m2, m3.
3. Read marks of three subjects into m1, m2 and m3.
4. If m1 >= 35 and m2 >= 35 and m3 >= 35 print Pass. Otherwise goto step 5.
5. Print Fail.
6. Stop

Developing this flowchart is left as an exercise to the student.

1.7.3 Flowcharts for loops

Looping refers to the repeated use of one or more steps. i.e. the statement or block of statements within the loop are executed repeatedly. There are two types of loops. One is known as the fixed loop where the operations are repeated a fixed number of times. In this case, the values of the variables within the loop have no effect on the number of times the loop is to be executed. In the other type which is known as the variable loop, the operations are repeated until a specific condition is met. Here, the number of times the loop is repeated can vary.

The loop process in general includes :

- Setting and initialising a counter

- execution of operations

- testing the completion of operations

- incrementing the counter

The test could either be to determine whether the loop has executed the specified number of times, or whether a specified condition has been met.

Programming considerations :

Most of the programming languages have a number of loop constructs for efficiently handling repetitive statements in a program. These include :

- do-while loop

- while loop

- for loop

- for-next loop

In most of the looping situations, we make use of counters. In situations where the loop is to be repeated on the basis of conditions, relational operators are used to check the conditions.

Example : To find the sum of first N numbers. This example illustrates the use of a loop for a specific number of times. Fig. 5a.
The steps are :

1. Start
2. Create memvars S , N, I
3. Read N
4. Set S (sum) to 0
5. Set counter (I) to 1.
6. S = S + I
7. Increment I by 1.
8. Check if I is less than or equal to N. If no, go to step 6.

9. Print S
10. Stop

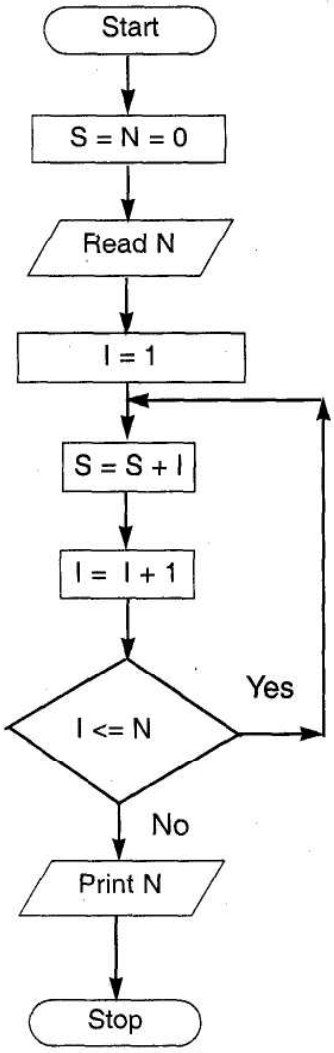


Fig.5a Flowchart to find sum of first N numbers

The flowchart is shown in Figure 5a.

Example : To check whether character read from keyboard is Z. If it is Z then print END, else read another character. This example shows the test which executes till a particular condition is satisfied.

The steps are :

1. Start
2. Create memvar C
3. Read C,
4. Check if C = 'Z'. If no goto step 3.
5. Print END
6. Stop

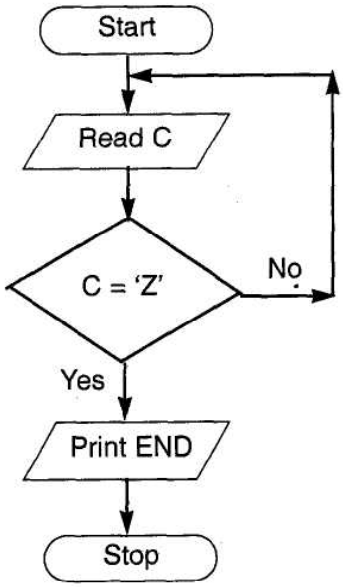


Fig.5b Flowchart to read a character till Z is input

The flowchart is shown if figure 5b.

1.7.4 Predefined Process

In a large application, we use programs written by others. Also when we invoke a library routine of a language, we are using predefined process. In a predefined process the required inputs are known and the expected output. We do not know how the routine handles the task. For us it is a like a black box. Predefined processes have

great use as they enable us to use programs written by others and save a lot of time. It also permits the integration of various parts of the software into a single unit. The predefined routine can be put at any place in the flowchart. It is a single symbol of flowchart that represents an entire flowchart created elsewhere.

Programming Considerations:

Today structured and modular programming is accepted as the best way to developed software applications. Each module treats the other module as a predefined process. The development of the library routines also envisions the use of predefined process. It prevents us from having to write separate programs again and again each time to do the same task, in different applications.

1.7 Check Your Progress.

1. **Write the programming considerations for the following** :

a) Computations :

.....

.....

b) Decision making :

.....

.....

2. Draw the flowcharts for the following :

a) Printing the first five odd numbers.

b) Read age of a person. If age less than 60 then print “Not a senior citizen” otherwise print “Senior Citizen”.

1.8 SUMMARY

In this chapter we learnt the concept of Algorithm & flowchart

- The algorithm is a sequence of Instructions designed in such a way that if the instructions are executed in a specific sequence the desired result will be obtained.
- A Flowchart is a symbolic representation of a solution to a given task.
- Program flowchart & System flowchart are two types of flowcharts
- Flowchart uses many symbols/shapes to denote different types of instructions.
- Flowchart symbols have been standardized by the American Standard Institute.

At the end we studied Flowcharts for Computation, Flowcharts for decision making, flowcharts for loops and flowchart for predefined process.

Source *http://jayaram.com.np* (e book)

1.9 CHECK YOUR PROGRESS - ANSWERS

1.2 & 1.3

1. a) An algorithm is a sequence of instructions designed in such a way that if the instructions are executed in a specific sequence the desired results will be obtained. The instructions should be precise and concise and the result should be obtained after a finite execution of steps.

b) A flowchart is a symbolic representation of a solution to a given task. Flowcharting is a tool that can help us to develop and represent graphically program logic sequence.

c) There are two types of flowcharts : Program Flowcharts which are used by the programmers and which show the program structure, logic flow and operations performed. It also forms an important part of the documentation of the system and system flowcharts which are used by system analyst to show various processes, sub systems, outputs and operations on data in a system.

d) Any two steps in problem solving are :
(i) Detailed study of the problem
(ii) Identification of input data, output requirements and conditions and limitations

1.4

1. a) The decision box 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 in the decision box. All the possible paths should be accounted for. During execution, the appropriate path will be followed depending upon the result of the decision.

b) Flowlines are solid lines with arrowheads which are used to indicate the flow of operation. They show the exact sequence in which the instructions are to be executed.

2. a - (iii)
b - (iv)
c - (ii)
d - (i)

1.5

1. Advantages of Flowcharts :

(i) A flowchart shows the execution of logical steps without the syntax and language complexities of a program.

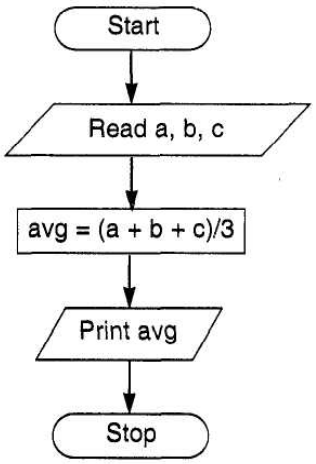
(ii) In real life programming situations a number of programmers are associated with the development of a system and each programmer is assigned a specific task of the entire system. Hence, each programmer can develop his own flowchart and later on all the flowcharts can be combined for depicting the overall system. Any problems related to linking of different modules can be detected at this stage itself and suitable modifications carried out. Flowcharts can thus be used as working models in design of new software systems.

1.6

1. a) To find average of three numbers a, b ,c.

b) To find area of a rectangle whose length and breadth area read.

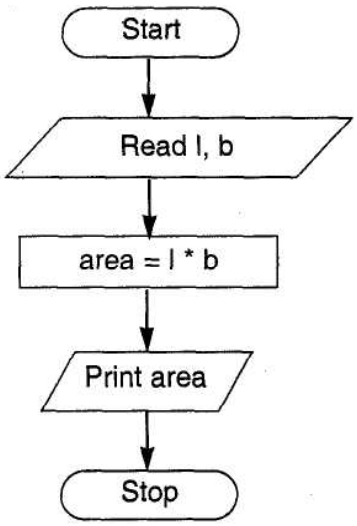
2. While developing flowcharts the points to be taken into consideration are :



The steps are :

1. Start
2. Read numbers a, b, c
3. Compute the average as (a + b + c)/3
4. Print average
5. Stop.

(i) Defining the problem.



The steps are :

1. Start
2. Read length l and breadth b
3. Compute the area as l * b
4. Print area
5. Stop.

(ii) Identifying the various steps required to form a solution.

(iii) Determining the required input and output parameters.

(iv) Getting expected input data values and output result.

(v) Determining the various computations and decisions involved.

1.7

1.

a) Programming considerations for computation : Most languages have provision for creating memvars. The exact syntax depends on the language used. In most cases your programs have to create and initialize the memvars before you can use them.

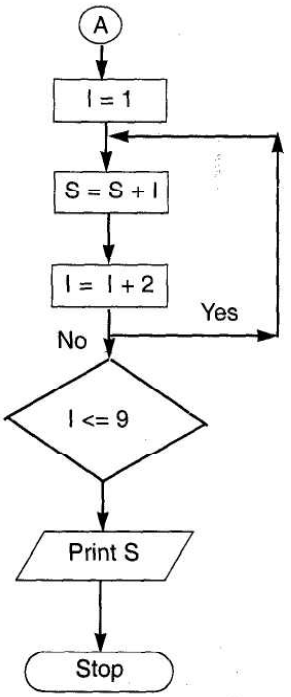
b) Programming considerations for loops : Most programming languages have commands for performing test and branch. Some of the conditional constructs available in programming languages for implementing decision making in programs are as follows: If, If - else - endif, If - elseif - endif, Do case - endcase, Switch.

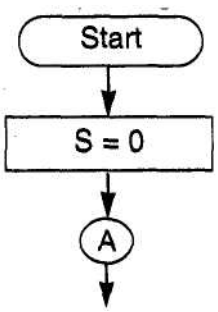
The operators available for implementing the decision test are as follows:

Relational Operators, logical Operators.

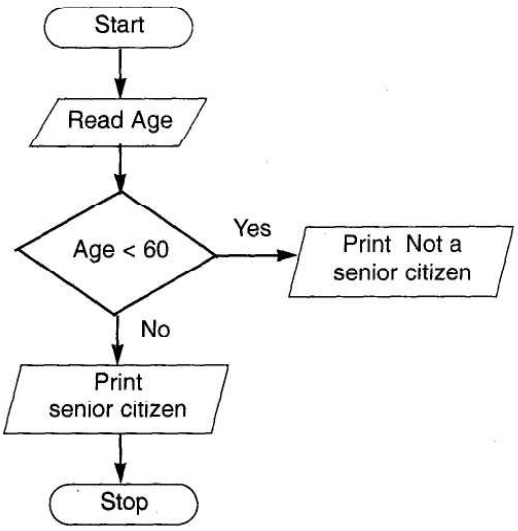
The branching to another set of commands can be implemented by using functions, procedures etc

2.a) Flowchart for printing the first five odd numbers :





b) Read age and print comment



1.10 QUESTIONS FOR SELF-STUDY

1) Answer the following in 7-8 Sentences

a) What characteristics should an algorithm have?
b) State the important points to be considered when developing flowcharts.
c) What is meant by fixed loops and variable loops?

2) Write short note on the following in 10-12 lines

a) Flowcharts for loops
b) Flowcharts for decision making

3) Write the algorithm and draw flowcharts for the following :-

a) Convert distance entered in Km to Metres.
b) Find the product of the just n numbers.
c) Find the sum of digits of a three digit number.

1.11 SUGGESTED READINGS

The Spirit of C : Mullish cooper

Let us C : Yashwant kanitkar

The C programming Language: Kernigham & Ritchie

CHAPTER 2

BEGINNING WITH C

2.0 OBJECTIVES

Friends, After studying the chapter you will be able to:

- state the development and background of C language
- explain how to create and run C programs
- begin learning the C language, its syntax and grammar
- explain what are C tokens and types of C tokens
- explain the various data types in C
- explain instruction in C and their types
- discuss what is meant by symbolic constants
- discuss to write simple program in C after having studied the above

2.1 INTRODUCTION

The chapter begins with the introduction of the C language. It describes the history, development and features of the C language. This chapter will also begin the actual learning of the C language. Whenever we learn a new language, we should first learn what alphabets, symbols and numbers form a part of that language, We then use these to construct **keywords** and **variables** and finally combine all of the above to form meaningful instructions of that particular language.

A set of instructions written in order to execute a task is a **program.** These instructions are formed by using certain symbols and words according to certain rules.

These rules are known as the **syntax rules** or **grammar.** Each language has its own syntax rules. Every instruction in a program should strictly follow the syntax of the

particular language in which the program is being written. C language also has its own vocabulary and grammar.

In this chapter we begin with the basics of the C language. We have already learnt flowchart development in the previous chapter. We shall write the first few programs by first developing the flowchart for the program and then writing the actual program for the same. In the subsequent chapters however, the development of flowcharts for the example programs is left an exercise to the student. In this chapter, we shall cover the C tokens one by one. These include : Keywords, identifiers, constants, strings, special symbols and operators. We shall learn how to define them, how to use them and what are their types. Next we shall familiarise ourselves with the instructions in C and their types. Out of these instructions, data type declarations form a part of this chapter. The classification of data types is discussed out of which primary data types are covered. C arithmetic on various data types, symbolic constants, their declaration and use are also part of this chapter. When we learn this, it will be possible for us to write small but meaningful and complete programs in C.

2.2 INTRODUCTION TO C

2.2.1 Features of C :

The C programming language was developed at AT&T's Bell Laboratories in USA in 1972. This language has been designed and written by **Dennis Ritchie.** C is a very robust programming language and has a rich set of **built in functions.** C is simple to learn and easy to use. C language is suitable for writing both **system software** and **application programs** and business packages. The Unix operating system was also developed at Bell laboratories around the same time. Thus C is strongly associated with Unix. In fact, Unix is coded almost entirely in C.

As we have already seen, programming languages are catagorised as **high level programming languages** and **low level programming languages.** C is between these two catagories. The C compiler has the capabilities of the machine language and the features of a high level language. Initially C was mainly used in academic environments. But with its growing popularity C is now one of the most popular high level programming language and is running under many operating systems. C language is highly **portable** i.e. C programs written for one computer can be run on another computer with very little or no modifications.

2.2.2 What is a C program ?

C has only 32 **keywords** but a number of **built in functions.** Thus a C program is basically a collection of functions which are supported by the C library. We can make efficient use of these functions in our programming tasks. One can also add new functions to the library.

Before studying the language and its features in detail, let us begin by writing and executing a sample program written in C. This will illustrate the method of entering, editing, compiling the program.

Example : The first C program

```
main()
{
        /* The First C program*/
        printf("\nMy first C program");
        /*end of program*/
}
```

The output of this program will be :

My first C program

Let us now study this program.

The first line of the program is **main().** We have stated earlier that a C program is a collection of functions. **main()** is also a function. This **main()** is a special function in C. This is the line at which the execution of our program begins. Every program should have this function **main().** There is a pair of parenthesis after the function name which is necessary.

The open brace bracket'{' indicates the beginning of the function **main()** and the close brace bracket'}' in the last line marks the end of the function. The statements which are included within these braces make up the body of our function **main().** Thus the function body will contain the instructions to perform the given task. This means that we should include the set of statements belonging to the function in a pair of braces.

The lines beginning with the '/*' and ending with '*/' are called as **comment lines.** Comment lines are not executable statements and anything written between /* and */ is ignored by the compiler. We use the comments in a program to improve its understanding. Comments in our programs will help users and other programmers in understanding the programs easily. Debugging and testing also becomes easy when helpful comments are inserted at appropriate places in our programs.

Important points to be noted about comments :

- Comments must be enclosed within /*and */

- You can have any number of comments in your program

- Comments can be inserted at any place in a program but they can't be nested

- Comments can be written over more than one line.

printf is the next line of our program. This **printf** function is an executable function. **printf** is a predefined function for printing the output from a program. A predefined function means a function which has already been written and compiled. Thus the **printf** function will produce the output:

My first C program

The **printf** line ends with a semicolon ';'. Every statement in C must end with a semicolon. The '\n' character is a special character which outputs the statement on a new line. We shall learn about it subsequently.

Thus, we can see that our program is nothing but a set of separate statements. However, one must write these statements in the order in which they are to be executed. When writing a program in C the important points to remember include :

- You can insert blank spaces between two words to improve readability

- A statement can start from any position. There are no rules in C to indicate the position at which a statement should be written

- Each statement must end with a semicolon.

Although programs can be entered in uppercase or lowercase, uppercase letters are used only for **symbolic constants.** Hence one should make a habit of writing programs in lowercase.

Executing the program :

Now that we have seen the set of statements of our program, let us see how to execute this program. We shall use the MS DOS operating system and the C compiler for DOS while executing the program on our PC. The following steps should be carried out to execute the program :

- The program should be created and edited by using any word processing application in the non document mode.

- You should save your program file with the .c extension.

- Once you have saved your file with the .c extension, you compile your program. If any syntax errors are found during compilation they will be reported and the compilation will not be completed. You then remove the errors and once again compile the corrected program.

- Successful compilation will generate the object file of your program. This file will have the .obj extension. You then link the file and the executable code of your program will be generated. The executable file of the program has the .exe extension.

- This program can now be run at the DOS prompt by typing its name

Your C program has now been successfully written and compiled.

This was just a sample program to get started with C. We shall now commence the study of the C language and its grammar from the following sections.

2.1 & 2.2 Check Your Progress.

1. Fill in the blanks :

a) The C language is written by .....

b) C language has .....keywords.

c) ..... is the function in C where program execution begins.

d) ..... lines are not an executable part of the C program.

e) Every statement in C should end with a .....

2. Answer in two sentences.

a) What are the features of the C language?

.....

.....

b) What are comment lines ?

.....

.....

c) Write in short about main ( ).

.....

.....

2.3 THE C CHARACTER SET

Now we shall study the characters, numbers and special symbols of C. A character denotes any alphabet, digit or special symbol used to represent information. The characters are then used to form words, numbers and expressions.

The C character set comprises of the following :

Letters or Alphabets	**:**	A, B, C,……X, Y, Z
		a, b, c,…… x,y, z
Digits	:	0,1, 2, 3, 4, 5, 6,7,8,9
Special Symbols	:	,.;:?'"! | / \ ~ _ % & ^ *- + <> = (){}[]# @
White spaces	:	blanks, horizontal tabs, new line, form feed, carriage return

Note : White spaces may be used to separate words and improve readability of the program. But they cannot be used between the characters of keywords and identifiers.

2.4 C TOKENS

The next step after the definition of the character set is the formation of words, keywords etc. The smallest individual units in a C program are called **tokens.** The alphabets, numbers and special symbols are combined to forms these tokens. The types of tokens in C are :

- Keywords
- Identifiers
- Constants
- Strings
- Special Symbols
- Operators

We shall study each of these tokens in detail.

2.4.1 Keywords and identifiers :

Keywords :

Keywords are words whose meanings have already been defined and these meanings cannot be changed. Keywords are also called as **reserved words.** Keywords should not be used as variable names (though some compilers allow you to construct variable names like keywords). All the keywords must be written in lowercase.

There are 32 keywords in C as given below :

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

At this moment it is not necessary for you to learn all these keywords. The detailed discussion of these keywords and their meanings shall be dealt with as and when we shall study their usage.

Identifiers :

Identifiers are names given to variables, functions and arrays. These are the names which are **user defined.** They are made up by a combination of letters and digits. Normally an identifier should not be more than 8 characters long. The use of underscore is also permitted in identifiers. However, it is imperative that identifiers should begin with a letter.

Some examples of identifiers are : min1

Max-temp

temp() etc.

C does not permit use of blank spaces, tabs, commas or special characters in identifiers. Thus:

mi n1

max*temp

12temp

are invalid identifiers in C.

2.4.2 Constants :

A constant in C is a **fixed value** which does not change during the execution of a program. C supports several types of constants as under:

The basic classification of constants is :

Numeric Constants and **Character constants**

These are further classified as :

Numeric	**Constants**	**Character**	**Constants**
Integer	Real	Single Character	String
Constants	constants	constants	constants

(i) Integer constants:

An **integer constant** is a sequence of digits. An integer constant must have at least one digit and cannot have a decimal point. An integer constant can be positive or negative. If there is no sign preceding the constant it is assumed to be positive. Embedded spaces, commas and non digit characters are not permitted between the digits of an integer constant.

There are three types of integers, **decimal, octal** and **hexadecimal.**

Valid examples of **decimal integer** constants are :

3543

-89022

0

+58

-332

The following are illegal constants since they make use of non digit characters, embedded spaces etc. :

23,487

$550

3 333

An **octal** constant is a combination of digits from 0 to 7, with a leading 0.

For eg.

044

0558

0

A sequence of digits preceded by Ox or OX is a **hexadecimal integer.** It can also include alphabets from A to F or a to f. As you have already studied, the letters A to F represent the numbers form 10 to 15. Some examples of valid hexadecimal integers :

0x450

0xA343

Ox

OxCDA

Octal and hexadecimal integers are very rarely used in programming. The largest integer value that can be stored is machine dependent. On 16 bit computers the range for integer constants is between -32,768 to + 32,767. On 32 bit processors the range is much larger. It is also possible to store larger integer constants by making use of qualifiers **u** (unsigned), **L** (long) and **UL** (unsigned long). We shall study these qualifiers later.

(ii) Real Constants :

Real constants are also called as **floating point** constants. When it is necessary to represent quantities which vary continuously like temperature, distance etc. it is required that the number has a fractional part eg. 234.567. This is the fractional form of a real constant. Thus when constructing a real constant it must have a decimal point. A real constant shall have atleast one digit. It can be either positive or negative with the default sign being positive. No commas, blank space or any other non digit characters are allowed when constructing real constants. Some valid examples of real constants are :

-0.987

35.89

1000.5434

+456.932

Another way of representing real constants is the **exponential form.** Exponential form is also known as scientific notation. When representing a number in its exponential form, the real constant is represented in two parts : the **mantissa** and the **exponent.** The part of the number appearing before the alphabet **e** is the mantissa and the part following e is the exponent. The general form of the number is thus :

mantissa e exponent

Thus while constructing a real constant in an exponential form :

- The mantissa and the exponent are separated by the letter e.

- The mantissa can be positive or negative with the default sign of the mantissa being positive.

- The exponent should have atleast one digit which is a positive or negative integer.

Some valid examples of exponential form of real constants are :

3.2e4 (e4 means multiply be 10^4)

4.8e-2 (e-2 means multiply by 10^{-2})

-43e+3 (e3 means multiply by 10^3)

(iii) Single character constants :

A single character constant is a **single character enclosed in a pair of single quotes.** It can either be a single alphabet, a single digit or a single special symbol. Some examples of character constants are :

'4' 'A' '^'

Note here that the character constant 4 is not the same as the number 4. Character constants have integer values which are known as their **ASCII** values.

Following are invalid character constants :

"pa"

"z"

"98"

(iv) String constants :

A string constant is a sequence of characters enclosed in double quotes. These characters may be letters, digits, special characters as well as blank spaces. eg.

"Good Morning"

"3456"

"*****"

are valid string constants in C.

It is important to remember that the single character constant eg. 'w is not equal to a character string constant "W". Also a single character string constant does not have an integer value as in the case of a character constant.

(v) Backslash character constants :

C supports some special backslash character constants which are used in output functions. Some of these backslash character constants are given below. Note that each one of them represents one character, although it actually consists of two characters. These character combinations are known as **escape sequences.**

Constant	**Meaning**
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\?'	question mark
'\\'	backslash
'\0'	null

2.4.3 Variables :

A variable is a data name used to store a data value. Variable names are names given to locations in the memory where different constants are stored. Unlike a constant which remains unchanged during the execution of a program, a variable can take different values at different times during execution.

A variable name is a combination of alphabets, digits or underscores. The first character however must be an alphabet. No commas, or blanks are allowed while constructing variables. The variable name should not be a keyword. Variables are case sensitive i.e. sum and SUM are two distinct variables. Some valid variable names are :Max_length age xyz avg a1 T1

2.4.1 & 2.4.2 & 2.4.3 Check Your Progress.

1. Write 1-2 lines about:

a) Backslash character constants

.....

.....

b) Identifiers

.....

.....

c) Keywords

.....

.....

d) What is the difference between a variable and a constant?

.....

.....

e) What are octal integers?

.....

.....

f) What is the difference between a single character constant and a string constant?

.....

.....

2. Match the following :

Column A	**Column B**
(i) Constant	a. Escape sequences
(ii) Identifier	b. Fixed value
(iii) Keyword	c. Octal constant
(iv) Real Constant	d. Floating point constant
(v) Backslash character constant	e. Reserved Word
	f. User defined

3. State whether the following are True or False

a) "Welcome" is a string constant.

b) 29.85 is an integer constant.

c) 1.2e+10 is a floating point constant.

d) You_1 is a valid identifier.

e) An identifier must begin with a character.

4. Are the following valid variable names in C?

a) _pq

b) Min_temp

c) Max temp

d) 1a

e) add1

2.5 INSTRUCTIONS IN C

We have seen the different types of constants, variables and keywords in the previous section. These are combined to form meaningful instructions and subsequently our program. The four basic types of instructions in C are:

Type Declaration Instructions :

These instructions are used to declare the **type** of the variables in a C program.

Input/Output Instructions :

These instructions perform the function of supplying input data and obtain the output results from the program.

Arithmetic Instructions:

These instructions are used to perform arithmetic on variables and constants.

Control Instructions :

The control instructions are used to control the sequence of execution of the various statements in a C program.

Of these, let us attempt to study the **type declaration** and **arithmetic instructions** in this section. The Input/Output instructions and control instructions shall be studied in the following chapters.

2.5.1 Type Declaration Instructions:

Type declaration instructions are used to declare the **type** of the variables in a C program. C supports a number of **data types.** These different data types allow the user to select the data type according to the needs of the application. Let us first see the data types supported by C and then the type declaration instructions.

The various data types supported by C are :

- Primary Data Types
- User defined Data Types
- Derived Data Types
- Empty data set

In this chapter, we shall discuss the primary data types. The derived data types include arrays, functions, structures and pointers and shall be dealt with subsequently.

(i) **Primary Data Types :**

The four fundamental or primary data types are :

Integer **(int)**

Character **(char)**

Floating point **(float)**

double precision floating point **(double)**

These primary data types themselves are of many types. eg. integers could be **long** or **short,** signed or unsigned etc. Here we take a look at the primary data types :

a) Integer (int) Data Type :

The range of values of the **int** data type is between -32,768 to 32,767. Integers are whole numbers. The range of values of integers is dependent upon the particular machine being used. Integers generally occupy two bytes of memory storage. Therefore for a 16-bit word length, the range of integer values is between -32,768 to 32,767. eg. 2809, -3888 , 0, 32, -18 are all valid integers.

Out of the two bytes used to store an integer, the sixteenth bit is used to store the sign (+ or -) of the integer. This sixteenth bit is 0 if the number is positive and 1 if the number is negative.

b) Floating Point (float) Data Type :

In order to declare real numbers we make use of the floating point data type. Floating point numbers are stored as 32 bits i.e. they occupy four bytes of memory, with 6 digits of precision (on 16 bit as well as 32 bit machines). Six bit precision means that there would be six digits after the decimal point. We use the **float** keyword to declare variables of floating point type. eg. 78.22, -9012.5988, 0.00988, 356.8 are examples of float data type.

c) Character Data Type :

A single character is defined as **char** type data. Characters are usually stored in 8 bits i.e. one byte of internal storage. The **signed** or **unsigned** qualifiers may be applied to **char.** By default, char is assumed to be **signed** and hence has a range of values from -128 to 127. On the other hand, declaring **unsigned** char causes the range of values to fall between 0 to 255, with all values being positive.

(ii) Type Declaration :

Let us now study the type declaration instructions of these primary data types. The type declaration instruction declares the type of the variable being used. Any variable being used in a program has to be declared first before using it. The type

declaration statements for all the variables appearing in the program are usually written at the beginning of the C program.

The syntax for declaring the variables is thus :

data type v1, v2, ...vn ;

where **data type** indicates the type and v1, v2, ..., vn indicate the names of the variables. If you are declaring multiple variables in the same type declaration, then they should be separated by commas. The type declaration statement itself should end with the semicolon.

Examples of declaring variables and their types :

int a;

char first;

float percent, avg;

Here in the first statement the variable a is declared to be of type **int,** percent and avg are real variables whose type is declared as **float,** first is a character variable whose type is declared as **char.** Note that multiple variables of the same type can be declared in the type declaration statement of a specific type. Thus if you want to declare a as **int** and b as float then both have to be declared separately as :

int a;

float b;

int a, b will declare both to be of type **int.**

(iii) Assigning values to variables :

Variables can be assigned values with the use of the assignment '=' operator. eg.

a = 12;

ch1 = 'D';

r1 = 13.85;

It is also possible to assign the value of an expression on the right to the variable on the left. eg;

c = a + b;

z = p/q * r;

p = p + 10;

Here the expressions to the right are evaluated first and then the resultant value is assigned to the variable on the left. It is important that before evaluating the expression on the right, all the variables used in the expression are assigned values.

Variables can also be assigned values at the time of their declaration. The following statements are valid in C :

int i = 10;

float pi = 3.14;

char ch1 = 'Q';

This process of giving initial values to variables is called **initialisation.** It is also possible to initialise multiple variables using multiple assignment operators as :

a = b = c = 1;

x = y = z = 100;

2.5.1 Check Your Progress.

1. Answer the following in one- two lines

a) What are the different types of instructions in C?

.....

.....

b) What are the primary data types in C?

.....

.....

2. Write in short about :

a) Integer Data Type

.....

.....

b) Character data Type.

.....

.....

3. State whether the following declarations are valid :

a) int j = "PQ";

b) p = 84.29 float;

c) int a, b, c;

d) int a = 10, b, c;

e) float p,q, int i;

2.5.2 Arithmetic Instructions :

The general syntax of an arithmetic expression in C is :

variable = expression;

This implies that an arithmetic instruction in C consists of a **variable name** on the left hand of the '=' sign, and a combination of variable names, constants and operators on the right hand side of the '=' sign. C can handle any complex mathematical expression. The expression on the right hand side of the '=' sign is evaluated and the result thus obtained is assigned to the variable name on the left of the '=' sign. (The constants and the variables together are called **operands.** These are operated upon by the arithmetic operators). All the variables which have been used in the expression on the right of the arithmetic instruction have to be assigned values before the evaluation of the expression.

Some examples of valid arithmetic instructions are :

c = a + b;

square = a x a;

x = axa + 2xb;

The variable to which the value is assigned after the evaluation of the expression has to be on the left. Also remember that each and every operator has to be explicitly written Thus the following are illegal statements :

p x q = a;

x + y-z = final;

p = qr;

a = bc(mn);

Arithmetic operations can be performed on **ints, floats** and **chars.** Thus the following is perfectly valid :

```
char a, b;
int c;
a = 'p';
b = 'm';
c = a + b;
```

Note here that although a and b are **char,** their ASCII values are added in the arithmetic expression. The ASCII chart for the various characters is given below for your reference :

Characters	**ASCII values**
0-9	48-57
A - Z	65-90
a-z	97-122
Special Symbols	0 -47, 58-64, 91 -96, 123-127

2.6 OPERATORS IN C

An **operator** is a symbol which tells the computer to perform certain mathematical or logical manipulations. The operators are used in mathematical and logical expressions.

Operators in C are classified as under:

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

Let us study each of these operators.

2.6.1 Arithmetic Operators :

2.6.1.1 The arithmetic operators in C are :

+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo Division

Hierarchy of Operations :

The precedence in which operations in an arithmetic statement are performed is called the hierarchy of operations. An arithmetic expression is evaluated from left to right according to priority of the operators. The hierarchy of operators is :

* / %	High priority
+ -	Low priority

= (Assignment) Last priority

Thus in an expression multiplication, division and modulo division get the highest priority, the addition and subtraction operators come next and the assignment operator gets the last priority.

eg. p = 10 + a * c/b;

Here * and / have same priority so first a * c will be calculated and then the product will be divided by b. Next is the priority for +. Therefore the value obtained is added to 10 and finally the assignment operator is used to assign the resultant value to the variable p. We can make use of a pair of parenthesis to change the priority of operations. If there are more than one set of parenthesis in a statement, the innermost parenthesis will be evaluated first, then the operations of the second innermost parenthesis will be performed and so on.

eg . val = a + b * c + d;

In this statement with the normal priority of operations the sequence of evaluation will be :

b * c will be performed first and its product will be added to a and the total sum then added to d. However, using a pair of parenthesis as follows :

(a + b) * (c + d) will first add a to b and c to d and then find the product of both the sums. Thus depending upon what sequence of operations is to be followed be careful to keep the priority of operations in mind and make correct use of parenthesis.

Equipped with this knowledge of C tokens and instructions, let us try to write simple programs in C. Note that we have made the use of the assignment operator '=' in all these programs to assign values to variables. Assignment operator is discussed in subsequent sections.

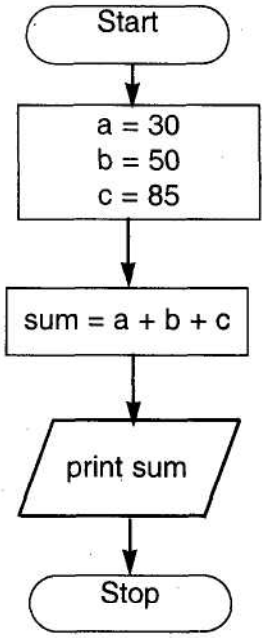
Example :

```
/*******************************************/

/* Calculate sum of three numbers */

main()

{
        int a, b, c, sum;
        a = 30;
        b = 50;
        c = 85;
        sum = a + b + c;
        printf("\nThe sum of a,b, c is :%d", sum);

}
```

The output of the program

The sum of a,b, c is : 165

The flowchart for the program to add three numbers is given below :



Flowchart for program to add three numbers a,b, c

This program defines three integer variables a, b and c and assigns them values 30, 50 and 80 respectively. The sum of these variables is evaluated and the result of the evaluation is assigned to the variable sum. The statement printf("\nThe sum of a,b,c is :%d", sum); prints the output i.e. the value of the variable sum. Note that **printf** is a function used to print the value of the variable on the screen. The '\n' backslash character constant as we already know is for a new line.

The general form of the **printf** statement is

printf("format string", <list of variables>);

At this point it is sufficient to understand how to make use of the format string to print **int, float** and **char** data types. The format string for these types is as under:

%f	for printing float or real values
%d	for printing integer values
%c	for printing character values

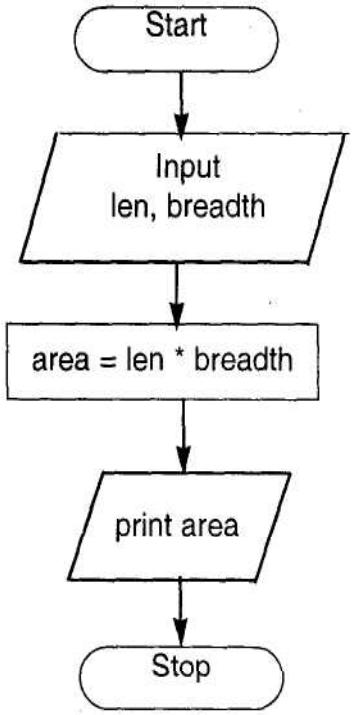
Example :

```
/*****************************************/
/* Calculate the area of a rectangle given its length and breadth*/
main()
{
        float len, breadth, area;
        len = 4.5;
        breadth = 2.8;
        area = len * breadth;
        printf("\nThe area is :%f", area);
}
```

The Output of the program will be:

The area is : 12.599999

Note here that for printing the value of the variable area, you use "%f", since %f is used to output floating values. Let us now modify the above program to output the length, breadth and area all three to see how multiple values can be output with a single printf statement



Flowchart for program to find area, given len and breadth

Example 3 :

```
/*****************************************/
/* Calculate the area of a rectangle given its length and breadth*/
main()
{
        float len, breadth, area;
        len = 4.5;
        breadth = 2.8;
        area = len * breadth;
        printf("\nLength = %f\nBreadth =%f\nArea = %f", len, breadth,area);
}
```

The output of the program will be :

Length = 4.500000

Breadth = 2.800000

Area = 12.599999

In order to output the values on separate lines, we make use of the '\n'. \n as we have already seen is a character constant to for new line.

All the above examples we have seen so far have assigned values to the variables used in the program itself. But in most practical applications the user will provide the input to the program variables. To supply input values from the keyboard we shall make use of the **scanf** function and rewrite the above program with **scanf** function to provide input to variables. The flowchart to accept user input and then compute the area is also shown.

Example :

```
/************************************/

/* Calculate the area of a rectangle given its length and breadth*/


main()
{
        float len, breadth, area;
        printf("\nEnter values for length and breadth\n");
        scanf("%f",&len) ;
        scanf("%f", &breadth);
        area = len * breadth;
        printf("Length = %f Breadth =%f Area = %f", len, breadth.area);
}
```

A sample Output:

```
Enter values for length and breadth
10.0
25.0
Length = 10.000000 Breadth = 25.000000 Area = 250.000000
```

The first **printf** statement will print the sentence :

Enter values for length and breadth

The **scanf** will read the values for the length first and then the breadth. When you input multiple values, separate them by using a white space like a space, tab or newline. Note that we have used the %f for reading floating point values. Area will then be calculated and printed on the screen. The **&** symbol is to be used before the variable name while reading data with the **scanf. &** is a pointer operator. Its meaning will be studied in later chapters. For the time being remember to precede every variable with & when using **scanf** to read the values of variables.

2.6 Check Your Progress

1. Choose the correct option.

a) The hierarchy decides :

(i) which operator is used first (ii) which operator is not to be used

(iii) which operator is the fastest (iv) none of the above

b) In the expression 3 x 5 + (8 + 3) / 5 which of the following will be executed first?

(i) 3x5 (ii) 8 + 3

(iii) /5 (iv) 3 x5 and /5 together

c) The expression, z = 10*14-8 evaluates to :

(i) 60 (ii) 132

(iii)-8 (iv)0

d) 25 % 3 evaluates to :

(i) 8.33 (ii) 8

(iii) 1 (iv) .33

e) 20/7 evaluates to :

(i) 2.85 (ii) 2

(iii) 0.8 (iv) 8

2. Which of the following are valid ? (Put V for valid and I for invalid).

a) z = p + q * r;

b) 14 = b-8;

c) a + b = c - d;

d) r = c/b;

e) m = 100/q;

3. Write programs in C for the following :

a) Input the maximum and minimum temperature recorded in a day and calculate the average temperature for the day.

b) Convert the distance entered in metres to kilometres.

c) Write a program in C to interchange two integer numbers a and b.

2.6.1.2 C Arithmetic:

Let us now see integer, float and mixed mode arithmetic.

(i) Integer arithmetic :

When the operands in a single arithmetic expression are integers or integer constants, the expression is an integer expression. The operation on these operands is called integer arithmetic. Integer arithmetic always yields an integer value.

eg. If p and q are declared as integer variables then assuming value of p = 10 and q = 4 the arithmetic will yield the results as follows :

p + q = 14

p - q = 6

p * q = 40

p / q = 2

(In this case the decimal part is truncated since the result is an integer value)

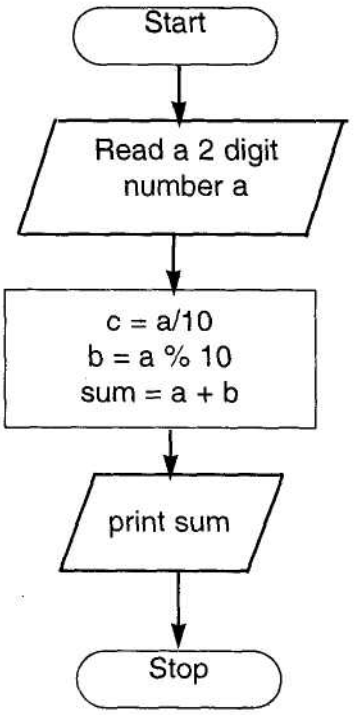
p % q = 5 (remainder obtained after carrying out division)

Also remember that a division like 4/7 evaluates to zero since 4 and 7 are both

integer constants and will return an integer value.

Let us make use of the above arithmetic to find the sum of digits of a 2 digit number. We shall make use of the division and modulo division operators in this example.

Example : Write a program to calculate the sum of the digits of a 2 digit number with the help of the flowchart given below.



Flowchart for program to add the digits of a 2 digit number

```
main()
{
        /"Calculate sum of digits of a two digit number */
        int a,b,c, sum;
        printf("\nEnter a two digit number:");
        scanf("%d", &a);
        c = a/10;
        b = a%10;
        sum = c + b;
        printf("\nThe sum of digits is : %d", sum);
}
```

A sample run of the program :

Enter a two digit number: 83

The sum of digits is : 11

Since the division is performed on **int** numbers a/10 will give the quotient and

a%10 will give the remainder. Thus we will be able to separate the digits of the number. These are stored in c and b and then added to find the sum of the digits.

(ii) Real Arithmetic :

Real arithmetic involves real operands which means that all operands in the expression are either real constants or real variables. Thus if p and q are real, and values of p and q are 10.0 and 4.0 respectively then,

p + q = 14.0

p - q = 6.0

p/q = 2.5

The modulo (%) operator cannot be used with real operands.

(iii) Mixed mode arithmetic :

In mixed **mode arithmetic** some operands are integers and others are real. Such an expression is called as mixed mode arithmetic expression. If any one of the operand is real, then the expression always yields a real value.

eg. if a is declared an integer variable and b is declared **float,** an operation on **a** and b will yield a real value. Thus,

- An arithmetic operation between an integer and integer will always yield an integer result
- An arithmetic operation between a real and real will always yield a real result
- An arithmetic operation between an integer and real will always yield a real result.

Let us further understand this with the help of the following table :

int a = 9;			**int** a = 9		
int b = 4;	**a/b**	1	**float** b = 4.0	**a/b**	1.75
float a = 9.0			**float** a = 9.0	**a/b**	1.75
float b = 4.0	**a/b**	1.75	**int** b = 4		

The following programs will demonstrate the arithmetic discussed so far:

Example : Calculate the simple interest using the formula :

simple interest= pnr/100, where p is the principal amount, n is the number of years and r is the rate of interest.

```
main()
{
    /*Program to calculate simple interest */
    int n;
    float p, r, interest;
    printf("\nEnter principal amount:");
    scanf("%f",&p);
    printf("\nEnter number of years :");
    scanf("%d",&n);
    printf("\nEnter rate of interest:");
    scanf("%f",&r);
    interest = p * n * r/100;
    printf("\nThe simple interest is : %f", interest);
}
```

A sample output:

Enter principal amount: 1000.0

Enter number of years : 2

Enter rate of interest :10.0

The simple interest is : 200.000000

Example 2 : Write a program to convert temperature from Fahrenheit to Celsius :

```
main()
{
        /*Temperature Conversion Program */
        float faren, cel;
        printf("\nEnter temperature in Fahrenheit:");
        scanf("%f", &faren);
        cel = (faren - 32.0)/ 9*5;
        printf("\nThe temperature in Celsius is : %f", cel);
}
```

A sample output:

Enter temperature in Fahrenheit: 100.0

The temperature in Celsius is :37.777779

In the above program follow carefully the hierarchy of operations and the use of parenthesis to achieve the desired result. Modify the above program to input temperature in Celsius and convert it to Farenheit.

2.6.1 Check Your Progress.

1. What will be the output of the following ?

a)
```
int a,b,c;
a = 4.3;
b = 2.1;
c = a*b;
printf("%d", c);
```
.....

.....

b)
```
int a = 10;
float b = 24.5, c;
c = a + b;
printf("%f", c);
```
.....

.....

c)
```
int a = 45, b, c;
b = a/10;
c=a%10;
printf("%d%d", b,c);
```
.....

.....

2. Answer in 1-2 sentences :

a) What is meant by integer arithmetic?

.....

.....

b) Describe briefly mixed mode arithmetic.

.....

.....

2.6.2 Relational Operators :

When we wish to compare any two values, we make use of **relational operators.** eg. when we wish to compare prices, age etc. we make use of relational operators. An expression which uses a relational operator is a **relational expression.** The value of a relational expression is either zero or one. If it is one, then the specified relation is **true,** if it is 0, then the relation is **false.** eg.

2<10	is true	hence will return a value equal to 1
2>10	is false	hence will return a value equal to 0

The relational operators in C are as under:

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

Arithmetic expression can be used on either side of a relational operator.eg.

p-q>a + b

In such a situation, both the arithmetic expressions are evaluated first and then the results so obtained are compared. Thus arithmetic operators have a higher priority as compared to relational operators. Extensive use of relational operators is made in decision statements which will be discussed in later chapters.

2.6.3 Logical operators :

Logical operators are used when we want to test more than one condition and then make decisions. C has the following logical operators :

&&	logical AND
||	logical OR
!	logical NOT

eg. if a < b && p+ q == 40

The above expression combines two relational expressions by the logical operator. Such an expression is called as a **logical expression** or a **compound relational expression.** The logical expression also yields a value zero or one. Thus, the above expression will yield a value 1 only when a is less than b and p + q is equal to 40 i.e when both the conditions are true

Truth table for logical AND (&&)

Exp1	Exp2	Value of Exp1 && Exp2
Non-zero	Non-zero	1
Non-zero	0	0
0	Non-zero	0
0	0	0

Truth table for logical OR (||)

Exp1	Exp2	Value of Exp1||Exp2
Non-zero	Non-zero	1
Non-zero	0	1
0	Non-zero	1
0	0	0

We shall make use of these logical operators when we study decision control statements.

2.6.4 Assignment Operators :

As we have already seen, assignment operators are used to assign the result of an expression to a variable. The assignment operator in C is '='.

2.6.2 2.6.3 2.6.4 Check Your Progress.

1. Fill in the blanks.

a) Relational operators have a ..... priority as compared to arithmetic operators.

b) The logical operators in C are.....,.....and.....

c) The relational operator not equal to is written as.....

d) .....operators are used to compare values.

e) If exp1 and exp2 both are non zero a||b will return.....

2. Write in short about relational operators in C.

.....

.....

2.6.5 Increment and Decrement Operators :

C has ++ as the **increment** and — as the **decrement** operator. The increment operator adds 1 to the operand and the decrement operator subtracts 1 from the operand. Both are **unary** operators.

Form of the increment operator:

++a ; or a++;

which is equivalent to

a = a + 1;

Form of the decrement operator:

--a; or a--

which is equivalent to

a = a -1;

++a and a++ behave differently when they are used in expressions on the right side of an assignment. Consider the following :

a = 5;

b = ++a;

Here the variable b will be assigned the value 6.

However

a = 5;

b = a++;

will assign the value 5 to b first and then the value of a will be incremented.

Thus the **prefix operator** will first increment the operand by 1 and then assign the value to the variable on the left, whereas the **postfix operator** will first assign the value to the variable on the left and then increment the value of the operand by 1. Similarly in the case of the decrement operator, the prefix operator will first decrement the value of the operand by 1 and then assign it to the variable on the left and the postfix operator will be assign the value to the variable on the left and then decrement the value of the operand by 1.

Example : Let us write a C program to demonstrate the use of the increment and decrement operators :

```
main()
{
        /* Program to illustrate the use of ++ and —) */
        int a, b;
        a = 10;
        b = 20;
        printf("The value of a : %d", a);
        printf("The value of b : %d, b);
        a++;
        b++;
        printf("The incremented value of a : %d", a);
        printf("The incremented value of b : %d, b);
        a--;
        b--;
        printf("The decremented value of a : %d", a);
        printf("The decremented value of b : %d, b);
}
```

The output of the program will be:

```
The value of a : 10
The value of b : 20
The incremented value of a : 11
The incremented value of b : 21
The decremented value of a : 10
The decremented value of b : 20
```

2.6.6 Conditional Operator:

C has one conditional operator. The syntax of the conditional operator in C is

exp1 ? exp2 : exp3;

where exp1, exp2 and exp3 are expressions. Here, exp1 is evaluated first. If it is true (non zero), then exp2 is evaluated and it becomes the value of the expression. On the other hand if exp1 is false, then exp3 is evaluated and its value becomes the value of the expression.

Thus if

a = 10;

b = 5;

z = (a >b)? a : b;

Here

exp1 is (a > b)

exp2 is a and

exp3 is b

First (a > b) is evaluated. Since in this case, a > b is false, exp3 is evaluated. Therefore z is assigned the value of b.

2.6.5 & 2.6.6 Check Your Progress.

1. What values will the variables have in the following:

a) z = (a < b) ? a+b : a - b for a = 5 and b = 10 ?

z = ...

b) i = 10;

i++;

a = i + 10;

a = ...

i = _____

c) z = (i + 10 < j) ? 100:10 for i = 10, j = 5

z = ...

2.6.7 Bitwise Operators :

Bitwise operators are used for manipulation of data at the **bit** (binary digit) level. Bitwise operators cannot be applied for **float** or **double.** The use of bitwise operators is for testing the bits, or shifting them to the right or left. The bitwise operators are as follows :

Operator	Meaning
&	Bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
<<	shift left
>>	shift right
~	One's complement

2.6.8 Special Operators :

Special operators in C include the comma (,), sizeof, pointer operators (& and *) and member selection operators (. and ->). We shall discuss only the , and sizeof operators here.

The **comma** (,) operator is used for linking related expressions together. When such expressions are linked by the comma operator, they are evaluated from left to right and the value of the rightmost expression finally becomes the value of the combined expression. Thus,

z = (a = 10, b = 2, a-b);

will first assign the value 10 to a, then the value 2 to b. It will then evaluate a - b and assign the value of 8 to the variable z.

The **sizeof** operator is used to obtain the number of bytes that the operand occupies in memory. The operand can be a variable, constant or a data type qualifier.

eg. z = sizeof(average);

This operator is normally used for determination of the lengths of arrays and structures. It is also used to allocate memory dynamically to variables during the execution of the program.

2.7 DEFINING SYMBOLIC CONSTANTS

There are numerous situations in programs where we are required to use certain values repeatedly. eg. the value 3.14 is such a constant which we use in many geometrical problems. While using such constants, it may so happen that we wish to change the value for some reason or other. Under such conditions if we are making use of such a value in our program severel times, we will have to modify this value at each and every occurrence of it in the program. C provides a facility of assigning such constants to a **symbolic name.** These constant values are assigned to symbolic names, usually at the beginning of the program. Once these values have been assigned symbolic names at the beginning, they are automatically substituted at the appropriate places wherever the symbolic name appears.

The symbolic constants are defined as follows :

#define symbolic_name value of the constant

eg.

#define PI 3.14

#defineMIN 0

Note that symbolic constants are not variables, so they do not appear in declarations.

- Usually, symbolic constants are written in upper case to differentiate them from the variables.

- Symbolic constants can appear anywhere in a program, however they have to be defined before they are referenced.

- There should be no space between # and define.

- There should be no semicolon after defining a symbolic constant

- Symbolic constants are not declared for their data types. The data type of a symbolic constant depends upon its value.

2.6.7, 2.6.8 2.7 Check Your Progress.

Answer in 1-2 sentences :

a) What are Bitwise operators?

.....

.....

b) What is the comma operator?

.....

.....

c) What is the sizeof() operator?

.....

.....

d) What is meant by symbolic constants?

.....

.....

2.8 SUMMARY

In this chapter we studied different features of C programming Language.

- C is simple to learn and easy to understand
- C is a very robust programming language.
- C has rich set of built in functions.
- C is highly portable
- Suitable for system and application software both.

The C character set contains -

- Letters or Alphabets - A - Z & a - z
- Digits 0 - 9 and its combination
- Special Symbols - { } , ( ) , % $ etc
- White Spaces : blanks , tabs, new line , form feed and carriage return.

C Tokens

- Keywords - are words whose meanings have already been defined and these meaning cannot be changes. Keywords are also called as reserves words. There are 32 keywords in C
- Identifiers - are the name given to variables functions and array. They are user defined.
- Constants - Fixed values which does not change during the execution of program.

- Variables - are the data names used to store data values.
- Operators - Arithmetic operators (+ , - , * , / %)
  - Relational operators (< , > , <=, >=, = =, !=)
  - Logical Operators (AND, OR, NOT)
  - Assignment operators (=)
  - Increment and Decrement Operators (++a, a++, --a, a--)
  - Conditional Operators (exp1 ? exp2 : exp3)
  - Bitwise Operators(& - Bitwise AND, ! - Bitwise OR, ^ Bitwise Exclusive OR , << shift left , >> Shift right, ~ one's complement)

2.9 CHECK YOUR PROGRESS - ANSWERS

2.1 & 2.2

1. a) Dennis Ritchie

b) 32

c) Comment

d) semicolon

2. a) The C programming language was developed at AT&T's Bell Laboratories in USA in 1972 by Dennis Ritchie. C is a very robust programming language and has a rich set of built in functions. C is simple to learn and easy to use. C language is suitable for writing both system software and application programs and business packages. The C compiler has the capabilities of the machine language and the features of a high level language. C is now one of the most popular high level programming language and is running under many operating systems. C language is highly portable i.e. C programs written for one computer can be run on another computer with very little or no modifications.

b) Comment lines are not executable statements and are written between /* and */. They are ignored by the compiler. Comments in our programs will help users and other programmers in understanding the programs easily. Debugging and testing also becomes easy when helpful comments are inserted at appropriate places in our programs. A program can have any number of comments and they can be inserted at any place in a program. They can be more than one line.

c) main() is a special function in C. This is the line at which the execution of our program begins. Every program should have this function main(). There is a pair of parenthesis after main(). The open brace bracket '{' indicates the beginning of the function main() and the close brace bracket '}' in the last line marks the end of the function. The statements which are included within these braces make up the body of our function main() and form the executable code of our program.

2.4.1, 2.4.2 & 2.4.3

1 a) Backslash character constants: C supports some special backslash character constants which are used in output functions. Each one of them represents one character, although it actually consists of two characters. These character combinations are known as **escape sequences.** eg.'\f' - form feed, '\n' - new line

b) Identifiers are names given to variables, functions and arrays. These are the names which are user defined. They are made up by a combination of letters and digits. Normally an identifier should not be more than 8 characters long. The use of underscore is also permitted in identifiers. However, it is imperative that identifiers

should begin with a letter. eg. max, in_val.

c) Keywords are words whose meanings have already been defined and these meanings cannot be changed. Keywords are also called as reserved words. Keywords should not be used as variable names (though some compilers allow you to construct variable names like keywords). All the keywords must be written in lowercase. There are 32 keywords in C. eg. int, float.

d) A constant in C is a fixed value which does not change during the execution of a program. C supports the constants viz.Numeric Constants which are further classified as Integer and Real and Character constants which are further classified as Single Character and String constant. A variable on the other hand is a data name used to store a data value. Variable names are names given to locations in the memory where different constants are stored. A variable can take different values at different times during execution.

e) Integer constants are of decimal, octal and hexadecimal types. An octal constant is a combination of digits from 0 to 7, with a leading 0. For eg. 044, 0558

f) A single character constant is a single character enclosed in a pair of single quotes. It can either be a single alphabet, a single digit or a single special symbol. eg. 'a', '}', '5'. Character constants have integer values which are known as their **ASCII** values. On the other hand a string constant is a sequence of characters enclosed in double quotes. These characters may be letters, digits, special characters as well as blank spaces. eg. "abc", "example23" etc.

2 (i) -b
(ii) -f
(iii) -e
(iv) -d
(v) -a

3 a) True
b) False
c) True
d) True
e) True

4 a) Invalid
b) Valid
c) Invalid
d) Invalid
e) Valid

2.5.1

1 a) The four basic types of instructions in C are:

Type Declaration Instructions: used to declare the type of the variables. Input/ Output Instructions: These instructions perform the function of supplying input data and obtain the output results from the program.

Arithmetic Instructions: used to perform arithmetic on variables and constants. Control Instructions: used to control the sequence of execution of the various statements in a C program.

b) The four fundamental or primary data types are : Integer (int), Character (char), Floating point (float), double precision floating point (double). These primary data types themselves are of many types.

2 a) Integer (int) Data Type: This is one of the primary data types in C. Integers are whole numbers. The range of values of integers is dependent upon the particular

machine being used. Integers generally occupy two bytes of memory storage. Therefore for a 16-bit word length, the range of integer values is between -32,768 to 32,767. Out of the two bytes used to store an integer, the sixteenth bit is used to store the sign (+ or -) of the integer. This sixteenth bit is 0 if the number is positive and 1 if the number is negative.

b) Character Data Type: This is one of the primary data types in C. A single character is defined as char type data. Characters are usually stored in 8 bits i.e. one byte of internal storage. The signed or unsigned qualifiers may be applied to char. By default, char is assumed to be signed and hence has a range of values from -128 to 127. On the other hand, declaring unsigned char causes the range of values to fall between 0 to 255, with all values being positive.

3 a) Invalid
b) Invalid
c) Valid
d) Valid
e) Invalid

4. The type declaration instruction declares the type of the variable being used. Any variable being used in a program has to be declared first before using it. The syntax for declaring the variables for their types is:

data type v1, v2, ...vn ;

where data type indicates the type and v1, v2, ..., vn indicate the names of the variables. When declaring multiple variables in the same type declaration, they should be separated by commas. The type declaration statement should end with the semicolon.

2.6

1 a) i) which operator is used first
b) ii) 8+3
c) ii) 132
d) iii) 1
e) ii) 2

2 a) V
b) I
c) I
d) V
e) V

3 a)
```
main()
{
    float max, min, avg;
    printf("Enter maximum temperature:");
    scanf("%f'', &max);
    printf("Enter minimum temperature :");
    scanf("%f'', &min);
    avg = (max + min)/2;
    printf("\nThe average temperature of the day : %f", avg);
}
```

b)
```
main()
{
    float metres, km;
```

```
        printf("Enter distance in metres :");
        scanf("%f", &metres);
        km = metres/1000;
        printf("Distance in kilometres = %f", km);
    }
```

c)
```
    main()
    {
        int a,b, c;
        printf("Enter value of a :");
        scanf("%d", &a);
        printf("Enter value of b :");
        scanf("%d", &b);
        c =a;
        a = b;
        b = c;
        printf("Value of a = %d\nValue of b= %d", a, b);
    }
```

2.6.1

1 a) 8

b) 34.500000

c) 4 5

2 a) When the operands in a single arithmetic expression are integers or integer constants, then expression is an integer expression. The operation on these operands is called integer arithmetic. Integer arithmetic always yields an integer value. In case of integer division the decimal part is truncated since the result is an integer value.

b) In mixed mode arithmetic some operands are integers and others are real. Such an expression is called as mixed mode arithmetic expression. If any one of the operand is real, then the expression always yields a real value. An arithmetic operation between an integer and integer will always yield an integer result. An arithmetic operation between a real and real will always yield a real result and an arithmetic operation between an integer and real will always yield a real result.

2.6.2, 2.6.3 & 2.6.4

1 a) lower

b) AND, OR, NOT

c) !=

d) Relational

e) 1

2. Relational operators are used when we wish to compare any two values. An expression which uses a relational operator is called a relational expression. The value of a relational expression is either zero or one. If it is one, then the specified relation is true, if it is 0, then the relation is false. The relational operators in C are :<, <= , >, >=, ==, !=.. An arithmetic expression can be used on either side of a relational operator. Here both the arithmetic expressions are evaluated first and then the results so obtained are compared. Thus arithmetic operators have a higher priority as compared to relational operators.

2.6.5 & 2.6.6

1. a) 15

b) 21 11

c) 10

2.6.7 & 2.6.8 & 2.6.9

1. a) Bitwise operators are used for manipulation of data at the bit (binary digit) level. Bitwise operators cannot be applied for float or double. The use of bitwise operators is for testing the bits, or shifting them to the right or left. Some of the bitwise operators in C are as & - Bitwise AND, ! – Bitwise OR.

b) The comma (,) operator is a special operator in C and is used for linking related expressions together. When such expressions are linked by the comma operator, they are evaluated from left to right and the value of the rightmost expression finally becomes the value of the combined expression.

c) The sizeof operator is a special operator in C and is used to obtain the number of bytes that the operand occupies in memory. The operand can be a variable, constant or a data type qualifier. This operator is normally used for determination of the lengths of arrays and structures. It is also used to allocate memory dynamically to variables during the execution of the program.

2. C provides a facility of assigning constants to a symbolic name. These constant values are assigned to symbolic names, usually at the beginning of the program. Once these values have been assigned symbolic names at the beginning, they are automatically substituted at the appropriate places wherever the symbolic name appears. The symbolic constants are defined as follows :

#define symbolic_name value of the constant

Symbolic constants are not variables, so they do not appear in declarations. Usually, symbolic constants are written in upper case to differentiate them from the variables. They can appear anywhere in a program, however they have to be defined before they are referenced. Symbolic constants are not declared for their data types. The data type of a symbolic constant depends upon its value.

2.10 QUESTIONS FOR SELF-STUDY

1. What are the primary data types in C? Discuss them.

2. Write detailed notes on :

a) Relational and Logical Operators

b) Constants in C

c) C Arithmetic

d) Symbolic Constants

e) Type declaration of data type.

3. Answer the following in 7- 8 lines each

a) What is the conditional operator in C ?

b) What are keywords and identifiers?

c) Write a short note on the features of the C language.

4. Write the following programs in C.

a) Input the salary of a person. Calculate the dearness allowance as 15% of the salary and house rent allowance as 5% of the salary. After this determine the total salary.

b) Reverse the digits of a 4 digit number.

c) Find the sum of digits of a 4 digit number

d) Convert a distance entered in metres to kilometres.

e) Find the average of marks of 5 subjects

2.11 SUGGESTED READINGS

Programming in ANSI C : Balguruswamy

Exploring C : Yashwant Kanitkar

C for Beginners : Madhusudan Mothe

CHAPTER 3

INPUT/OUTPUT

3.0 OBJECTIVES

Friends after studying the chapter you will be able to

- state what is formatted and unformatted Input/Output
- explain the catagories of input/output
- discuss console input/output functions available in the standard C library
- make use of these input/output functions in all subsequent programs

3.1 INTRODUCTION

As we already know, programs take some data as input, process this data and display the results as output. In the previous chapter, we have seen how to supply input data to the C program, with the help of **scanf.** scanf reads the data from the terminal. Similarly, we have seen the use of **printf** to output the results on the console. Thus **scanf** and **printf** are functions used to input and output data from the C program. Most high level languages have built in input/output statements. C however, does not have any built in input/output statements as a part of its grammar. Therefore, all input and output operations are to be carried out with the help of functions like **printf** and **scanf.** Many functions have now become standard for input/output in C. These functions are known as the standard I/O library. Most of the standard Input/Output functions are included in the <stdio.h> (standard input-output header file). Therefore, for effecting input/output functions, your C program must include the statement

#include <stdio.h>

This statement will tell the compiler to search for this file, and place its contents at this point in the program. The contents of the header file will now become a part of your program. It is to be noted however, that some functions like **printf** and **scanf** do not require the inclusion of the standard input/output header file.

3.2 CATAGORIES OF INPUT/OUTPUT FUNCTIONS

The input/output functions available in the standard input/output file stdio.h can be classified into three broad catagories :

Console Input/Output Functions :

In this category, input is supplied from the keyboard and output displayed on the video display unit.

Disk Input/Output Functions :

These functions are useful to perform input/output operations on floppy

disks or hard disks i.e. secondary storage devices.

Port Input/Output Functions :

These functions are used to perform I/O operations on ports.

Out of these three catagories, console input/output form a part of this chapter. File input/output is discussed in separate chapter “File management”. Port Input/Output functions do not form a part of this study material.

3.1 & 3.2 Check Your Progress.

1. Fill in the blanks :

a) The function used to input data is .....

b) The function used to output data is .....

c) Most of the standard input/output operations are included in the .....file.

2. Answer in 1- 2 sentences :

a) How are input/output carried out in C?

.....

.....

b) What is meant by console input/output functions ?

.....

.....

3.3 CONSOLE INPUT/OUTPUT FUNCTIONS

Formatted and Unformatted Input/Output:

Input and Output can be read and printed in unformatted and formatted forms. Formatted input/output allows us to read the input or display the output to be formatted as per our requirements. Formatting implies specifying where you want the output to appear on the screen, how many spaces do you wish to have between the various outputs, the number of decimal places after the decimal point in case of float data etc. Thus we input or output data which has been arranged in a particular format.

The simplest form of input/output functions is whereby you supply information with the help of the keyboard and write it to the standard output device (the screen.) Console Input/Output functions are further classified as formatted and unformatted console input/output.

3.3.1 Unformatted Console Input/Output Functions :

(a) Character Input/Output:

Character input/output functions are useful for reading characters from the keyboard and writing single characters to the monitor.

Reading a character:

In order to read a single character we use the **getch(), getche()** and **getchar()** function. These functions return the character that has been most recently typed in. **getch()** returns the character which is typed without echoing it on the screen. **getche()** echoes (displays) the character that you typed on the screen. **getchar()** works similar

to **getche()** but requires the user to hit the Enter key after the character is typed in.

When you use these function in the program you are able to input a single character from the console. The following program will illustrate :

Example : To illustrate the use of **getch(), getche()**

```
main()
{
        char ch1;
        printf("\n Enter a character:");
        getch(); /*getch() will not echo the character on the screen*/
        printf("\nEnter a character:");
        ch1 = getche(); /* will echo the character on the screen */
}
```

A sample output

```
Enter a character:
Entera character: A
```

The general form of the **getchar()** is :

variable_name = getchar();

Here variable name is a valid variable name. Its type declared has to be **char.** When this statement is encountered, the computer system will wait for the user to type any keyboard character and then will assign this character as the value of the **getchar()** function. The character value of **getchar()** will be assigned to the variable_name specified on the left. eg.

```
char ch1;
ch1 = getchar();
```

Here variable ch1 is defined to be of type **char.** With the **getchar()** function, a character is read from the keyboard and assigned to the variable ch1.

The **getchar()** function echoes the character that you have typed on the screen. This means that the character you typed will be displayed on the screen. You are required to press the Enter key on the keyboard after you type the required character.

Writing a character:

We make use of the **putch()** and **putchar()** function for writing characters on the terminal. The general form of the **putchar()** function is

putchar(variable_name);

where variable_name is a variable of type **char** which contains a character. This function will display the character contained in variable_name on the screen.

eg.

```
char ch1;
ch1 = 'A';
putchar(ch1);
```

This set of statements will display the character A on the screen.

Let us illustrate the use of character input output in programs with the help of the following programs.

Example : Character input/output

```
/*Program to illustrate the use of getche() and putch()*/
main()
```

```
{
        char ch1;
        printf("\nInput a character:");
        ch1 = getche();      /* Read a character*/
        putch(ch1);
}
```

A sample output of the program :

```
Input a character: q
q
```

The **getchar()** function reads a character into ch1 and the **putchar()** outputs it on the terminal.

Example : Use of **getchar()** and **putchar()**

```
#include "stdarg.h"
include "sdtio.h"
main()
{
        char ch1, ch2;
        printf("\nEnter a character:);
        ch1 = getchar();
        printf(''\nEnter another character:");
        ch2 = getchar();
        putchar(ch1);
        putch(ch2);
        putch("A");
        putchar("Z");
}
```

A sample output:

```
Enter a character :P
Enter another character: O
POAZ
```

There are a number of character functions supported by C. These are contained in the file <ctype.h>. Therefore, when you want to make use of these functions you will have to include this file in your program as :

```
#include <ctype.h>
```

eg. **isalpha(ch1)**: This function checks whether the character ch1 is an alpabet.If it is, the function assumes a non-zero (TRUE) value, else it assumes 0 (FALSE). The character test functions supported by C are :

Function	**Test**
isalnum(ch1)	to check whether ch1 is alphanumeric
isalpha(ch1)	to check whether ch1 is an alphabet
isdigit(ch1)	to check whether ch1 is a digit
islower(ch1)	to check whether ch1 is a lowercase letter
isprint(ch1)	to check whether ch1 is a printable character
ispunct(ch1)	to check whether ch1 is a punctuation mark
isspace(ch1)	to check whether ch1 is a whitespace character
isupper(ch1)	to check whether ch1 is an uppercase letter

In each of these functions, the functions returns a nonzero (TRUE) value if the condition evaluates to true else it returns 0. We shall study the use of these functions when we study conditional statements.

3.3.1 Check Your Progress.

1. Match the following :

	Column A	Column B
a)	putch()	(i) checks whether character is in uppercase
b)	getche()	(ii) does not echo character on screen
c)	isupper()	(iii) echoes the character on the screen
d)	getch()	(iv) Function to output characters

2. Answer in 1-2 sentences :

a) What is meant by formatted input/output ?

.....

.....

b) What is the difference between **getche()** and **getchar()?**

.....

.....

3.3.2 Formatted Console Input/Output:

After having dealt with unformatted console input/output let us study formatted input/output. Formatted functions facilitate supplying input and providing output in fixed formats.

Formatted Input/Output:

In order to input data we have seen a limited use of the **scanf** function. In this section let us see this function in detail. The general form of the **scanf** is :

scanf("format string", list of variables);

The format string will specify the field format for the list of variables. Note the comma between the format string and the list of variables.

The format string can contain

- format specifications which consist of the **conversion specification** (%) followed by the data type character (which specifies the type of the data) and a number (optional) which will specify the field width. **%d** and **%f** are examples of conversion characters which we have already used to read **int** and **float** data types respectively. The conversions characters for various data types are summarised in the following table:

Data Type		Conversion Character
Integer	short signed	%d or%i
	short unsigned	%u
	long signed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
character	signed	%c
	unsigned	%c
string		%s

Note: Strings are not a part of this chapter and will be covered subsequently. Also the different variations of integer data types like long and short, of real data type like float and characters shall be studied in the chapter Data Types and Storage Classes. These are included here for the purpose of illustrating the complete table.

Additional specifiers which are optional in conversion specifications include :

Specifier	Description
w	specifies the field width
.	decimal point which is used to separate the field width

- escape sequences (blanks, newline character, tabs). Escape sequences as we have already seen begin with a '\' sign.

In order to output data we have made a limited use of the **printf** function in the previous chapter. **printf** helps us to produce outputs which can be formatted and therefore become understandable and easy to use. **printf** can be used to align and space the output on the screen. The general form of the **printf** is **:**

printf("format string", list of variables);

The format string contains :

- characters which will be printed on the screen as they are

- format specifications which begin with the % sign (which have been shown in the above table)

- escape sequences like \n, \t, \b

The control string indicates the number of variables and their types which follow in the list of variables. The variables in the variable list should match in number, order and type with the format string. eg. if the **printf** statement is as follows :

printf("%d%f%d",a,b,c);

then there have to be three data items as specified in the format string, where the first data item should be of type **int,** the second of type **float** and the third of type **int.** Thus they match in number, type and order.

Let us see how to use the **scanf** and **printf** functions to read **int**, **float** and **char.**

3.3.2.1 scanf for Integers :

The general form of the **scanf** function to input integers is

***%** w d*

where % is the symbol which indicates that a conversion sequence follows, **w** is the field width specified for the integer (this is optional) and **d** indicates that the data type to be read is in the integer mode.

eg. scanf("%3d %2d", &n1, &n2);

Here the function will associate a field width of 3 with n1 and a field width of 2 with n2. When you supply the input

234 48

the value 234 will be assigned to n1 and 48 to n2. Here you have to take care while specifying the field width. eg. if you supply information as :

48942 32

then n1 will be assigned a value 489 (since its field width is defined as 3) and n2 will be assigned the value 42. The number 32 will not get assigned to n2. **scanf** terminates reading of a value as soon as the number of characters specified by the field width is reached. Therefore, to avoid errors, we can eliminate the field width specifications and simply write the **scanf** function as :

scanf(“%d %d”, &n1, &n2);

When you are inputting multiple data items with the use of the **scanf,** you must separate them by spaces, tabs or newlines. In case you enter a float value for an int data type, **scanf** may strip the fractional part and it may even skip reading further input.

printf for integers :

The format specification for printing an integer number is :

% w d

You already know that %d is the conversion character for integers and w indicates . the field width. It is important to note here, that if a number is greater than the width specified , it will be printed in full. The field width specification will be overriden. When the number is printed, it is printed in a right justified manner. This means that there will be leading blanks if required. eg. As shown below, in the first case no field width is specified, in the second case a field width of 6 is specified and since the number is only 4 digits two leading blanks will appear. In the last case, field width specified is smaller than the actual number, hence it is overridden and the number is printed fully.

Format	Output
printf(“%d”, 1000);	1 0 0 0
printf(“%6d”, 1000);	1 0 0 0
printf(“%2d”, 1000);	1 0 0 0

The following additional formatting features are available for **int**:

- If you want your output to be left justified you can put a minus “-” sign directly after the % sign.

- You can also pad the leading blanks with 0s by placing a 0 before the field width specifier and after the % sign.eg.

Format	Output
printf(“%-6d”,1000);	1 0 0 0
printf(“%06d”,1000);	0 0 1 0 0 0

Let us write a program to illustrate the use of **scanf** and **printf** for integer types

Example : To make use of **scanf** and **printf** for integers

```
main()
{
        int a;
        printf(“\nEnter value for a:”);
        scanf(“%d”, &a);
        printf(“\nWithout specifying width :%d”, a);
        printf(“\nTo demonstrate right justification :%8d”,a);
        printf(“\nTo demonstrate left justification :%-6d”, a);
        printf(“\nTo demonstrate leading zeroes :%08d”,a);
}
```

A sample run of the program :

Enter value for a :3680

Without specifying width :3680

To demonstrate right justification : 3680

To demonstrate left justification :3680

To demonstrate leading zeroes :00003680

3.3.2.2 Input/Output of Real numbers : Let us now study how to make use of **scanf** and **printf** for formatted input and output of real numbers.

scanf for real numbers :

scanf uses %f specification for both, decimal point and exponential notation. To read two numbers a and b both which have been declared type **float** we can use the following method:

```
float a, b;
scanf("%f %f", &a, &b);
```

Values for variables a and b can be input in decimal or exponential notation as :

876.323 876.323E-1

then the above statement will assign the value 876.323 to a and 87.6323 to b.

Example : To use **scanf** to read float numbers :

```
main()
{
    float a,b;
printf("\nEnter values for a in decimal form and b in exponential form
:\n");
    scanf("%f%e", &a, &b);
    printf("\nValue of a:%f\nValue of b :%f", a, b);
}
```

A sample output:

Enter values for a in decimal form and b in exponential form :

367.9808 1.34e5

Value of a : 367.980800

Value of b : 134000.000000

printf for real numbers :

In the above example we have used **printf** to output real numbers in decimal notation. Also it prints both the numbers with a precision of 6. In reality, we can use **printf** to output real numbers in both decimal notation and exponential notation.

In order to print using the decimal notation the format specification is :

%w.pf

w is an integer which indicates the minimum number of positions to be used to display the value, integer p indicates the number of digits to be displayed after the decimal point (this is also known as precision). The value is rounded to p digits after the decimal point when it is printed. The value is printed right justified as in the case of **int.** The default precision for float is 6.

The format specification for exponential notation is :

%w.pe

The integer p specifies the precision of the number of digits after the point. The default precision is 6. The field width w should be large enough to satisfy the condition

w >= p + 7

The value is printed right justified in the field width w. Leading blanks will appear if necessary. As in the case of **int,** you can also make use of 0 or - for printing leading zeroes or left justification respectively.

Let us examine the output of the number a = 108.244 using different format specifications :

Example : To print a float using various format specifications :

```
main()
{
        float a = 108.244;
        printf("\nThe normal form : %f", a);
        printf("\nWith format specifier :%10.4f", a);
        printf("\nLeft justification :%10.4f", a);
        printf("\nLeading zeroes : %010.4f",a);
        printf("\nExponential notation : %10.2e",a);
}
```

The output of the program :

The normal form	:	108.244003
With format specifier	:	108.2440
Left justification	:	108.2440
Leading zeroes	:	00108.2440
Exponential notation	:	1.1e+02

3.3.2.1 & 3.3.2.2 Check Your Progress.

1. Fill in the blanks :

a) The conversion specification for integers is ..... and for float is .....

b) To print the output in left justified format we make use of the ..... .

c) Leading zeroes can be printed using .....

d) The w specifier is used in the specification to specify .....

e) To separate multiple data items input through scanf you make use of....., ... or .....

2. Answer in 1- 2 sentences :

a) Explain the format specification of printf for real numbers in decimal form.

.....

.....

b) Explain the format specification of printf for real numbers in exponential form.

.....

.....

c) How will you use scanf to read numbers in decimal and exponential form?

.....

.....

3.3.2.3 Input/Output of character strings :

Characters or strings (which is an array of characters) can also be input and output using **scanf** and **printf** functions.

scanf for character and strings :

We have already seen the use of **getchar()** to read a single character from the keyboard. We can use **scanf** function to achieve the same effect. The general form to read a character string is

%wc or %ws

where w indicates the field width. %c can used to read a single character and %s is used for a string of characters. It is important to note that the reading of the input with the %s specifier terminates as soon as a blank space is encountered.

Example : The following example illustrates the use of **scanf** and **printf** for characters and strings.

```
main()
{
        char ch1;
        char str1 [20], str2[30];
        printf("\nEnter a character :");
        scanf("%c", &ch1);
        printf("\nEnter a string :");
        scanf("%s", str1);
        printf("Enter a new string :");
        scanf("%s",str2);
        printf("\n%c", ch1);
        printf("\n%s\n%s", str1, str2);
}
```

A sample run:

```
Enter a character : s
Enter a string : High-Level-Language
Enter a new string : High Level Language
s
High-Level-Language
High
```

In the above example, the strings str1 and str2 are declared as character arrays. This topic shall be covered in detail in the subsequent chapters. For the time being just use it as has been shown and remember that a string is an array of characters. Note that in the second example of string input only the first word has been read. Reading

of input data terminates as soon as a blank space is encountered. However, this limitation can be overcome by making use of the **%[ ]** specification, which enables **scanf** to read input strings which contain blank spaces.

The %[ ] specifier : Some versions of scanf support the %[ ] specifier. It has the following conversion specifications for strings :

- %[characters] : This means that only the characters specified in the brackets are allowed in the input string. eg. %[a-z] will mean only lower case characters are allowed. This implies that we can make use of the %[ ] specification to read strings which have blank spaces in them.

- %[^character]: This implies that the characters specified after ^ are not allowed in the input string. eg. %[^a] implies that character a is not allowed in the input string.

The following example demonstrates the use of the above conversion specifications for strings :

Example :

```
main()
{
        char str1[40], str2[40];
        printf("Enter string :");
        scanf("%[A-Z a-z ]", str1);
        printf("\nThe string entered is : %s", str1);
        printf("\nEnter new string :");
        scanf("%[^A-Z]",str2);
        printf("\nThe second string is : %s", str2);
}
```

A sample output:

```
Enter string : High Level Language
The string entered is : High Level Language
Enter new string :small case Allowed
The second string is : small case
```

In the first format specification, it is specified that the string can contain lowercase alphabets, uppercase alphabets and blanks, hence the input string is read correctly and printed. In the second case, uppercase alphabets are not allowed, hence reading of the string terminates as soon as the upper case alphabet A is entered. Follow the program carefully and try it with different types of specifiers and inputs.

printf for characters and strings :

The format specification for printing a character is :

%wc

where w specifies the width in columns for the character. i.e. the character will be displayed in a column width w. Default value for the width is 1. The output is right justified. In order to make it left justified you can place the minus sign before the field width.

The format specification for printing a string is :

%w.ps

where w is the field width for display and p implies that only the first p characters of the string should be printed. The display is right justified and can be made left justified by making use of the minus sign before the width specifier.

The following program demonstrates the use of **printf** to print characters and strings in a formatted form :

Example :

```
main()
{
        char a = 'A';
        char str1 [30] ="Times of India";
        printf("Character in different formats :\n");
        printf("\n%c\n%3c\n%5c\n%3c\n%c", a,a,a,a,a);
        printf("\nString in different formats :");
        printf("\n%40s", str1);
        printf("\n%30.8s",str1);
        printf("\n%-40s",str1);
}
```

The output of the program :

```
Character in different formats :

A
   A
      A
   A
A


String in different formats :

                                Times of India
                         Times of
Times of India
```

In the second string output only the first eight characters of the string are output since the format specifies that only 8 characters be displayed. The last line is printed left justified whereas the first one is displayed right justified in a width of 40 columns.

In most of the programming examples seen so far, we have seen that it has not been necessary to make use of the standard input/ouput header file. This is because the **scanf** and **printf** functions do not need the stdio.h file to be included in your program. However, some functions like **getchar()** and **putchar()** may require that you include the stdio.h file in your C program. While writing these programs, make sure whether or not the inclusion of the header file is essential for the particular compiler you are using.

3.3.2.3 Check Your Progress.

1. Write short notes on :

a) scanf for characters and strings :

.....

.....

b) the use of %[ ] specifier

.....

.....

c) The general form of printf for character and strings.

.....

.....

3.3.2.4 Input/Output of mixed data types :

You can also use **scanf** function to input data which may include mixed data types like integers, floats and chars. The order and type specifications should be given precisely in the **scanf** statement. If an attempt is made to read an item whose type does not match, the function does not read anything further and returns the values read.

Thus the statement

scanf("%d %f %c", &int1, &float1, &char1);

will read the data in the sequence of an **int** followed by **float** followed by **char.** In the event, the correct sequence of the data types is not followed **scanf** will terminate further reading.

In the same manner like using **scanf** for reading, we can use the **printf** to output mixed data types. The format string specifies the number of variables and their data types, Hence the variable list should match the format string in number, type and order.

A program to read mixed data types with **scanf** and print them with **printf :**

Example :

```
main()
{
        int i , j;
        float a , b;
        char ch = 'Z';
        char str 1[20] = "Programming in C";
        printf("\nEnter integers i and j and float a,b\n");
        scanf("%d%d%f%e", &i, &j, &a, &b);
        printf("Mixed mode formatted data output :\n");
        printf("Integers a and b :%d%6d\nCharacter and string :%3c\t%
        -30s\nfloats\n%f\t%e", i, j, ch, str1, a, b);
}
```

A sample output:

```
Enter integers i and j and float a,b 100      200      88.99    6.75e2
Mixed mode formatted data output:
Integers a and b :100 200
```

Character and string : Z Programming in C

floats

88.998001 6.75898e+02

More about scanf:

When **scanf** completes reading the list of data items it returns a value which is equal to the number of items that have been read successfully. This value can then be used to determine whether any errors have occurred during data input.

eg. scanf("%d%d%c", &a,&b,&ch);

is supposed to read three values where a and b are **int** and ch is **char.** If you enter the following values :

5 5 a

scanf will return 3 which implies that it has read all three data items correctly. On the other hand if you input the data as follows :

5 a 5

scanf will return a value 1 which means that only one data item has been successfully read.

- Remember that all the format specifications in the control string should match the arguments in order, number and type.

- The data items which are input must be separated by spaces, tabs etc. **scanf** will terminate reading further input if it encounters an invalid mismatch of data.

When using printf:

- When using **printf** make appropriate use of headings for variables names wherever necessary.

- Space the output in such a form that it appears neat and easy to read.

- Output messages wherever required.

3.3.2.4 Check Your Progress.

Write true or false :

a) scanf returns an integer value.

b) printf cannot be used to output mixed data types.

c) Mixed data types can be input in any order using scanf.

d) Formatted data output is not possible with printf.

e) The variable list has to match the format string in type when using printf.

3.4 SUMMARY

In this chapter we studied different Input output functions. These functions are known as the standard I/O library. Most of the standard Input/Output functions are included in the <stdio.h> (standard input-output header file). Therefore, Every C program must include the statement

#include <stdio.h>

This statement will tell the compiler to search for this file, and place its contents at this point in the program. The contents of the header file will now become a part of your program.

Catagories of Input/Output Functions : The input/output functions available in the standard input/output file stdio.h can be classified into three broad catagories :

Console Input/Output Functions where input is supplied from the keyboard and output displayed on the video display unit.

Formatted and Unformatted Input/Output: Formatted input/output allows us to read the input or display the output to be formatted as per our requirements.

Unformatted Console Input/Output Functions :

Character Input/Output: Character input/output functions are useful for reading and writing single characters from the keyboard and to the monitor. To read a single character we use the getch(), getche() and getchar() functions.

3.5 CHECK YOUR PROGRESS - ANSWERS

3.1 & 3.2

1. a) scanf

b) printf

c) stdio.h

2. a) C does not have any built in input/output statements as a part of its grammar. Therefore, all input and output operations are to be carried out with the help of functions like printf and scanf. Many functions have now become standard for input output in C. These functions are known as the standard I/O library. Most of the standard Input/Output functions are included in the <stdio.h> (standard input-output header file). Therefore, for effecting input/output functions, a C program must include the <stdio.h> file. Some functions like printf and scanf do not require the inclusion of the standard input/output header file.

b) Console Input/Output is one of the catagories of input/output functions available in the standard input/output file <stdio.h>. In this category, input is supplied from the keyboard and output displayed on the video display unit.

3.3.1

1. a) - iv

b) - iii

c)- i

d) - ii

2. a) Formatted input/output allows us to read the input or display the output to be formatted as per our requirements. Formatting implies specifying where you want the output to appear on the screen, how many spaces do you wish to have between the various outputs, the number of decimal places after the decimal point in case of float data etc. Thus in formatted input/output we input or output data which has been arranged in a particular format.

b) getche() and getchar() are both functions used to read a character. getche() and getchar() both echo (display) the character that is typed on the screen. However the difference between these is that getchar() requires the user to hit the Enter key after the character is typed in and getche() does not.

3.3.2.1 & 3.3.2.2

1. a) %d %f

b) after the % sign

c) 0 after the % sign

d) field width

e) space, tab, newline

2.a) We can use printf to output real numbers in both decimal notation and exponential notation. In order to use printf for the decimal notation the format specification is :

%w.pf

w is an integer which indicates the minimum number of positions to be used to display the value, integer p indicates the number of digits to be displayed after the decimal point (this is also known as precision). The default precision is 6. The value is rounded to p digits after the decimal point when it is printed. The value is printed right justified.

b) printf can be used to output real numbers in the exponential form. The format specification for exponential notation is :

%w.pe

The integer p specifies the precision of the number of digits after the point. The default precision is 6. The field width w should be large enough to satisfy the condition

w >= p + 7

The value is printed right justified in the field width w. Leading blanks will appear if necessary. As in the case of int, you can also make use of 0 or - for printing leading zeroes or left justification respectively.

c) In order to input real numbers we make use of scanf. scanf uses %f specification for both, decimal point and exponential notation. Values for variables can be input in either decimal notation or exponential notation.

3.3.2.3

1. a) The general form of scanf to read a character string is

%wc or %ws

where w indicates the field width. %c can used to read a single character and %s is used for a string of characters. It is important to note that the reading of the input with the %s specifier terminates as soon as a blank space is encountered.

b) The %[ ] specifier : Some versions of scanf support the %[ ] specifier.It provides additional features when used with scanf for strings. It has the following conversion specifications for strings :

- %[characters] : This means that only the characters specified in the brackets are allowed in the input string. We can thus make use of the %[ ] specification to read a string which has blank spaces.

- %[^character] : This implies that the characters specified after ^ are not allowed in the input string.

c) The general form of printf for characters and strings has the format specifications as follows :

The format specification for printing a character is :

%wc

where w specifies the width in columns for the character. i.e. the character will be displayed in a column width w. Default value for the width is 1. The output is right justified. In order to make it left justified you can place the minus sign before the field width.

The format specification for printing a string is :

%w.ps

where w is the field width for display and p implies that only the first p characters of the string should be printed. The display is right justified and can be made left justified by making use of the minus sign before the width specifier.

3.3.2.4

1. a) True

b) False

c) False

d) False

e) True

3.6 QUESTIONS FOR SELF - STUDY

1. Write short notes on :

a) Catagories of Input/Output Functions in C

b) scanf and printf for Integers

c) Input/Output of real numbers

2. State and correct errors if any in the statements given below assuming the following declaration in a program have been made:

```
int a, b;
float i,j;
char ch,ch1;
```

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

b) scanf("%f%f%c", &i,&j,&a);

c) printf("%6.2f%d%d", &i,&a,&b);

d) printf("%c%c%c", ch,ch,ch1);

3. Given an input string "Formatted Input/Output" write a program to print the following from this string :

a) Formatted

b) Formatted In

4. For the values of a = 600, b = 89.546 and ch = 'A' attempt the following from this string :

a) print the value of a in :

(i) right justified manner

(ii) leading zeroes in a width of 8

(iii) left justified manner

b) print the value of b in :

(i) decimal format with leading zeroes and precision of 4

(ii) exponential format

c) print the value of ch :

(i) In a width of 10

(ii) in a width of 1

(iii) in a width of 10 all on different lines

5. Answer the following :

a) What is meant by formatted and unformatted input/output?

b) How will you use printf to print mixed mode data types? Explain with an example.

c) How will you make use of scanf to determine whether the data input has been correctly read or not? Explain with an example.

d) What are the formatting features available for printf?

3.7 SUGGESTED READINGS

The Spirit of C : Mullish cooper

Exploring c Yashwant Kanitkar

C for Beginners : Madhusudan Mothe

CHAPTER 4

CONTROL STATEMENTS

4.0 OBJECTIVES

Friends, After studying this chapter you will be able

- explain what to do in programming situations involving branching depending upon certain conditions
- discuss the control statements in C for this purpose which are : **if, if-else, switch, conditional operator** and **goto.**
- state additional features like nested ifs, the **if-else** ladder
- be able to make use of **logical operators** with **if** statements for more structured programming
- develop a number of programs using these various control statements.

4.1 INTRODUCTION

Decision Control:

While writing programs, it is often necessary to change the order of execution of depending upon certain conditions. It is also required to execute an instruction or a set of instructions until a specific condition is met. Thus, it is necessary to take decisions and depending upon the result of the decision execute appropriate statements i.e. we are required to execute a set of instructions in a particular situation and an entirely different set of instructions in other situations.

The C language has the following decision making instructions :

- if statement

- switch statement

- conditional operator statement

- goto statement

This chapter will discuss in detail all the above control statements available in C one by one starting with the simple **if** and progressing to more complex structures

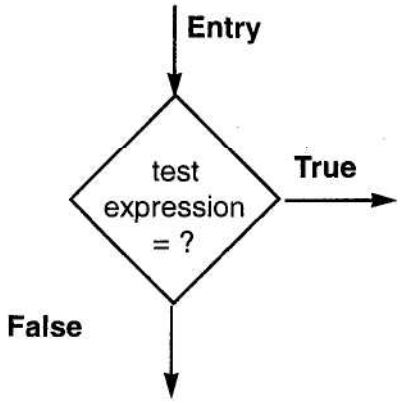
like the nested if and if-else ladder. Having mastered this, you are sure to be able to write numerous programs in C.

4.2 THE IF STATEMENT

The **if** statement is used to control the flow of the execution of statements in a program. The general form of the **if** statement is

if (test condition)

The condition to be evaluated is placed in a pair of parenthesis immediately following the keyword **if**. First, the test condition is evaluated. If it is true then the statement (or set of statements) following **if** are executed. If the condition evaluates to false, the statement (or set of statements) following **if** are skipped. Thus there is a two way branching for the **if** statement: one for the true condition and the other for the false condition. Thus the **if** condition is expressed as :



The if construct

The test condition is generally expressed with the help of the **relational operators** in C. We have already seen relational operators in the previous chapter. To revise :

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

The test condition is first evaluated and then depending upon whether it is true or false, the corresponding set of instructions are executed subsequently. C offers a number of forms for the **if** statement. Let us now see the various forms of the **if** statement:

4.2.1 The simple if statement:

The simple **if** statement takes the following form :

```
if (test condition)
    {
        statement block;
    }
```

statement -a;

The *statement block* can be a single statement or a set of statements which are to be executed if the **if** condition evaluates to true. The statement block is to be enclosed within a pair of braces or else the compiler will only execute a single statement following

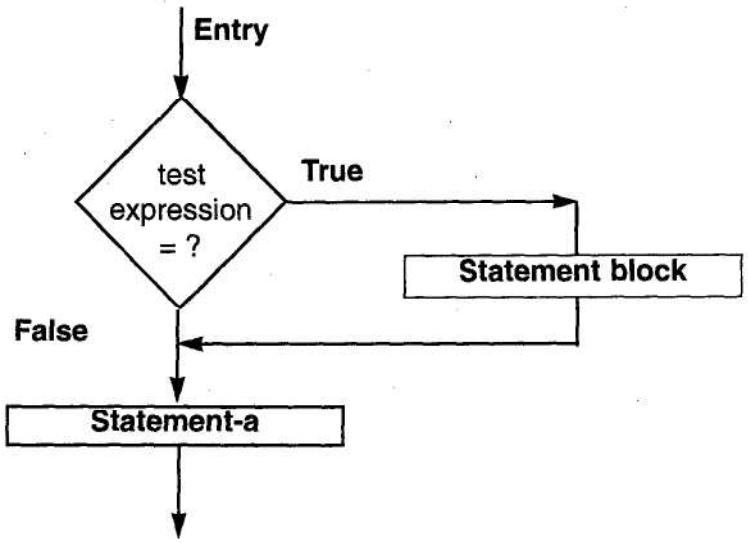


Fig. 1 - Flowchart for simple if statement

if. If the condition evaluates to false, then the block is skipped and statement-a is executed. Here it is important to note that when the condition evaluates to true the statement block is executed and then the statement-a is also executed. Fig.1 illustrates the flowchart of the simple if statement.

A few examples using the **if** statement will further elaborate this :

Example:

```
/* Program to illustrate a simple if statement */
main()
{
        int a, b;
        printf ("\nEnter values for a and b :");
        scanf("%d%d", &a,&b);
        if(a >b)
        {
        printf("\na is greater than b");
        }
        printf("\nEnd of example");
}
```

A sample run of the program :

```
Enter values for a and b :10 20
End of example
Enter values for a and b :20 10
a is greater than b
End of example
```

When you run this program, and enter values for a and b, (in our case as 10 and 20 respectively) the test condition (a >b) is evaluated. Since a>b is false, the if loop is skipped and the statement “End of example” is printed. In the second run of the program the values of a and b are given as 20 and 10 respectively, a >b evaluates to true and the statement “a is greater than b” is printed. The statement “End of example” is also printed.

Example 2 :

```
/*Program to find whether the input number even */
main()
{
        int num;
        printf("\nEnter any number:");
        scanf("%d", &num);
        if (num % 2 == 0)
        {
        printf("\nYou entered the number %d," num);
        printf("\nThe number is even");
        }
        printf("\nEnd of example");
}
```

Sample output :

```
Enter any number :20
The number is even
End of example
Enter any number :3
End of example
```

In the first run the input number entered is even, hence statement after **if** is executed, in the second run the number input is odd, therefore if construct is skipped. Note the use of braces after the if. When you want to execute multiple statements after **if** then they must be enclosed within a pair of braces.

4.2.1 Check Your Progress.

1 What is the output of the following programs ?

a)
```
main()
{
        int a = 300, b = 0, c;
        if(a >= 400)
                b = 100;
                c = 200;
        printf("\n a = %db = %d c = %d", a,b,c); }
}
```

.....

.....

b)
```
main()
{
        int a = 600, b, c;
        if(a >= 400)
                b = 100;
                c =20;
        printf("\n a = %db = %d c = %d", a,b,c);
}
```

.....

.....

2. The following is a program in C :

```
main()
{
    int x = 10, y = 5, n;
    if(n > 0)
    x = x + 10;
    y = y + 10;
}
```

What will be the values of x and y for n = 0 and n = 1 in the above program?

.....

.....

.....

3. Correct the errors if any and rewrite the statements given below :

a)
```
main()
{
    int i, j;
    i = 10;
    j = 5;
    if(I >=j);
        printf("\ni is greater than j");
}
```

.....

.....

b)
```
main()
{
    int i, j;
    i = 10;
    j = 10;
    if(i=j)
    printf("\ni is equal to j");
}
```

.....

.....

4.2.2 if-else statement :

There are situations where you may want to execute a statement (or a set of statements) when the **if** condition evaluates to true and another statement (or a set of statements) when the condition evaluates to false. In such cases we make use of the **if-else** statement. The general form of the **if-else** statement is

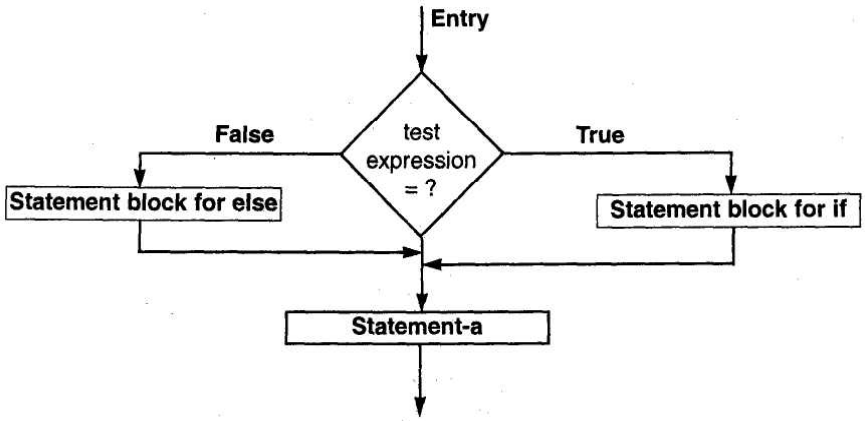


Fig2 -Flowchart for if-else statement

```
if (test condition)
{
        statement-block;
}
else
{
        statement  block;
}
statement-a;
```

The test condition is evaluated. if it is true the statement block following **if** (called the if block) is executed, if it is false the statement block following **else** (called the else block) is executed. The control is subsequently transferred to statement-a. In any case only one statement block (either following **if** or following **else)** is executed. Fig2 illustrates the flowchart of **if-else.**

We see the use of the **if-else** statement with the help of the following programs :

Example :

```
/* Program to illustrate a if-else statement */
main()
{
        int a, b;
        printf ("\nEnter values for a and b :");
        scanf("%d%d", &a,&b);
        if(a>b)
        {
                printf("\na is greater than b");
        }
        else
        {
        printf("\na is less than or equal to b:');
        }
        printf('\nEnd of example");
}
```

A sample run of the program :

```
Enter values for a and b :10 20
a is less than or equal to b
End of example
Enter values for a and b :20 10
a is greater than b
End of example
```

In the first sample run of the program the if condition evaluates to false, therefore the else block is executed. In the second run if condition evaluates to true hence the statement block after if is executed. In either case the statement "End of example" is output.

Example : Modify the above program of even number using if-else to print either the number is even or it is odd.

```
/*Program to find whether the input number is odd or even */
main()
{
        int num;
        printf("\nEnter any number:");
        scanf("%d", &num);
        if (num % 2 == 0)
                printf("\nThe number is even");
        else
                printf("\nThe number is odd");
        printf("\nEnd of example");
}
```

A sample output :

```
Enter any number :20
The number is even
End of example
Enter any number :3
The number is odd
End of example
```

Example 3:

```
/* Program to illustrate the use of if-else*/
main()
{
        int marks;
        printf("Enter marks for student:");
        scanf("%d", &marks);
        if(marks >= 40)
        {
                printf("\nCongratulations ! You pass !!");
        }
        else
        {
                printf("\nSorry to say you fail");
        }
        printf("\nEnd of example");
}
```

As the program illustrates, the condition is checked. If marks input are greater than 40, the student is declared pass else declared fail. In either situation, only one block is executed and the subsequent statement “End of example” is executed.

4.2.2 Check Your Progress.

1 What is the output of the following programs :

a)
```
main()
{
    int a = 300, b, c;
    if(a >= 400)
        b = 100;
    else
    {
        b = 10;
        c = 50;
    }
    printf(“\n a = %db = %d c = %d”, a,b,c);
}
```
.....

b)
```
main()
{
    int x, y;
    x = 1;
    y = 1;
    if (x ==y)
        printf(“\nx and y are equal”);
    else
        printf(“\nx and y are not equal”);
}
```
.....

2. The following is a program in C :

```
main()
{
int x, y, n;
x = 10;
y = 5;
if(n >0)
    x = x + 10;
else
    x = x + 20;
y = y + 20;
}
```

What will be the values of x and y for n = 0 and n = 1 in the above program?

.....

4.2.3 Nested if - else statements :

It is possible to nest **if-else** statements in an **if** or **else** statement. i.e. we can make use of more than one **if-else** statements by nesting them. There is not limit to the number of **if-else** nesting. The general form of nested **if-else** is **:**

```
if (test condition 1)
{
        if (test condition 2)
        {
                statement block-1
        }
        else
        {
        statement block-2;
        }
else
        {
        statement block-3;
        }
}
statement-a;
```

The flowchart in Fig3 shall explain the logic of these nested **if-else** statements. If test condition 1 is true, then test condition 2 is evaluated. If true, statement block-1 is executed otherwise statement block-2 is executed. If test condition has evaluated to false, then statement block-3 is evaluated. Note that the second **if-else** construct is nested in the outer **if** statement. You can nest any number of **if-else** blocks in a program.

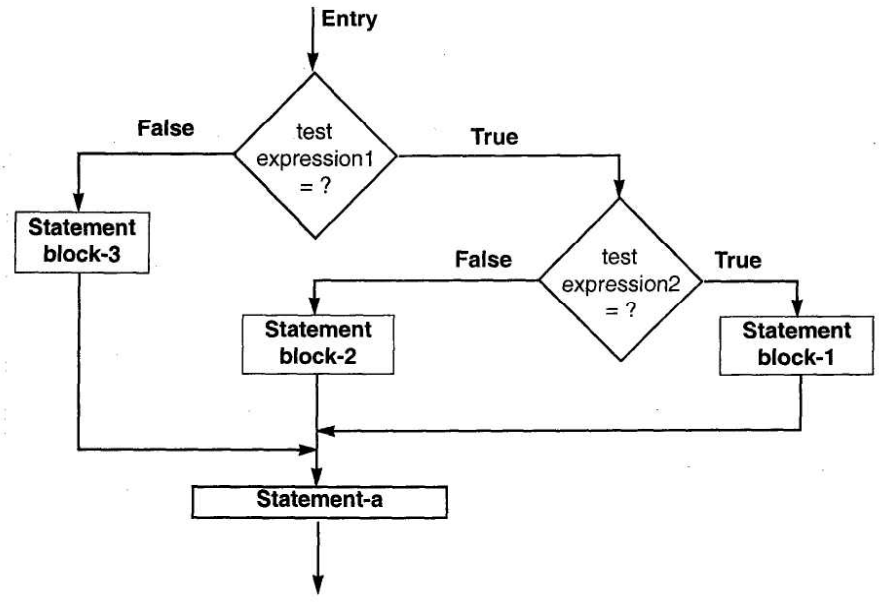


Fig 3 - Flowchart for nested if-else statement

Let us see nested **if-else** with the help of the following examples :

Example 1 :

```
#include <ctype.h>
main()
{
char ch1;
printf("\nEnter a character:);
ch1 = getchar();
if(isalpha(ch1) >0)
        printf("\nYou entered an alphabet");
else
        if(isdigit(ch1)>0)
                printf("\nYou entered a digit");
        else
        printf("You entered a character which is not alphanumeric");
}
```

A sample output:

```
Enter a character :a
You entered an alphabet
Enter a character:%
You entered a character which is not alphanumeric
```

Note the first line of the program

#include <ctype.h>

This program has made use of the character functions **isalpha** and **isdigit.** These functions are in the file **cytpe.h** and therefore this file should be included in the program with the **#include** statement. In the first sample run the character entered is a, hence **isalpha** is true. The **else** block is therefore skipped. In the second run **isalpha** evaluates to false. Therefore the **else** block is entered. Here again there is a **if** statement to check whether character is a digit. This condition evaluates to false so the else block is executed. Follow the program sequence carefully with further sample runs.

Example 2 :

```
/* Program to illustrate nested if-else */
main()
{
int marks;
        printf("Enter marks for student:");
        scanf("%d", &marks);
        if (marks <75)
        {
                if (marks >= 60)
                {
                printf("\nCongratulations ! You get a first class");
                }
                else
                {
                printf("\nYou get a second class");
                }
```

```
        }
        else
                {
                printf("\nHearty Congratulations ! You secure a distinction");
        }
        printf("\nEnd of example");
}
```

The study of flow of this program is left to the student as a self study.

4.2.3 Check Your Progress.

1. Write programs in C for the following :

a) Enter the sex of a person as character 'M' for male and 'F' for female. If person is female and age is less than 60 output "Not eligible for senior citizen benefit" else output "Eligible for senior citizen benefit'. If person is male and age is less than 65 output "Not eligible for senior citizen benefit" else output "Eligible for senior citizen benefit".

b) Input the cost price and selling price of an item. Check whether the sale has effected a profit or a loss and by how much.

2. State if the following statements are true or false :

a) One if statement can have more than one else clause.

b) There have to be the same number of if statements are the number of else statements in a program.

c) An if-else can be nested in an else.

d) if (a == b) is a valid expression of an if statement.

e) Every if should have a corresponding else.

4.2.4 The if-else ladder:

When multipath decisions are involved we make use of the **if-else** ladder. The **if-else** ladder is a chain of **if**s where each else has an associated **if.** The form of the **if-else** ladder is :

```
if(condition_1)
        statement_1;
                else if(condition_2)
                        statement_2;
                                else if(condition_3)
                                        statement_3;
                                        _____________
                                        _____________
                                        else if(condition_n)
                                        statement_n;
                                        else
                                        default;
statement-a;
```

In the **if-else** ladder the conditions are evaluated from the top down the ladder. As soon as a condition is found true, the statement (statement block) associated with it is executed and control goes out of the ladder to the statement-a. If all conditions become false, the default statement in the final **if** statement is executed and subsequently

statement-a is executed.

Example :

The program makes use of the **if-else** ladder.

The percentage marks are entered and the grades are allotted as follows :

per >= 60	First Class
per >=50 and per <= 60	Second Class
per >= 40 and per <= 50	Pass Class
per < 40	Fail

```
main()
{
    /* A program to print grades */
    int per;
    printf("\nEnter percentage:");
    scanf("%d", &per);
    if(per >= 60)
    {
        printf("\nFirst class");
    }
    else
    {
        if(per>=50)
        {
        printf("\nSecond Class");
        }
        else
        {
            if(per >=40)
            printf("\nPass Class");
            else
            printf("\nFail");
            }
    printf("\nEnd of Program");
}
```

As you can see, the program becomes difficult to understand and debug with such use of multiple **if-else** constructs. To overcome these difficulties we make use of the logical operators with **if-else.** Let us see how to do this in the next section.

4.2.5 Use of Logical Operators with if statement:

We have already seen the logical operators of C in the previous chapter. Let us see the use of these logical operators with the **if** statement. As we have noted before, nested **if-else** statements and **if-else** ladders make programs difficult to read and understand. They have also to be written carefully and debugging becomes difficult. The corresponding if's and else's have to be matched correctly. In such situations, we make use of logical operators and combine the **if** statements with the help of these operators. The logical operators in C are :

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

These logical operators have already been introduced in the previous chapters. Let us see how to make use of them with the help of the following program :

Input the percentage of marks obtained by a student.

If percent greater than or equal to 60 - First class

If percent between 50-59 - Second class

If percent between 40-49 - Pass class

If percent less than 40 - Fail

Example :

```
/* Program to illustrate the use of logical operators */
main()
{
        int per;
        printf("\nEnter percentage :");
        scanf("%d",&per);
        if(per>=60)
                printf("\nFlrst Class");
        if((per>=50)&&(per<60))
                printf("\nSecond Class");
        if((per>=40)&&(per<50))
                printf("\nPass Class");
        if(per<40)
        printf("\nFail");
}
```

Here, we make use of the **&&** operator to combine two conditions. If both the conditions evaluate to true then the statement following the if will be executed. If either of the condition evaluates to false, or if both evaluate to false then the statements following **if** are skipped.

Example :

```
main()
{
        /* Program to check greatest of three numbers a,b, c */
        int a, b, c;
        printf("\nEnter value for a :");
        scanf("%d", &a);
        printf("\nEnter value for b :");
        scanf("%d", &b);
        printf("\nEnter value for c :");
        scanf("%d", &c);
        if((a >b) && (a > c))
```

```
            printf("\na is the greatest number");
        else
        {
            if((b>a) && (b>c))
                printf("\nb is greatest");
            else
                printf("\nc is greatest");
        }
}
```

A sample output:

```
Enter value for a : 100
Enter value for b : 150
Enter value for c : 5
b is greatest
```

4.2.6 Hierarchy of Logical Operators :

When using logical operators, relational operators and arithmetic operators together it is important to know in what sequence they will be evaluated. This is especially important in situations when you are combining multiple expressions involving these operators with the help of logical operators. The following table shows the hierarchy of arithmetical and logical operators. The operator at the topmost position has the highest priority and down the list the priority goes on decreasing.

Operator	**Type**
!	Logical NOT
* / %	Arithmetic and modulus
+ -	Arithmetic
< <= > >= =	Relational
== !=	Relational
&&	Logical AND
||	Logical OR
=	Assignment

This table will be useful to you when you are combining number of expressions while writing programs.

4.2.4, 4.2.5 & 4.2.6 Check Your Progress.

1. Correct the following set of statements and rewrite :

a)
```
char ch1, a = 'n', b ='z'
if (ch1 = a or ch1 = b)
    printf ("\n You entered the correct code");
else
    printf("Enter again");
```

.....

.....

.....

.....

b) if(a<=10 and b = 5)
printf("\nCorrect values)
.....
.....

c) int i = 5, j = 2;
if(i = 5 || j != 2)
printf("Check again!");
.....
.....

d) (a == b) && (!c) | b & c);
.....
.....

2. Write a program in C to determine whether a character entered from the keyboard is:
a capital letter or a small letter or a digit or a special symbol
(Hint : Make use of the ascii values of characters as given in the previous chapter)

3. Write a program to find the smallest of three numbers a, b and c.
(First check whether all three are equal. If they are do not attempt to check further for the smallest.)

4.3 THE SWITCH STATEMENT

We have seen that we can make use of multiple **if-else** statements in our programs to control selections. However, as the number of options goes on increasing the complexity of programs also increases. The program becomes difficulty to read and understand. C provides the **switch** statement to avoid the use of such series of **if** statements. The general form of this **switch** control statement is :

```
switch (integer expression)
{
    case constant-1:
        statement block -1;
        break;
    case constant-2;
        statement block-2;
        break;
    ____
    ____
    default:
        default statement block;
        break;
}
statement-a;
```

An integer expression is enclosed in a parenthesis following the **switch** keyword. It is any C expression which will yield an integer value. The keyword **case** is followed by an **integer** or **character** constant. Each constant in each case should be different from all others. The **switch** statements works as follows :

- First, the integer expression following **switch** is evaluated. The expression evaluates to an integer as we learnt.

- Then, the value of the expression is matched one after the other, with each **case** constant. When a match is found, the statement block following that particular **case** is executed. The **break** statement signifies the end of each case and causes an exit from the switch statement and control is transferred to statement -a. The **default** statement is optional. It will be executed if the value does not match any of the case values.

Here care has to be taken to include the **break** statement. If a **break** statement is not inserted in the case then the program not only executes the statements following the **case** where the match has been met, but also all the following **case** statements and the **default** statement as well.

Fig.4 shows the flowchart for the **switch** statement. Go through it carefully and study it.

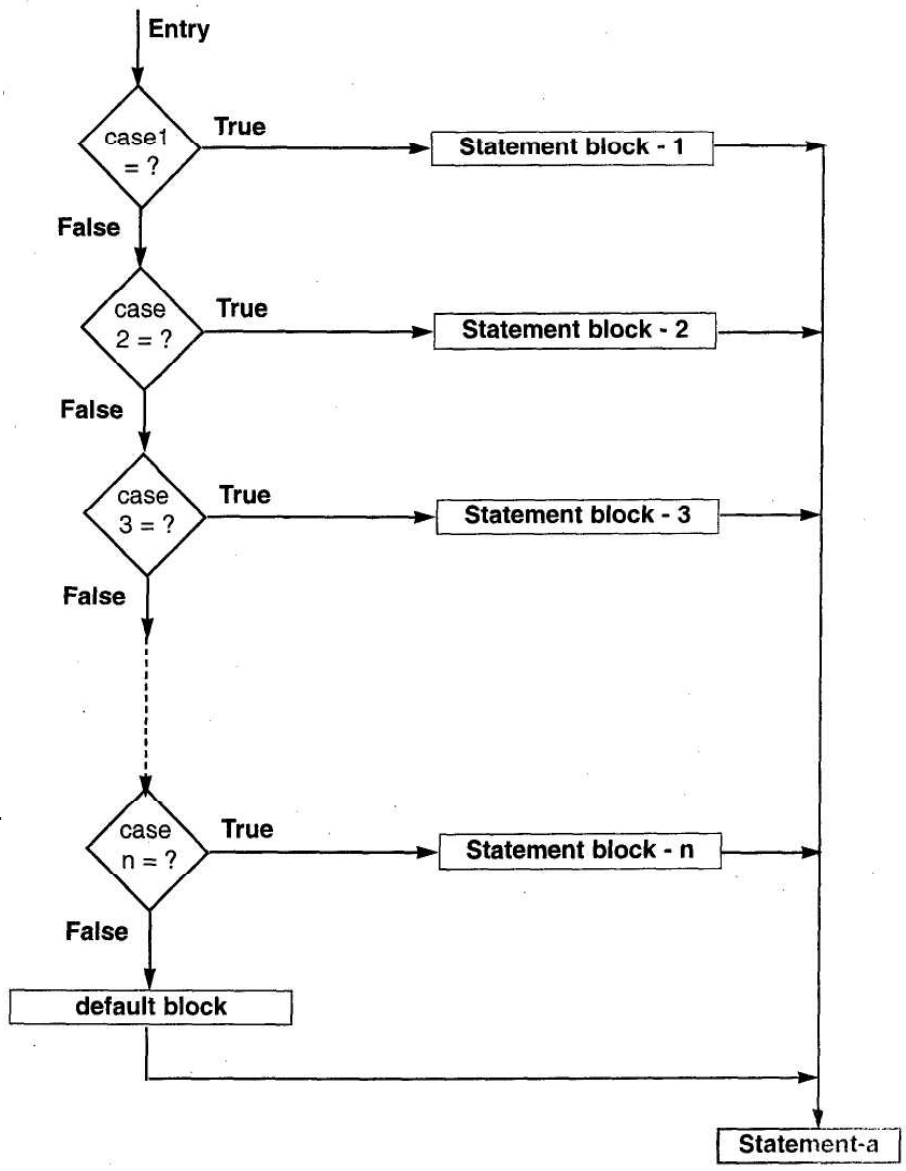


Fig 4- Flowchart for case control structure

We shall now see the use of the **switch** statement with the following example :

Example 1:

```
/* Example to illustrate the use of the switch statement */
main()
{
int grade;
printf("Enter code for grade :");
scanf("%d", &grade);
switch(grade)
{
        case 1:
                printf("\nDistinction");
                break;
        case 2:
                printf("\nFirst Class");
                break;
        case 3:
                printf("\nSecond Class");
                break;
        case 4:
                printf("\nFail");
                break;
        default:
                printf("\nResult Declared");
                break;
        }
        printf("\nEnd of example");
}
```

A sample output:

```
Enter code for grade : 4
Fail
End of example
```

The **switch** statement leads to a more structured approach as compared to the multiple **if**s. The **switch** statement is used often for menu selection. Let us make use of the switch statement to learn how to use **char** values in **case** and **switch** :

Example :

```
main()
{
char ch1;
printf("\n File");
printf("\n Edit");
printf("\n View");
printf('\n Window");
printf('\n Help");
```

```
printf("\n\nType the first alphabet of any menu item to select:");
scanf("%c",&ch1);
switch (ch1)
{
        case 'F':
        case 'f' :
                printf("\nYou have selected File option");
                break;
        case 'E':
        case 'e':
                printf('\nYou have selected Edit option");
                break;
        case 'V:
        case v:
                printf("\nYou have selected View option");
                break;
        case 'W:
        case 'w':
                printf("\nYou have selected Windows option");
                break;
        case 'H':
        case 'h':
                printf("\nYou have selected Help option");
                break;
        default:
                printf('\nNo option selected");
        }
}
```

Note in the above program that there are no statements after the **case** statements following the capital alphabet eg. there are no **case** statements after case 'F' : or case 'W'. In such situations, if you entered the character 'F' there are no statements after 'F'. Hence control goes to the next case which is 'f. The statements in this **case** therefore get executed. (Since there is no **break** statement in the case after 'F'). Thus your program will work for both options of uppercase and lowercase alphabets of the menu. Go through the program carefully and understand its working thoroughly. Such programs have to be written with great care and the order of case statements is also very important.

You can also mix **char** and **int** constants in a case eg.

```
switch(i)
{
        case 'a' :
                printf("\nA char constant");
        case 5 :
                printf('\nAn int constant");
        default:
                printf("\nMixinq of int and (+)(+)(+),
}
```

One more thing to remember about **case** : Even if you have to execute multiple statements after **case,** it is not necessary to enclose them within a pair of braces.

4.3 Check Your Progress.

1. State True or False.

a) A switch expression can be of any type.

b) Program execution stops when a break is encountered.

c) Every case statement need not have a statement after it.

d) Char values can be used in switch.

e) case (i <=20) is not valid in C.

2. Write a program to add, subtract, multiply, divide two numbers a and b using the switch statement. Each case in switch should perform one of the above operations. The default shall indicate that no operation has been performed.

3. Find the errors in the following statements and rewrite correctly.

a)
```
switch (i)
case 1;
case 2 :
default
.....
```

b)
```
switch (m)
case 'z':
      printf("\nz")
case '1'
      printf("\n1");
default:
      printf("End");
.....
```

4.4 THE CONDITIONAL (?:) OPERATOR

We have seen the **conditional operator** in the previous chapter. C has one conditional operator. The syntax of the conditional operator in C is

exp1 ? exp2 : exp3;

where exp1, exp2 and exp3 are expressions. Here, exp1 is evaluated first. If it is true (non zero), then exp2 is evaluated and it becomes the value of the expression. On the other hand if exp1 is false, then exp3 is evaluated and its value becomes the value of the expression. The conditional operators need not be used only in arithmetic statements.

Let us now see the use of this conditional operator for making two way decisions i.e. make use of the conditional operator like the **if** statement.

eg.

```
if (x==0)
      y = 0;
else
      y = 10;
```

In the above if-else statement, if x==0 is true then y is assigned a value 0 else y is assigned a value 1. Now in place of the **if-else** you can make use of the conditional operator as follows:

y = (x==0)?0:10 ;

Here also the value of the expression x == 0 is evaluated first. If it is true then y is assigned the value of the first expression (i.e. the value 0) else y is assigned the value of the second expression (i.e value 0).

Let us make the use of the conditional operator to write a program to calculate the discount offered for purchase depending upon the following criteria :

purchases <= 1000	1% discount
purchases > 1000 and purchases < 2500	2% discount

Example 1 : To illustrate the use of the conditional operator

```
main()
{
        /* Program to illustrate conditional operator */
        int pur;
        float dis, total;
        printf("\nEnter purchase value:");
        scanf("%d", &pur);
        dis = (pur <= 1000) ? pur * 0.1 : pur * 0.2;
        total = pur- dis;
        printf("\nThe discount is : %7.2f", dis);
        printf("\nThe discounted price is : %8.2f", total);
}
```

Nesting of Conditional Operators :

Conditional operators can be nested as in the case of **if** statements. The limitation of the conditional operator is that you can have only one statement after the ? or the : whereas with the use of **if-else** constructs you can have statement blocks after the **if** or **else** statements.

We shall now modify the above program to illustrate the use of conditional operator for nested **if**:

purchases >= 5000	5 % discount
purchases < 5000 and purchase>= 1000	2 % discount
purchase < 1000	1 % discount

Example 2 :

```
main()
{
        /*Program to illustrate use of conditional operator for nested if*/
        int pur;
        float total, dis;
        printf('\nEnter purchase value :");
        scanf("%d", &pur);
dis = (pur >= 5000) ? (pur * 0.05) : ((pur < 1000) ? (pur * 0.01) : (pur * 0.02));
        total = pur- dis;
        printf("\nThe discount is : %7.2f", dis);
        printf("\nThe discounted price is : %8.2f", total);
}
```

Sample output for various test conditions :

```
Enter purchase value : 6000
The discount is : 300.00
The discounted price is : 5700.00
Enter purchase value : 900
```

The discount is : 9.00

The discounted price is : 891.00

Enter purchase value : 4000

The discount is : 80.00

The discounted price is : 3920.00

4.4 Check Your Progress.

1. Rewrite the above program of the discount on purchase (Example 2) by making use of **nested if.**

2. Make use of the conditional operator and write a program to find out the salary on the basis of the following information :

salary = basic + 2 * basic	for basic = 2000
salary = basic + 3 * basic	for basic > 2000
salary = basic + basic	for basic < 2000

4.5 THE GOTO STATEMENT

C supports the **goto** statement for unconditional branching from one point in the program to another. The syntax for the **goto** statement is :

```
goto label;

_____

_____

label :
statement-1;
statement-2;
```

Thus, the **goto** statement requires a label. The label identifies the place in the program where the program control is to be transferred when the **goto** is encountered. The label is any valid variable name. The same label is to be placed before the statement to which the control is to be transferred. At this point the label is to be followed by a colon. The label can be placed anywhere in the program, either before the **goto** or after the **goto.** Thus when a **goto** is encountered, the flow of control jumps to the statement following the label unconditionally. The following example will illustrate the use of the **goto :**

Example:

```
/* Program to illustrate the use of goto*/
main()
{
        int i;
        printf(Enter value for i:");
        scanf("%d', &i);
        if (i < 0)
                goto out;
        else
        {
        printf(The number you entered is : %d", i);
```

```
        exit;
        }
        out:
        printf("You entered a negative number!");
}
```

In this program, when you enter a negative value for i, the **goto** statement will transfer control to the out label causing the **printf** to be executed. Note here that the **exit** is a standard library function which terminates the execution of the program. We have terminated the program in the else statement with the use of **exit,** since we do not want to output the statement "You entered a negative number" after executing the **else** loop.

A **goto** statement can be used to transfer the control to the beginning of the program to read further input data. Another use of **goto** is in situations when control is to be transferred out of a loop for peculiar conditions. However, one should try to avoid the use of **goto** statement in programs. This is because the programs become unreadable and difficult to debug when **goto** is used. Also, in most situations the purpose of the **goto** can be served by making use of the more logical constructs like if, for, while, switch.

Unconditional **goto** statements can cause a program to go into an infinite loop and execute forever. You will have to specially terminate the loop. The following example will illustrate :

Example :

```
main()
{
        int x;
        sq:
        scanf("%d", x);
        x = x * x;
        printf('\nSquare of x : %d", x);
        goto sq;
}
```

In the above program, the square of integer value x will be computed and then control will again go to **scanf**. This process will continue infinitely. Such infinite loops should be avoided in programs.

4.5 Check Your Progress.

1. Write in short about the goto statement.

.....

.....

4.6 SUMMARY

While writing programs, it is often necessary to change the order of execution of statements depending upon certain conditions. It is also required to execute an instruction or a set of instructions until a specific condition is met. It is necessary to take decisions and depending upon the result of the decision appropriate statements are to be executed i.e. we are required to execute a set of instructions in a particular situation and an entirely different set of instructions in other situations.

The C language has the following decision making instructions :

- if statement
- If else statement
- Nested if else statement
- else if ladder
- switch statement
- conditional operator statement
- goto statement

4.7 CHECK YOUR PROGRESS - ANSWERS

4.2.1

1. a) a = 300 b = 0 c = 200

   b) a = 600 b = 100 c = 20

2. x = 10y = 15

   x = 20 y = 15

3. a)
```
main()
{
    int i, j;
    i = 10;
    j = 5;
    if(i>=j)
    printf("\ni is greater than j");
}
```

   b)
```
main()
{
    int i, j;
    i = 10;
    j = 10; if(i==j)
    printf("\ni is equal to j");
}
```

4.2.2

1. a) a = 300 b = 10 c = 50

   b) x and y are equal

2. x = 30 y = 25

   x = 20 y = 25

4.2.3

1. a)
```
main()
{
    char sexcode;
    int age;
    printf("Enter sexcode (M for Male and F for Female");
```

```
        scanf("%c", &sexcode);
        printf("Enter age :");
        scanf("%d", &age);
        if(sexcode == 'F')
        {
            if(age < 60)
            printf("Not eligible for senior citizen benefit");
        else
            printf("Eligible for senior citizen benefit");
        }
        else
        {
            if(age < 65)
            printf("Not eligible for senior citizen benefit");
        else
            printf("Eligible for senior citizen benefit");
        }
    }
```

b)
```
    main()
    {
        int cp, sp, pl;
        printf("Enter Cost Price of the item :");
        scanf("%d", &cp);
        printf("Enter selling Price of the item :");
        scanf("%d", &sp);
        if(cp > sp)
        {
            pl = cp - sp;
            printf("The sale effected a loss of Rs. :");
            printf("%d", pl);
        }
        else
        {
            pl = sp - cp;
            printf("The sale effected a profit of Rs. :");
            printf("%d", pl);
        }
    }
```

2. a) False
b) False
c) True
d) True
e) False

4.2.3, 4.2.4 & 4.2.5

1. a)
```
    char ch1, a = 'n',b = 'z';
    if (ch1 == a || ch1 == b)
        printf ("\nYou entered the correct code");
    else
        printf("Enter again");
```

b)
```
if(a<= 10&&b==5)
printf("\nCorrect values");
```

c)
```
int i = 5, j = 2;
if(i==5||.j!=2)
printf("Check again!");
```

d)
```
(a == b) && (!c) | b && c
```

2.
```
main()
{
    char ch;
    printf("Enter a character:");
    scanf("%c", &ch);
    if(ch >= 65 && ch <= 90)
        printf("You entered an uppercase character");
    if(ch >= 97 && ch <= 122)
        printf("'You entered a lowercase character");
    if(ch>= 48 && ch <= 57)
        printf("You entered a digit");
    if((ch >= 0 && ch <= 47) || (ch >= 58 && ch <= 64) || (ch >= 91 && ch
    <= 96) ||
    (ch>=123&&ch<= 127))
        printf("You entered a special symbol");
}
```

3.
```
main()
{
    int a,b,c;
    printf("Enter values for a, b, c :");
    scanf("%d%d%d", &a, &b, &c);
    if ((a == b) && (b == c))
        printf("AII a,b,c are equal");
    else
    {
        if(a < b)
        {
            printf("The smallest number is a");
    }
    else
    {
        if(b < c)
            printf("The smallest number is b ");
    else
            printf("The smallest number is c");
        }
    }
}
```

4.3

1. a) False

b) False

c) True

d) True

e) False

2.
```
main()
{
    int a,b;
    char ch;
    a = 100;
    b = 25
    printf("Enter choice :");
    scanf("%c", &ch);
    switch(ch)
    {
        case 'A':
        case 'a':   printf("Addition is: %d", a + b);
                    break;
        case 'S':
        case 's':   printf("Difference is : %d", a - b);
                    break;
        case 'M':
        case 'm' :  printf("Product is : %d", a* b);
                    break;
        case 'D':
        case 'd':   printf("Quotient is : %d", a/b);
            break;
    default:    printf("No operation has been performed");
    }
}
```

3. a)
```
switch (i)
{
case 1:
case 2 :
default:
}
```

b)
```
switch (m)
{
    case 'z':
    printf("\nz");
case 1 :
    printf("\n1");
default:
    printf("End");
```

4.4

1.

```
main()
{
        int val;
        float dis, total;
        printf("Enter value of purchases :");
        scanf("%d", &val);
        if(val<1000)
        {
                dis = val * 0.01;
                total = val - dis;
        }
        else
        {
        if(val < 5000)
        {
                dis = val * 0.02;
                total = val - dis;
        }
        else
        {
                dis = val * 0.05;
                total = val - dis;
        }
        }
        printf("\nThe discount is : %.2f", dis);
        printf("\nThe discounted price is :%.2f, total);
}
```

2.

```
main()
{
        int sal, newsal;
        printf("Enter Salary :");
        scanf("%d", &sal);
        newsal = sal < 2000 ? sal * 2 :(sal > 2000 ? sal * 4 :sal * 3);
        printf("\nThe salary is : %d", newsal);
}
```

4.5

1. The **goto** statement provides unconditional branching from one point in the program to another. The syntax for the **goto** statement is :

```
goto label;
label :
_____________
_____________
statement-1;
statement-2;
```

The label identifies the place in the program where the program control is to be transferred when the **goto** is encountered. When a **goto** is encountered, the flow of control jumps to the statement following the label unconditionally. A **goto** statement can be used to transfer the control to the beginning of the program to read further input data. Another use of **goto** is in situations when control is to be transferred out of a loop for peculiar conditions. However, one should try to avoid the use of **goto** statement in programs. This is because the programs become unreadable and difficult to debug when goto is used. Unconditional goto statements can cause a program to go into an infinite loop and execute forever.

4.8 QUESTIONS FOR SELF STUDY

1. Explain the following with the flowchart :

a) The if-else statement.

b) The switch statement.

2. Write short notes on :

a) The conditional operator in C

b) The goto statement.

c) Use of logical operators with if statement.

3. Which are the decision making statements in C? Compare the multiple if-else statement and the switch statement.

4. List the hierarchy of arithmetic relational and logical operators in C starting from the highest.

4.9 SUGGESTED READINGS

Let us C : Yashwant kanitkar

The Spirit of C : Mullish cooper

Programming in ANSI C : Balguruswamy

CHAPTER 5

LOOPS

5.0 OBJECTIVES

Friends,

after study this lesson you will be able to-

- state the meaning of loops
- explain the loop constructs in C - while, do-while and for
- use loop nesting
- use break and continue statements in loops
- write much more sophisticated and interesting programs in C.

5.1 INTRODUCTION

Uptil now, we have seen how to use the sequential and decision control structure to write programs. In this chapter, we shall see the **loop control** structures in C. Loops are used in order to execute an instruction or a set of instructions repeatedly until a specific condition is met.

Thus, in looping, a sequence of statements will be executed until some condition for the termination of the loop is satisfied or they will be executed for a specified number of times. The first one is variable loop and the second from of loops is the fixed loop.

A program loop has two segments :

- the body of the loop (It is the set of statements which are to be executed till the condition is satisfied)

- the control statement

(The statement which will check the conditions and direct the program to execute the body till the condition is true)

Fig. 1 illustrates the loop control structure.

In the logic depicted in Fig.1 a, the test condition is first checked and then the body of the loop is executed if the condition is true. If the condition is not true the body

of the loop will not be executed. Such a loop is called as **entry controlled loop.** In Fig. 1b, the body of the loop is executed without checking the condition for the first time. Then, the test condition is checked. Such a structure is called as **exit controlled loop.** Thus, in an exit controlled loop, the body of the loop will be executed at least once (even if the test condition is false for the first time).

It is important that we state our test conditions for the loops with great care. The loop should perform the desired number of executions only and then transfer control outside the loop. If by mistake, we give an erroneous test condition, the body of the loop may be executed over and over again infinite number of times. The control may not be transferred out of the loops. Such situations set up infinite loops.

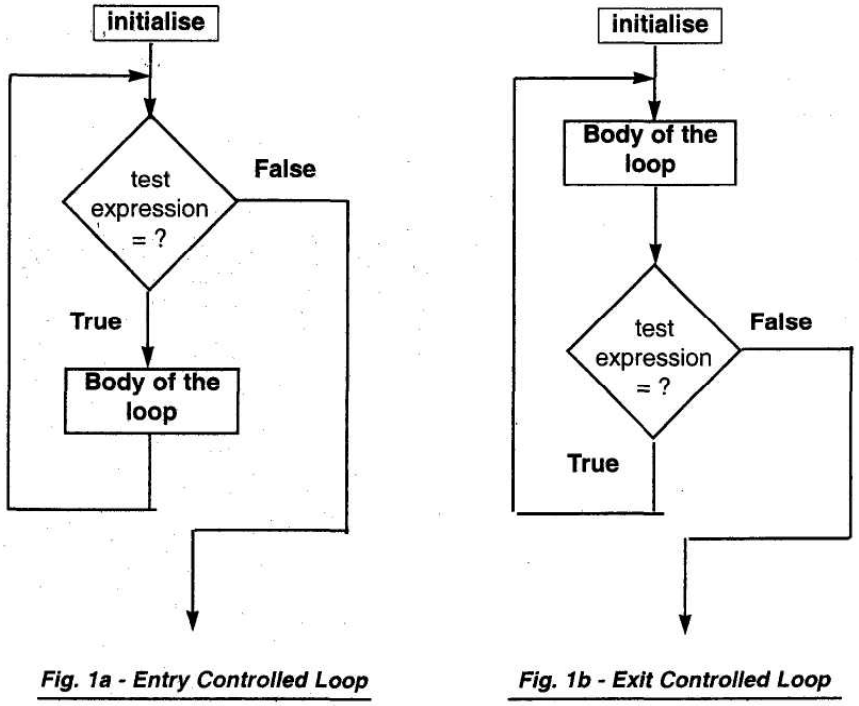


Fig. 1a - Entry Controlled Loop

Fig. 1b - Exit Controlled Loop

Fig. 1 - Flowchart for entry controlled and exit controlled loops

C language provides three types of loop constructs to repeat the body of the loop a specified number of times or until a particular condition is met. They are :

- The **while** statement
- The **do-while** statement
- The **for** statement

5.2 THE WHILE STATEMENT

The general form of the while statement is

```
while (test condition)
{
        body of the loop;
}
```

The **while** construct starts with the **while** keyword followed by the test condition in the parenthesis. The body of the loop is included in the pair of braces. (The pair of braces is not required if there is only one statement within the loop body. If there are more than one statements, the braces are essential. It is a good idea to have the braces anyway).

When the **while** keyword is encountered the test condition is checked. (This

means that the while loop is an entry controlled loop construct). If it evaluates to true, the body of the loop is executed. Then the control again goes back to **while.** The condition is checked and again the body of the loop executed if true. Thus, the process repeats until the condition in the while loop evaluates to true. When the condition finally becomes false, the body of the loop is skipped and control is transferred to the statement which immediately follows the body of the loop.

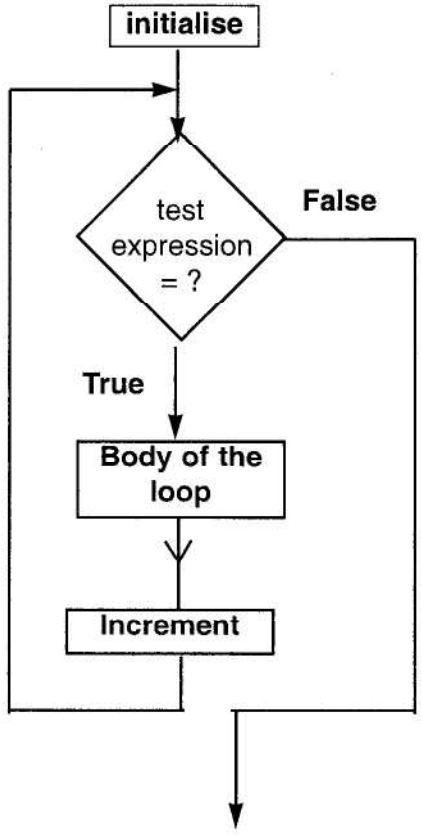


Fig. 2 - Flowchart for while loop

The following example will illustrate the use of the **while** statement:

Example:

```
/* Program to calculate the sum of first ten numbers */
main()
{
        int i, sum;
        sum = 0;
        i = 1;
        while(i<=10)
        {
                sum = sum + i;
                 i = i ++;
        }
```

```
        printf("\nThe sum is %d", sum);
}
```

The output of the program :

The sum is 55

The flowchart of the above program is depicted in Fig.2

This program calculates the sum of the first ten numbers. Thus it will get executed for the value of the variable i = 1, 2, 3, ... 10. When i becomes 11, the while condition will become false and the body of the loop will not be executed. The control will be transferred to the **printf** statement.

From the above example you can see that the **while** loop is useful in situations where you want to execute a set of instructions for a specified number of times. Thus a **while** loop in general will include the following steps :

- Initialise the counter (to execute the instructions for a specified number of times)
- Check the test condition
- Execute the body of the loop
- Increment the counter and again go back to checking the condition

The test condition in a while loop may use **relational** and **logical** operators. The while loop must have a test condition in such a way that finally it has to become false at some point, else it will fall in an infinite loop.

It may so happen that erroneous coding and initialisation results in a program falling in a situation where it will never come out of a loop. The following code is an example which will make the program fall in an infinite loop :

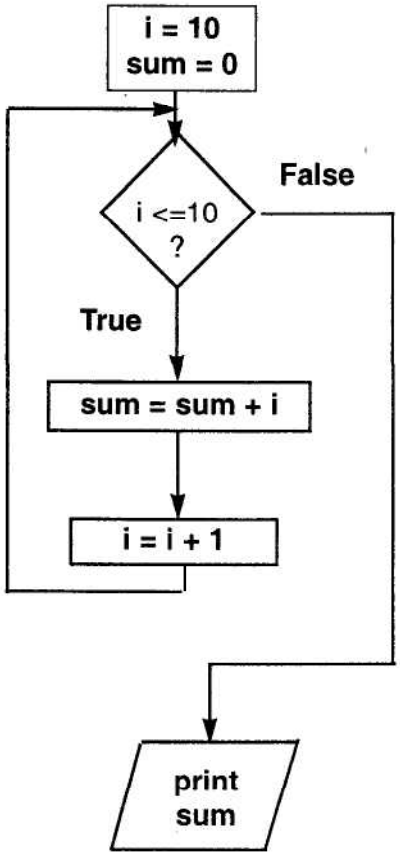


Fig. 2 - Flowchart to compute sum of first 10 numbers

Example:

```
main()
{
        int i;
        while (i<=10)
{
                i =1
                printf("\nThe value of i %d", i);
                i = i ++;
        }
}
```

Every time the program enters the body of the loop, it will reset the counter i to 1. Therefore the test condition will always be true and the program control will never fall out of the loop.

When using the counter, you can not only increment it, but also decrement it. Further the counter need not be an **integer** type only. It can also be **real,** Let us rewrite the above program by decrementing the counter:

Example:

```
/* Program to calculate the sum of first ten numbers */
main()
{
        int i, sum;
        sum = 0;
        i = 10;
        while( i>=1)
        {
                sum = sum + i;
                i = i-1;
        }
        printf("The sum is %d", sum);
}
```

Here we initialise the counter variable i to 10 and decrement it in the body of the loop. The loop gets executed till the value of i becomes 1 (i.e. decrements from 10 to 1)

In all the above examples we can make use of the increment/decrement operators in place of statement like i = i + 1; or i = i - 1. We have studied these operators in our previous chapters. To revise, let us see them again here :

C has **++** as the **increment** and **-** as the **decrement** operator. The **increment** operator adds 1 to the operand and the **decrement** operator subtracts 1 frorn the operand. Both are unary operators.

Form of the Increment operator:

```
++a ;
```

or

a++;

which is equivalent to

a = a + 1;

Form of the decrement operator:

--a;

or a--

which is equivalent to

a = a - 1;

Example :

The following example will illustrate the use of the **while** loop to read a character and print it until the escape sequence "\n" is encountered :

```
/*Program to illustrate the use of getchar() and putchar() in a while loop*/
#include "stdio.h"
main()
{
char ch 1;
ch1 = getchar();            /* Read a character*/
while(ch1 != "\n");
{
        ch1 = getchar();
        putchar(ch1);
        }
}
```

The Odd Loop : (Variable Loop)

We have used the while loop so far to write programs where we knew the number of times the loop was to be executed. But in actual practice, there may be numerous situations where it is not known in advance how many times the loop is to be executed. Such programming situation is demonstrated with the help of the following example :

Example :

```
/*Program to demonstrate the odd loop */
main()
{
        char ans = 'Y', ch1;
        while(ans == 'Y')
        {
                ch1 = getche();
                printf("\nYou entered : %c", ch1);
                printf("Do you want to continue (Y/N) ?");
                scanf("%c", &ans);
        }
}
```

In this program, you read a character from the keyboard using the **getche()** function and print the same on the screen. You then prompt the user whether he wishes to input another character. If the user enters 'Y', the while loop gets executed again. This process will continue till you keep entering a 'Y' to enter more characters. The moment you enter any character other than 'Y', the loop will terminate. Thus, it is possible to execute such while loops as many times as desired.

5.1 & 5.2 Check Your Progress

1. What will be the difference in the output, if any, in the examples (i) and (ii) in each of the following :

a) (i)
```
int a = 1;
while(a==1)
{
    a = a - 1;
    printf("\n%c", a);
}
```
(ii)
```
int a = 1;
while(a==1)
    a = a -1;
printf("\n%d", a);
```

b) (i)
```
int a = 2;
while(a >=1)
{
    a = a -1;
    printf("\n%d", a);
}
```
(ii)
```
int a = 2;
while(a >= 1)
    a = a -1;
printf("\n%d", a);
```

2. Which of the following statements are valid :

a) c = a ++ - b;

b) while(i = 10)

c) while(p <=q)

d) while(z!=10)

e) while(i > 10 && (j < 50 || k> 20))

3. Fill in the blanks :

a) In an ..... controlled loop, the body of the loop will be executed at least once.

b) A program loop has two segments ..... and .....

c) The loop constructs which C provides are ....., ... and .....

d) The ..... loop is used where it is not known in advance how many times the loop is to be executed.

5.3 THE DO-WHILE STATEMENT

As we have seen in the previous section, the **while** loop checks the test condition before executing the body of the loop i.e. it is an entry controlled loop. Hence, if the test condition is false for the first time itself the loop may not get executed at all.

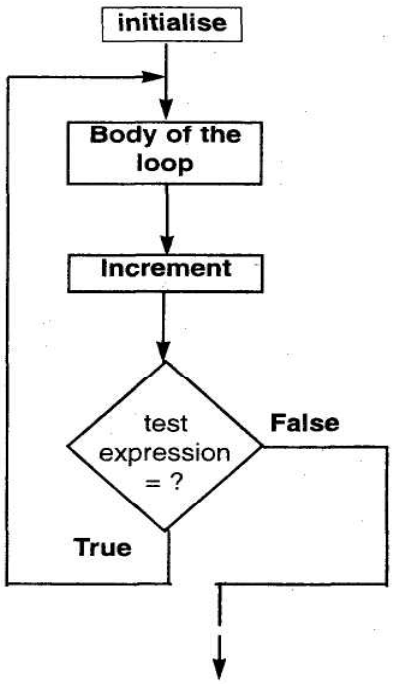


Fig. 3 - Flowchart for do - while loop

Another form of loop control is the **do-while.** This loop construct is an exit controlled loop i.e. it checks the test condition after executing the loop body. The body of the loop thus gets executed at least once in a **do-while** loop. This is the only difference between the while and do-while. Otherwise, while and do-while behave exactly in the same way. There are very few programming situations in actual practice where the **do-while** loop is required.

The general form of the **do-while** is :

```
do
{
        body of the loop
}
while(test condition);
```

When the program encounters the do statement, it executes the body or the loop first. It then checks the test condition and if true transfers the control back to the first statement in the loop body. This process continues till the test condition is true. When the condition becomes false, the subsequent statement is executed. Fig3. shows the flowchart for the **do-while** construct.

We shall see the use of **do-while** with the following example :

Example:

```
/* Program to illustrate the use of the do-while*/
main()
{
        int i, prod;
        i=1
        prod=1;
```

```
        do
{
                prod = prod * i;
                i++;
        }
        while(i<=10);
        printf("The product of the first 10 numbers is %d", prod);
}
```

This program calculates the product of the first ten numbers 1,2,3... 10.

5.3.1 More about while and do-while :

- In the **While** and **do-while** loop constructs multiple expressions can be combined with the help of logical operators.

eg. while((i<=10)&&(a>b))

Here the while loop will be executed only if both the expressions are true since they are connected by the logical AND operator.
A few other examples :
while ((a< = b) || (z > 0))
while ((a< b) || (b > c) || (c >d)

- Nesting of while loops :

It is also possible to nest one while loop inside another. Let us see how to do this with the help of the following example :

Example : The program illustrates nested **while**

```
main()
{
        int i, j;
        i = 1;
        j = 1
        while (i <=5)
        {
                j = 1;
                while (j <=5)
                {
                        printf("*");
                        j=j + 1;
                }
                printf("\n");
                i = i + 1;
        }
}
```

The output of the program is :

```
*****
*****
*****
*****
*****
```

This program contains two nested **do-while** loops. The inner dowhile executes five times for each iteration of the outer while. Note that when the inner loop is

executing it prints one * each time on the same line. After j becomes 6 this loop exits, and the **printf** takes the cursor to the new line. i is incremented and the inner loop again gets executed five times. Carefully follow the program through each step.

5.3 Check Your Progress.

1. Answer the following :

a) What is the difference between the **while** and the **do-while?**

.....

.....

b) What is meant by nesting of loops **?**

.....

.....

c) How will you combine multiple expressions in a while loop **?**

.....

.....

2. Determine how many times each of the following loops will be executed.

a)
```
x = 5;
while (x<= 10)
{
      printf("%d", x);
}
```

.....

.....

b)
```
i = 1;
do
{
      printf("\nExample of do-while");
      i = i + 1;
}
while (i <=1);
```

.....

.....

c)
```
i = 1;
while (i <=1)
{
      printf("\nExample of while");
      i = i + 1;
}
```

.....

.....

3. Write a program to find a^b **by making use of while, (i.e a x a xa .. b times)**

.....

.....

5.4 THE FOR STATEMENT

The **for** statement is probably the most frequently use loop construct. The for statement provides initialisation of counter, test condition and counter increment all in a single line. The general form of the for statement is :

```
for(initialisation; test-condition; increment)
{
        body of the loop;
}
```

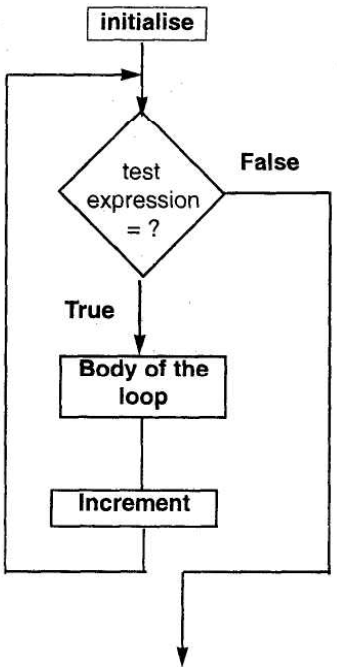


Fig. 4 - Flowchart of for loop

We specify the following things in the parenthesis following the **for** keyword :

- First is the initialisation of the loop counter. This is done with the use of assignment operators as :

i = 1 or k = 1 or count = 0

- Second is the test condition which determines when the loop will exit. The body of the loop is executed till the test condition is true. When it becomes false, the loop is exited.

- Third is incrementing the value of the loop control variable (counter) after the loop body has be executed. The new value of the counter is again checked to see if it satisfies the loop condition.

Fig. 4 shows the flowchart for the **for** loop :

The for loop is an **entry controlled** loop construct. It first checks the test condition and if it is true only then it executes the loop body. We shall see how to use the **for** loop with the help of this example :

Example :

```
/* Program to illustrate the use of the for loop */
main()
```

```
{
        int i;
        for(i = 1; i <=5; i++)
        {
        printf("The value of i %d\n", i);
        }
}
```

The program prints the values of i from 1 to 5.

It is clear from the above, that the value of the **loop counter** also called the **control variable** is initialised to 1 in the for loop. The test condition (i <= 5) is then checked. If it is true, the loop body is executed. The counter is then incremented and the test condition checked again. This sequence continues till the test condition is true. When it becomes false the loop body is exited. Note that the statements enclosed within the parenthesis are separated by semicolons. There is no semicolon after the increment statement and after the **for** statement.

5.4.1 More about for loops :

- As in the case of **while,** we can not only increment but also decrement the value of the control variable in the **for** statement.

- You can also omit one or more sections of the for loop. eg.

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

Here the initialisation of the counter variable is done before the **for** statement. Also the counter is incremented within the **for** loop and not in the for statement. This is allowed. **However, the semicolons are a must although there are no statements of initialisation and counter increment in the for statement itself.** Be careful to use them if you are trying to initialise or increment the counter outside of the for statement. Note that if you do not set any test condition the for statement will fall in an infinite loop.

- You can also initialise more than one variable in a for statement as follows :

```
k=1;
for (i = 1; i<=10 ; i++)
{
body of the loop
}
```

can also be written as

```
for(k=1, i=1; k=10, i++)
{
body of loop
}
```

Note that the initialisation section initialises the variable k and i. These variables are separated by a **comma.**

- The increment section can also have more than one part where each part will

be separated by a comma. eg.

for(i = 1; i<=10; i++, p=p+1)

is a valid for statement.

- It is also possible to increment and compare the counter in the same statement eg. for (i = 0; ++i <=10;)

Here the counter is incremented and compared in the same statement ++i <=10. Since the increment operator has a higher priority the counter will first be incremented and then compared. Remember it is necessary to initialise i to 0. Also take a note of the semicolon after ++i <= 10.

-The test condition need not be limited only to the loop control variable. It can be a compound relation combining two or more arithmetic relations with logical relations eg. for(i = 1; k=10 && sum <100;i++)

This loop will check both the test conditions (i <=10 and sum <100). If both are true then the for loop will be executed. (Refer the truth table of logical operators in the previous chapter).

Before proceeding to the next section carefully study all the above variations of the for loop and practice them with various examples.

5.4.2 Nesting of for loops :

We have seen how **if** statements and **while** statements can be nested. Similarly it is possible to nest **for** statements. The nesting of loops can be best demonstrated with the help of an example :

Example : We shall use the same example of printing * which we have used for demonstrating the nested while loop.

```
/* Program to illustrate nesting of for loops */
main()
{
int i, j.sum;
for(i=1 ;i<=5;i++)
{
        for(j=1 ;j<=5;j++)
        {
        printf("*");
        }
        printf("\n");
}
}
```

The output of this program will be :

```
*****
*****
*****
*****
*****
```

For each value of i (from i = 1 to i = 5) the inner loop (for j = 1 to j = 5) will be executed five times. The variable j will take values from 1 to 5. When j is incremented next time, its value becomes 6 and the inner for loop is exited. The printf("\n") will take the cursor to the next line. Then the value of i (the control variable of the outer loop) will be incremented. If the test condition is true, the program will enter the body of the loop. The variable j will again be initialised to 1 and the loop will be executed five times. This process will continue till the test condition becomes false.

5.4 Check Your Progress.

1. Correct the following statements and rewrite :

a) (for i == 1, i < 10, i++);

.....

b) for (count = 0; count <= 5 && k > 2; count++;)

.....

c) i = 0;
for (i < 5, i++)

.....

2. Write the following program using for loop to generate the following output:

*****

****

***

**

*

3. Write a program to print the sum of the first five odd numbers using for loop.

4. Write a program to print the first five odd numbers after any number input from the keyboard.

5.5 THE BREAK STATEMENT

In some situations it may be required to jump out of the loop without going back to the test conditions. For example, if you are checking a list of names and at the first occurrence of a particular name you wish to exit the for loop or as soon as you encounter the first even number in a list you wish to exit the for loop. In such situations we make use of the **break.** We have seen the use of the **break** keyword when we studied the **case** construct. **break** works in the same way in **for** and **while** loops as it works in the **case** construct.

When the **break** statement is encountered in a loop, the loop is exited and the control is transferred to the statement immediately following the loop.

The use of the **break** keyword for various loop constructs is illustrated below :

with the while statement as:

```
while(test condition)
{
        --------
        --------
        if(condition)
        break;
        --------
        --------
}
statement-a;
```

with a do statement as :

```
do
{
        --------
        --------
        if(condition)
        break;
        --------
        --------
}while(--------)
statement-a;
```

with the for statement as :

```
for (--------)
{
        --------
        --------
        if(condition)
        break;
        --------
        --------
}
statement-a;
```

In the process of executing the loop in each of the above situations, when the **break** is encountered the loop is exited. Thus in the above, if the condition in the parenthesis after if is true, the loop is exited and control is transferred to statement-a.

Let us use the break statement to write a program

Example : To illustrate the use of break

```
main()
{
        char ch;
        ch = getchar();
        while(ch != 'N')
        {
                if(ch='z')
                break;
                printf("%c", char);
                ch = getchar();
        }
}
```

The program will read a character from the keyboard till the user inputs 'N'. Within the loop body if the character read is 'z' it will break from the while loop. Study the program carefully and rewrite for various conditions.

5.6 THE CONTINUE STATEMENT

The **continue** statement passes the control to the beginning of the loop. Thus when a **continue** is encountered in a loop, the following statements inside the body of the loop are skipped and control is transferred to the beginning of the loop for the next iteration.

Thus, the difference between the **break** and the **continue** is that break causes the loop to be terminated whereas continue causes the following statements to be skipped and continue with the next iteration.

The use of **continue** in loops is illustrated below :

in while

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

in the do-while

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

in the for loop

```
for(initialisation;test condition;increment)
{
_____________
_____________
if(condition)
continue;
_____________
_____________
}
```

In the **while** and **do while** when the **continue** statement is encountered the remaining statements in the loop body are skipped and the test condition is checked. In the case of **for** statement, when the **continue** statement is encountered the remaining statements in the body of the loop are skipped, the counter is incremented and then the test condition is checked.

Let us see the use of the continue :

Example :

```
main()
{
        int num = 1;
        while(num != 0)
        {
                printf("\nEnter number:");
```

```
            scanf("%d", &num);
            if(num%2 == 0)
                continue;
            printf("\nNumber is %d", num);
        }
    }
```

A sample output:

```
Enter number: 2
Enter number: 3
Number is 3
Enter number: 7
Number is 7
Enter number: 0
```

If a number entered is even then the remaining part of the loop is skipped with the **continue** statement. On the other hand, if the number is odd the number is printed. The user will be inputting numbers till he enters a 0. Upon entering a 0, the program ends.

5.5 & 5.6 Check Your Progress.

1. Write in brief about 2/3 lines

a) The **break** statement:

.....

.....

b) The **continue** statement:

.....

.....

1. Select the correct option :

a. The break statement is used to exit from :

(i) an if statement
(ii) a for loop
(iii) main()
(iv) none of the above

b. The continue statement :

(i) continues program execution from the first line outside the loop construct
(ii) passes control to the beginning of the loop
(iii) cannot be used in a loop
(iv) is a substitute for break

c. A while statement :

(i) executes at least once
(ii) executes only if the condition is true
(iii) is exactly similar to the do-while
(iv) is an exit controlled loop construct

d. In a for loop the statements in the bracket are:

(i) separated by commas
(ii) are separated by a semicolon
(iii) Multiple statements cannot be written at all
(iv) None of the above

5.7 SUMMARY

In this chapter we have seen different types of loop control structure.

Loops are used in order to execute an instruction or a set of instructions repeatedly until a specific condition is met. In looping a sequence of statements will be executed until some condition for the termination of the loop is satisfied or for a specified number of times. A program loop has two segments :

C language provides three types of loop constructs to repeat the body of the loop a specified number of times or until a particular condition is met. They are :

- The while statement
- The do-while statement
- The for statement

The for loop and while loop are the example entry controlled loop. It first checks the test condition and if it is true only then it executes the loop body. Whereas do while loop is the example of exit controlled loop. This loop constructs checks condition after executing the loop body.

5.8 CHECK YOUR PROGRESS - ANSWERS

5.1 & 5.2

1. a) Output of (i) & (ii) is same i.e. 0.

b) Output of (i) is 1, 0 and output of (ii) is 0.

2. a) Invalid

b) Invalid

c) Valid

d) Valid

e) Valid

3. a) exit

b) the body of the loop, the control statement

c) while, do-while, for

d) odd

5.3

1. a) The while loop checks the test condition before executing the body of the loop i.e. it is an entry controlled loop. Hence, if the test condition is false for the first time itself the loop may not get executed at all. The do-while loop construct is an exit controlled loop i.e. it checks the test condition after executing the loop body. The body of the loop thus gets executed at least once in a do-while loop. This is the only difference between the while and do-while.

b) It is possible to nest one loop construct in another while or do-while loop construct. In such a situation, the inner loop gets executed the specified number of times for every iteration of the outer loop. Thus if an inner loop is to be executed five times and the outer loop thrice, then for each iteration of the outer loop the inner loop gets executed five times. The outer loop itself gets executed three times.

c) In the while and do-while loop constructs multiple expressions can be combined with the help of logical operators. The logical operators are AND, OR and NOT. The loop gets executed depending upon the evaluation of the expression connected

by the logical operators. The program becomes easy to read and more structured when the logical operators are used to combine multiple expression in one loop construct.

2. a) Infinite

b) One

c) One

3.
```
main()
{
    int a, b, prod, count;
    printf("Enter value of a :");
    scanf("%d", &a);
    printf("Enter value of b :");
    scanf("%d", &b);
    count = 1;
    prod = 1;
    while(count <= b)
    {
    prod = prod * a;
    count = count + 1;
    }
    printf("\na raised to b = %d", prod);
}
```

5.4

1. a) (for i = 1; i < 10; i++)

b) for (count = 0; count <= 5 && k > 2; count++)

c) i = 0;
   for (; i < 5; i++)

2.
```
main()
{
    Int i, j;
    for(i = 5; i >= 0; i—)
    {
    for(j = 0; j <= i; j ++)
    printf("*")
    printf("\n");
    }
}
```

3.
```
main()
{
    int i, sum;
    sum = 0;
    for(i = 1; i<= 10; i = i + 2)
    sum = sum + i;
```

```
        printf("The sum of first five odd numbers is : %d, sum);
}
```

4.

```
main()
{
        int num, i, sum;
        printf("Enter a number:'');
        scanf("%d'', &num);
        sum = 0;
        if (num % 2 != 0)
        {
        for(i = num ; i < num + 10; i = i + 2)
        sum = sum + i;
        }
        else
        {
        for(i = num + 1; i <= num + 10; i = i + 2)
        sum = sum + i;
        }
        printf("The sum is : %d'', sum);
}
```

5.5 & 5.6

1. a) The break statement : In some situations it may be required to jump out of the loop without going back to the test conditions. In such situations we make use of the break. break works in the same way in for and while loops as it works in the case construct. When the break statement is encountered in a loop, the loop is exited and the control is transferred to the statement immediately following the loop.

b) The continue statement : This statement passes the control to the beginning of the loop. Thus when a continue is encountered in a loop, the following statements inside the body of the loop and skipped and control is transferred to the beginning of the loop for the next iteration. In the while and do while when the continue statement is encountered the remaining statements in the loop body and skipped and the test condition is checked. In the case of for statement, when the continue statement is encountered the remaining statements in the body of the loop are skipped, the counter is incremented and then the test condition is checked.

2. a) a for loop

b) Passes control to the beginning of the loop.

c) executes only if the condition is true.

d) are separated by a semicolon

5.9 QUESTIONS FOR SELF STUDY

1. Write short notes on :

a) Additional features of the for statement.

b) Entry controlled and exit controlled loops.

c) The break and continue statements.

2. **Write a program in C** to find the first five numbers divisible by 7 between 101 to 200 and break as soon the numbers are found.

3. **Write a program** to find Armstrong numbers between 1 to 1000. A number is an Armstrong number if the sum of the cubes of each digit of the number is the number itself. eg. 153 = (1 * 1 * 1) + (5 * 5 * 5) + (3 * 3 * 3)

4. **Write a program** to print ascii values and their corresponding characters from 0 -255 making use of the while construct.

5. **Explain in detail with the help of a flowchart :**

   a) The do-while loop construct

   b) The for construct

5.10 SUGGESTED READINGS

Programming in ANSI C : Balguruswamy

Exploring C : Yashwant Kanitkar

Programming in ANSI C : Balguruswamy

CHAPTER 6

ARRAYS AND STRINGS

6.0 OBJECTIVES

Friends, After studying this chapter you will be able to

- discuss how to group related data items together with the use of arrays.
- explain single dimension, two dimensional and multi-dimensional arrays
- state what are strings
- use various standard string library functions and in programs
- create a two dimensional array of strings
- develop a number of programs using arrays.

6.1 INTRODUCTION

In this chapter, we shall study what is an array, what are the various types of arrays and what are strings. We shall learn how to define an array, what are array elements, in what way can you access array elements etc.

What is an array ?

An **array** is a group of related data items which share a **common name.** For example, we can define an array name to represent percentages of 100 students, or salaries of 1000 employees etc. Here the quantities (data items) must be similar. The complete set of values of such similar quantities is called an **array,** whereas each indivisual value in the array is called an **element.** Array elements could be **int, float, char** etc. In fact, arrays can be of any variable type. Array elements are stored in contiguous memory locations.

Thus arrays enable us to represent a collection of similar data items. Each indivisual element in an array (array element) is referred to by its **position** in the group. A particular value is indicated by writing a number in brackets after the array name. This number is called as the **index number** or the **subscript.**

eg. sal[3].

Here the array **name** is *sal* which is the array to represent salaries of employees.

3 is the subscript. Thus, sal[3] will represent the salary of the 4th employee.

(Remember that counting of elements begins with 0 and not 1 in the array. Thus sal[0] represents the salary of the first employee, sal[1] of the second and so on).

6.2 ONE DIMENSIONAL ARRAY

A one dimensional array has **only one subscript.** Thus a list of items which is given one variable name and uses only one subscript is called a single subscripted variable or a **one dimensional array.**

Array Declaration :

The array has to be declared before it can be used in a program. The general form of declaration of an array is :

type variable_name[size]

The ***type*** specifies the type of the elements that are to be stored in the array like **int, float** etc. The ***variable_name*** is any valid variable name in C. The **size** indicates the maximum number of elements that can be stored in the array.

eg. float per[50];

Here the array name is per, the array will hold elements of type **float** and the maximum number of elements that it can store is 50. Note that the array elements are stored in contiguous memory locations, i.e one after the other.

Another example of array declaration :

int marks[5];

This statement will declare an array of type **int** whose name is marks and which can store upto 5 values.

Assigning values to array elements :

Let us consider the array **int** marks[5]. The values to the elements of this array can be assigned as follows :

marks[0] = 82;

marks[1] = 50;

marks[2] = 75;

marks[3]= 68;

marks[4] = 90;

Accessing the elements of an array : In the above declared array the individual elements can be accessed by writing the subscript in the brackets after the array name. Thus marks[0] is the first element of the array, marks[1] is the second and so on.

Entering data into array elements :

We can also enter data into array elements as the following code illustrates :

```
for (i=0; i<10; i++)
{
        scanf("%d", &marks[i]);
}
```

This code makes use of **scanf** to read elements into the array marks one by one. Remember that the **for** loop should start with a 0, since the array subscripts begin with 0. The for loop will execute until the value of i becomes 9.

It is also important to note here that C performs no bounds checking on arrays, i.e. there is no check on whether the subscript has exceeded the size of the array. Therefore it is our responsibility to ensure that we do not attempt to enter data exceeding the bound of the array, otherwise this data will be simply placed in memory outside the array and the results might be unpredictable.

Let us write a small program to see the use of one dimensional arrays:

Example :

```
/* Program to illustrate the use of array */
main()

{
        int num[5];
        int sum, i;
        float avg;
        for(i =0; i< 5;i++)
                scanf("%d", &num[i]);
        sum = 0;
        for(i=0; i< 5; i++)
                sum = sum + num[i];
        avg = sum/5;
        printf("The average of numbers is %7.2'', avg);
}
```

This example reads 5 elements in an array num which is of type **int** and then calculates the average of these five elements. With the **for** loop we are inputting elements in the array. These values will be stored as individual array elements num[0], num[1] etc. The second **for** loop is used to access the indivisual elements of num one by one add them and then calculate the average. The average of the elements is then printed with **printf.** Here we have used **i** as the ***subscript variable.*** This variable takes different values and it is used to access different elements in the array. The second for loop which adds the values of the array element to the sum reads the array elements one by one starting with i=0. So the first element read is num[0].

Let us summarise what we have learnt about arrays till now :

- An array is a collection of similar elements

- The individual values in the array are called as elements

- The first element in the array is numbered 0

- Each individual element in an array is accessed by its position in the array. This position is called the **array subscript** or **index number.**

- The array has to be declared for type and dimension before it can be used

- All the elements in an array are stored in contiguous memory locations

Initialisation of arrays :

In the previous example we saw how to input values as array elements. We can also initialise array elements in the same way as we initialise ordinary variables at the time of its declaration.

The general form of initialisation of elements in an array is :

type array_name[size] = {val1, val2,....};

eg. int marks[5] = {40, 70, 32, 80, 76};

The above example declares an array **marks** which gets initialised as follows :

marks[0] = 40

marks[1] = 70

marks[2] = 32

marks[3] = 80

marks[4]= 76

Note that while initialising the array elements the individual values in the list should be separated be commas. The declaration statement itself should be terminated by a semicolon.

The size of the array may be omitted while initialising the array elements. The compiler will automatically allocate space for all the elements in the initialisation list.

eg. float num[ ] = {43.8, 56.87, -12.98, 33.33, - 89.023};

is a valid example of array intialisation. It will automatically allocate enough memory to the marks array to contain five elements of type **float.**

An array of type char can be declared as :

char name[ ] - {'A', 'E', 'I', 'O', 'U');

Note however, that while initialising an array at the time of declaring it there is no convenient way to initialise only selected elements. We have to initialise all the elements of the array.

When an array is declared, all the elements in the array are stored in contiguous memory locations.

eg. int num[5];

will immediately allocated 10 bytes of memory to the array. This is because each element occupies 2 bytes of memory since it is of type **int.**

The memory map for an array marks[5] of type **int** which has been initialised as given below :

int marks[5] = {10, 20, 5, 2, 8};

could be depicted as shown in fig 1 :

Assuming that the first element of the array marks[5] is stored at memory location

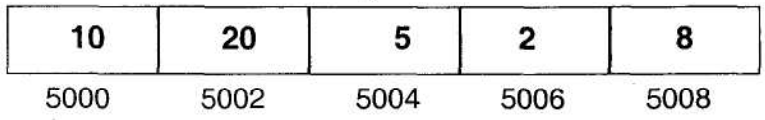


Fig. 1 Locations of array elements in memory

5000, the second element will be stored at 5002, the third at 5004 and so on. This is so because all array elements are stored in contiguous memory locations, and since the data type **int** occupies 2 bytes of memory, each element will be allocated 2 bytes.

If the array elements are not being initialised at the point of their declaration they contain garbage values in the beginning. You can indivisually assign a value 0 to every element using a loop :

for(i = 0; i< 10; i++)

mark[i] = 0;

Since the storage class of the array by default is of type **auto** the elements of the array are not initialised to zero at the time of declaration. If the storage class is declared to be **static** all the elements will have default initial value of 0. More about storage classes will be discussed later in the next chapter. Therefore if you wish to assign default initial values of array elements as 0 you may declare it as follows :

static int mark[10];

At this point, you may only remember that if you do not declare the storage class of an array, the elements will contain garbage values till specific values are assigned to them. One more thing about arrays is that you cannot assign one array directly to another like an ordinary variable, i.e. if arr1 and arr2 are two arrays then :

arr1 = arr2 is not valid in C.

You have to explicitly assign individual elements of one array to the corresponding elements of the other array.

Let us now write some programs to demonstrate the use of one dimensional arrays:

Example 1 :

```
/*Program to input numbers into array a and then copy them to array b */
main()

{
        int i, a[10], b[10];
        for (i = 0; i< 10; i++)

        {
                printf("Enter element:);
                scanf("%d", & a[i]);
                printf("\n");

        }

        for (i = 0; i< 10; i++)
                b[i] = a[i];
        for (i = 0; i< 10; i++)
                printf("%d\t%d\n", a[i], b[i]);

        }
```

In this example, we have entered 10 elements in array a and then copied those elements one by one to array b. In the third **for** loop we have output the elements of both array a and array b with the use of **printf.** Note the use of the backslash character constant '\t'. Follow the working of the program carefully and modify the above program to enter elements in both the arrays at the same time.

Example : To find the smallest number in an array

```
/* Program to find the smallest number in an array */
#define MAX 20
main()
{
        int i, n, min, num[MAX];
        printf("\nHow many elements do you wish to enter ? :");
        scanf("%d", &n);
        printf("Enter elements of array :\n");
        for(i=0; i < n; i++)
        scanf("%d", &num[i]);
        min = num[0];
        for (i = 1; i < n; i++)
```

```
        {
        if(num[i] <min)
        min = num[i];
        }
        printf("\nThe smallest element of the array is : %d", min);
}
```

Here the number of elements in the array are also input by the user. We have made use of the **symbolic constant** MAX with the #define MAX statement to define the maximum elements that the array can store to be 25. The first element of the array num is assumed to be smallest. Then the numbers from the second element are compared with this min, and if an array element smaller than min is found, min is assigned the value of that element.

Example : The sorting algorithm to sort elements of the array

```
/* Program to sort array elements in ascending order */
main()
{
        int i, j, temp;
        float num[5];
        printf("\nEnter elements of array :");
        for (i=0; i<5; i++)
                scanf("%f", &num[i]);
        for(i = 0; i< 4; i++)
        {
                for(j = i+1; j<5; j++)
                {
                        if(num[i] > num[j])
                        {
                                temp = num[i];
                                num[i] = num[j];
                                num[j] = temp;}
                        }
                }
        }

        printf("\nThe sorted array :\n");
        for(i=0;i<5;i++)
        printf("%6.2f\n", num[i]);
}
```

Follow the program carefully. It makes use of nested loops. In the first iteration of the outer loop where the counter is 0; the remaining elements (beginning with num[1] are compared with num[0]) in the inner loop. Wherever an element smaller than num[0] is found the positions of the two elements are interchanged. Thus in the first iteration, the smallest element of the array gets stored in num[0]. Then the process repeats for i = 1 and in the second iteration the next smallest number gets stored in num[1] and thus the comparison continues. Note that the outer loop is executed 4 times only (number of array elements -1) since at the second last iteration itself the sorting gets completed. The inner loop always starts execution from the value of i+1, since the ith element is to be compared with the remaining elements.

6.1 & 6.2 Check Your Progress.

1. Fill in the blanks:

a) An array is a group of ..... data items which have a common name.

b) The individual elements of an array can be accessed by writing the ..... in the brackets after the array name.

c) If the storage class of an array is declared to be .....all the elements will have default intial value of 0.

d) While initialising the array elements the individual values in the list should be separated by .....

2. Correct the following array declarations and rewrite :

a) int arr1 == [5,3,5]

.....

b) int arr1 = {5, 3, 5},

.....

c) char ch1[] = {“A”, “B”, “C”);

.....

d) float num[0] = {40.45, 29.1. 90.21, 18.8};

.....

3. Write C programs for the following :

a) Enter elements in an array of type **int** and sort them in descending order. Assume the array contains 10 elements.

b) Enter elements into an array of type **float.** Determine how many of them are positive and how many are negative. Assume an array size of 15.

c) Modify the above program and store all the positive numbers in a separate array and all negative numbers in a different array. Print the sum of elements of these new arrays.

6.3 TWO DIMENSIONAL ARRAYS

In the previous section we have studied one dimensional arrays. It is also possible to declare arrays with two or more dimensions. A two dimensional array is also called a **matrix.** Such a matrix or a **table** can be stored in a two dimensional array, eg. we may have the following table :

Student_Id	Sub1	Sub2	Sub3
1100	40	50	67
2100	80	34	56
1330	90	98	89
1331	76	76	76
1485	80	70	65

This table shows the data for five students, where marks of three subjects of each student are listed. Thus, this table is made up of five rows and four columns. Each **row** represents the marks obtained by a particular student in all the four subjects, whereas each **column** represents the marks obtained by all the students in a particular

subject. We define tables of such type with the help of two dimensional arrays.

The two dimensional array can be declared as follows :

type array_name[row_size][col_size];

Thus the above table can be represented in a two dimensional array as :

int stud[5][4];

The elements in a two dimensional array are stored **row wise**. Each dimension of the array is indexed from 0 onwards to its maximum size minus one. The **first index** selects the **row** and the **second index** selects the **column.** Thus stud[3][1] will select the second item from the fourth row, stud[0][2] will select the third item from the first row etc. The way in which two dimensional arrays are stored in memory is shown herewith with the help of the above example :

	Column0	**Column1**	**Column2**	**Column3**
	[0][0]	[0][1]	[0][2]	[0][3]
RowO	1100	40	50	67
	[1][0]	[1][1]	[1][2]	[1][3]
Row1	2100	80	34	56
	[2][0]	[2][1]	[2][2]	[2][3]
Row2	1330	90	98	89
	[3][0]	[3][1]	[3][2]	[3][3]
Row3	1331	76	76	76
	[4][0]	[4][1]	[4][2]	[4][3]
Row4	1485	80	70	65

Note that the element of the first row and first column is stored first, then the element of the first row and second column, then the element of the first row third column and so on. When all the elements of the first row are complete, the elements of the second row are stored and in the same way for all the rows.

Initialising Two Dimensional Arrays :

Two dimensional arrays can also be initialised like the one dimensional arrays with their declaration followed by the list of values of the elements. This list is enclosed in braces, with each element being separated by a comma. The elements are initialised row wise.

For example :

int arr[2][3] = {10,5,3,15,20,25};

This initialisation will intialise the array elements of array arr as follows:

arr[0][0] = 10

arr[0][1] = 5

arr[0][2] = 3

arr[1][0] = 15

arr[1][1] = 20

arr[1][2] = 25

Thus elements will be initialised row wise. There are three elements in each row of the array.

Alternative methods for the above initialisation are :

```
int arr[2][3] = {{10, 5, 3}, {15, 20, 25}};
or
int arr[2][3] =
        {
                {10,5,3},
                {15,20,25}
        };
```

In this way each row can be separated by braces. Commas are necessary after the closing of each row except the last row. Individual elements in a row are to be separated by commas. The initialisation statement should end with a semicolon. The following method may be used to initialise all the elements of a two dimensional array to 0 :

```
int arr[2][3] = {{0},{0}, {0}};
```

This means that you use a single zero for all elements of a column. Hence, you use 3 zeroes to initialise all elements of each of the three columns.

Another important point is that when initialising a two dimensional array the first dimension i.e. the **row** is optional, however the second dimension the **column** is a must. Thus the example illustrated above can also be declared as:

```
int arr[][3] = {10,5,3,15,20,25};
```

However the declaration

```
int arr[2][] = {10,5,3,15,20,25};
```

or

```
int arr[][] = {10,5,3,15,20,25};
```

is invalid.

Let us write C programs for demonstrating how to work with 2-dimensional arrays.

Example :The following example stores elements in a 2 dimensional array.

```
/* Program to store elements in a 2 dimensional array */
#define ROWS 5
#define COLS 5
main()
{
        int arr1 [ROWS][COLS], i, j, rows, cols;
        printf("\nEnter number of rows :");
        scanf("%d",&rows);
        printf("\nEnter number of columns :");
        scanf("%d", &cols);
        for(i=0; i< rows; i++)
{
        for(j = 0; j < cols; j++)
        {
        printf("\nEnter value :");
        scanf("%d", &arr1[i][j]);
        }
}
printf("\nThe elements are");
```

```
for(i=0; i< rows; i++)
{
        for(j = 0; j < cols; j++)
        {
        printf("%d\t", arr1[i][j]);
        }
        printf("\n");
}
}
```

A sample output:

```
Enter number of rows :2
Enter number of columns :3
Enter value :10
Enter value :5
Enter value :10
Enter value :5
Enter value :10
Enter value :5
The elements are
```

10 5 10

5 10 5

The elements of array arr1[R0WS][C0LS] are entered rowwise. Note the symbolic constants ROWS and COLS. The subscript i is used for rows and j for columns. For each value of i i.e row all column values are read. The array is then output rowwise.

Example : Find the sum of elements of the rows in 2 dimensional array

```
/* Program to calculate sum of elements in rows of the array */
main()
#define ROWS 5
#define COLS 5
main()
{
        int arr1[ROWS][COLS], i, J, rows, cols, sum;
        printf("\n Enter number of rows :");
        scanf("%d",&rows);
        printf("\nEnter number of columns :");
        scanf("%d", &cols);
        for(i=0; i< rows; i++)
{
                for(j = 0; j < cols; j++)
                {
                printf("\nEnter value :");
                scanf("%d", &arr1[i][j]);
                }
}
        for(i=0; i< rows; i++)
        {
```

```
            sum = 0;
            for(j = 0; j < cols; j++)

            {
            sum = sum + arr1[i][j];
            printf("%d\t", arr1[i][j]);
            }

            printf("%d\t", sum);
            printf("\n");
        }
    }
```

The working of the program and its output is left to the student as a part of self exercise.

6.3 Check Your Progress.

1. Correct the following array declarations and rewrite :

a) int arr1[3][];

.....

b) float num[1][2] = {1}{1};

.....

c) int arr1[2][2] = {{2,2};{3,3};};

.....

2. Answer in one sentences.

a) What is a string constant ?

.....

b) Define an array.

.....

c) What is meant by a subscript ?

.....

d) What is a matrix ?

.....

e) What is meant by String concatenation ?

.....

3. Write programs in C for the following :

a) Write a program to find the sum of the elements of each column of an array of size 5 rows and 4 columns. Print the array elements in the matrix form and sum of the elements of the column under the corresponding column.

b) Input elements in two arrays arr1 and arr2 both of equal dimensions not exceeding 5 rows and 5 columns. Write a program to find the sum of the corresponding elements of both the arrays and store it in the third array. Print both the arrays and the resultant array rowwise.

6.4 MULTIDIMENSIONAL ARRAYS

C allows arrays of more than 2 dimensions. Such arrays are multidimensional arrays. The exact limit of the number of dimensions is dependant on the compiler. Multidimensional arrays are rarely required. They are discussed here for the purpose of illustration only.

The following examples declare multidimensional arrays :

int arr1 [2][3][4];

float arr2[4][3][2][2];

The first array of type integer will contain 24 elements whereas the second array will contain 48 elements of type **float.** The elements of a multidimensional array can be initialised as follows eg.

```
int arr1 [2][3][4] = {
                        {
                            {1, 1, 1, 1},
                            {2, 2, 2, 2},
                            {3, 3, 3, 3}
                        },
                        {
                            {5, 5, 5, 5},
                            {6, 6, 6, 6},
                            {7, 7, 7, 7},
                        }
                        };
```

Follow the example carefully. A three dimensional array can be considered to be a two dimensional array in another array. The outermost array is an array which has two elements where each element itself is a two dimensional array whose dimensions are [3][4]. Note the way in which commas have been given in the initialisation.

Thus the above three dimensional array can be represented as a series of two dimensional arrays as follows :

0th 2-D Array	1	1	1	1
[3][4]	2	2	2	2
	3	3	3	3
1st 2-D Array	5	5	5	5
[3][4]	6	6	6	6
	7	7	7	7

In memory array elements are stored in contiguous memory locations as ;

1 1 1 1 2 2 2 2 3 3 3 3 5 5 5 5 6 6 6 6 7 7 7 7

Thus the first element of the array can be referred to as arr1 [0][0][0]. The counting of elements in the three dimensional array also begins with 0. We can refer to the second element as arr1[0][0][1] and so on. Thus arr[1][2][1] will be 6.

Note that the above description of 3 dimensional arrays is for the purpose of introduction of multidimensional arrays only.

6.5 STRINGS

An array of characters is a **string.** Any group of characters within double quotation marks is a constant string. Thus a **string constant** is a one dimensional array of characters

eg. “Programming Techniques Using C”

Since the string is enclosed in double quotes if we wish to include double quotes in our string we can implement it in the following way with the use of the backslash (\) :

"\"Programming Techniques Using C\""

We use **printf** to print the above statement as follows :

printf("\"Programming Techniques Using C\"");

and the output will be :

"Programming Techniques Using C"

A string variable is any valid C variable name and since a string is an array of **char** we always declare a string as an array. The general form of a string variable is :

char string_name[size];

The size determines the number of characters in the string_name. eg.

char name[20];

char address[25];

A **string constant** is an array of characters terminated by a null ('\0'). A string constant can be initialised as follows :

char str1[] = {'P', 'R', 'O', 'G' 'R' 'A', 'M' '\0'};

Each character in the array occupies one byte of memory. The last character is always '\0' (null character). When declaring the character array as above, the null character is a must. The elements of a character array are also stored in contiguous memory locations.

The above character string can also be initialised as follows :

char str1[] = "PROGRAM";

In this declaration the '\0' is not required. The null character is inserted automatically.

Thus the difference in both the initialisation is that if you enter elements each as a separate character then you enclose each character in single quotes, separate the characters with a comma and terminate the string with the '*0*' In case of initialising the complete string within double quotes the '0\' is not required.

As you have seen in both the above examples, the number of characters (size of the string) need not be specified while initialising. C will automatically determine the size of the array on the basis of the number of elements initialised.

6.5.1 Reading and Writing Strings :

The **scanf** function can be used to read strings. The format specification used to read strings as has already been introduced is '**%s**'. The **scanf** can be used as follows :

Example : To read string with the help of **scanf**

```
main()
{
        char name[15];
        printf("Enter your name :");
        scanf("%s", name);
        printf("Good Morning %s", name);
}
```

In the above example, name is declared to be a character array of size 15. The value of the name is read using **scanf.** Note that in the case of the character array the

& is not required before the variable name. When you type in characters and terminate the input with the Enter key, **scanf** will automatically insert the null character at the end.

A sample output of the above program :

Enter your name :

John

Good Morning John

Points to remember when using scanf

- Take care to ensure that you do not enter more characters than the size you have defined while declaring the character array. C does not perform any bounds checking on character arrays.

- Note that **scanf** terminates its input as soon as it encounters a white space. A white space may be a blank space, a tab, carriage return, form feed, new line. This means that **scanf** is not able to receive more than one word strings. For example if you input the following string :

Hello there

in the above array name (with a space between Hello and there) only Hello will be read in the array name, the blank space after Hello will terminate the string. So, in the above program if you input the words :

John Smith to the string name then the output of the program will be

Good Morning John

(Smith will be ignored because of the blank space after John).

But C provides alternative functions to overcome this situation. C has the functions **gets** and **puts** to read and output multiword strings. The use of **gets()** and **puts()** to read multiword strings is illustrated below :

Example : Input and output multiword strings using **gets()** and **puts()**

```
/*Program to illustrate the use of gets and puts */
main()
{
        char name[25];
        printf("Enter your name :");
        gets(name);
        puts("Good Morning");
        puts(name);
}
```

A sample run

```
Enter your name : John Smith
Good Morning
John Smith
```

gets() and **puts()** are capable of handling only one string at a time. Therefore we were required to use two **puts()** in the above program to print the two strings Good Morning and John Smith. Also, note that **puts()** automatically takes the cursor on the next line. Hence the two strings are output on different lines.

Another way to read more than one word into a character array is by making use of the **getchar()** function which we have seen earlier. The program given below illustrates the use of **getchar()** to input a line of text :

Example : Program to make use of **getchar()** to read multiword input

```
#include “stdarg.h”
#include “stdio.h”

/* Program to read a line with the help of getchar()*/
main()
{
        char ch1, line_1[81];
        int i;
        printf(“Enter a line of text. Press Enter to end\n”);
        i = 0;
        while ((ch1 = getchar()) != ‘\n’)
        {
                line_1[i] = ch1;
                i = i + 1;
        }
                line_1 [i] = ‘\0’;
                i = 0;
                while((ch1 = line_1[i]) !=’\0')
                {
                        putchar(ch1);
                        i++;
        }
}
```

A sample run :

```
Enter a line of text. Press Enter to end
Program to demonstrate the use of getchar() to input a text line
Program to demonstrate the use of getchar() to input a text line
```

In this program we have declared line_1 to be a character array of size 81. This is because a single line is assumed to be 80 characters wide and the last character has to be the null character. Every time you enter a character it is checked to see whether it is a newline character. If it is not, another character is read into line_1. As soon as the Enter key is pressed the **while** loop is exited, the counter is decremented by 1 and the null character is stored at that position in the array. The string is then output using the **printf.**

Additional programs for string manipulation :

C has no provision for directly assigning one string to another. Thus the following are invalid in C :

```
string = “STRING1”;
str1 = str2;
```

Example : In order to copy contents of one string into another we have to do it character by character as the following example illustrates. It also prints the length of the string i.e number of characters in the string :

```
/*Program to copy one string into another */

main()
```

```
{
        char str1[20], str2[20];
        int i;
        printf("Enter string :");
        scanf("%s", str1);
        for(i = 0;str1[i] !='\0'; i++)
        str2[i] = str1[i];
        str2[i] = '\0';
        printf("\nThe copied string is %s", str2);
        printf("\nThe length of the string is : %d", i);
}
```

Example : The following example illustrates how to append one string at the end of the other i.e. string concatenation :

```
/* String concatenation Program */

main()

{
        char str1[50], str2[25];
        int i, j ;
        printf("\nEnter first string :");
        scanf("%s",str1);
        printf("\nEnter second string :");
        scanf("%s", str2);
        i = 0;
        while(str1 [i] !='\0')
                i = i + 1;
        for(j = 0; str2[j] != '\0';j++)
        {
                str1[i] = str2[j];
                i = i + 1;
        }
        str1[i) = '\0';
        printf('\nConcatenated string is %s", str1);
}
```

A sample run:

```
Enter first string : Good
Enter second string : Morning
Concatenated string is : GoodMorning
```

Note in the above program, that first the end of str1 is determined with the while condition. When the '\0' is encountered, the new string is appended character by character to str1 from that position. After the end of str2 is encountered the '\0' is inserted in str1 and the concatenated string is output.

Arithmetic operations on characters :

Whenever we use any character constant or character variable, it is automatically converted into integer value by the system. Each character is assigned an integer

value, generally its ASCII value. Therefore it is possible for us to manipulate characters like we manipulate numbers. eg.

```
char ch1;
ch1 = 'A';
printf("%d", ch1);
```

The above code will print the ASCII value of the character A and not the character 'A' because of the %d specifier.

Arithmetic operations can also be performed on character constants and variables:

eg. a = 'p'+. 'q';

z = 'x' - a;

are valid expressions.

Similarly character constants can be used in relational expression like :

ch1 >= 'A'

ch2 < 'Z' etc.

6.5 & 6.5.1 Check Your Progress.

1. Fill in the blanks :

a) A string is an array of .....

b) A string constant is terminated by a.....character.

c) scanf terminates reading as soon as it encounters a .....

d) ..... is used to read multiword strings.

2. Remove the errors in the following and rewrite :

a) 'a' + '100'

.....

b) 7 - "p"

.....

c) str1(10) = "Good";

.....

d) char str[10] = 'One'

.....

e) str1 = {'a', 's' d, f, g, '\0');

.....

3. Write programs in C for the following :

a) Read a string of 25 characters and copy only the first ten characters of the string into another string. Print out both the strings.

b) Determine the values of the following expressions with the help of a program :

i = 'a' + 10; where i is an int

ch = 'z' - 'c' where ch is char

ch1 = 100 - *Y* and ch2 = 100 - 'm'. Print the difference in the values of ch1 and ch2.

Incorporate all these expressions in the same program.

6.5.2 String Handling Functions :

We have written programs in the above section to perform a number of functions on strings, like copying one string to another, finding the number of characters in a string, concatenating strings etc. C provides a number of useful functions of these type for handling strings. These functions are in the common string library functions which

are supported by most C compilers. You can directly use these functions in your programs. Some of the most common string library functions supported by most C compilers include :

Function	Use
strcat()	Appends one string at the end of the other
strncat()	Appends first n characters of a string at the end of another
strcpy()	copies one string into another
strncpy()	copies the first n characters of one string into another
strcnp()	compares two strings
strncmp()	compares the first n characters of two strings
strcmpi()/	
stricmp()	compares two strings ignoring the case
strnicrnp()	compares the first n characters of two strings without regard to case
strdup()	duplicates a string
strchr()	finds the first occurence of a given character in a string
strrchr()	find last occurence of a given character in a string
strstr()	finds the first occurence of a given string in another string
strlen()	Finds the length of a string
strlwr()	converts a string to lowercase
strupr()	converts a string to uppercase
strset()	sets all the characters of a string to a given character
strnset()	sets the first n characters of a string to a given character
strrev()	reverses a string

Let us see how to make use of some of these standard string functions :

Example : strlen()

This function is useful to calculate the length of the string i.e. it counts the number of characters in a string. It returns an integer value. eg.

```
/* To find the length of a string */
main()
{
        char arr1[];
        int len;
        arr1[].= "C Programming"
        len = strlen(arr1);
        printf("\nString = %s Length of the string is %d", arr1, len);
        len = strlen("Library Functions");
printf("\nNew string = %s Length of new string is %d", "Library Functions", len);
}
```

And the output would be :

String = C Programming Length of the string is 13

New String = Library Functions Length of new string is 17

Note that while calculating the length of the string the \0 is not counted.

Example : To copy one string into another using **strcpy()**:

This function copies the contents of one string into another.

The form of the **strcpy()** is :

strcpy(target, source);

The source string gets copied into the target string. Remember that the source and target are strings i.e. character arrays. The following example will illustrate :

```
main()
{
        char first[], second[];
        first[] = “First String”;
        strcpy(second, first);
        printf(“\nString to copy %s”, first);
        printf(“\nString to which copied %s”, second);
}
```

The output of the program :

String to copy First String

String to which copied First String

The source string gets copied into the target string character by character until a null character is encountered. As C performs no bound checking on character arrays, it is our responsibility to check that the target string is big enough to hold the source string. Note the syntax of the **strcpy()** function.

Example : **strcat()**:

The **strcat()** function concatenates two strings, i.e. it appends the target string at the end of the source string. The **strcat()** function takes the following form :

```
strcat(string1, string2);
```

Here string1 and string2 are character arrays, string 2 gets appended at the end of string 1. The null character at the end of string1 is removed and the string2 is appended from that position. Note that string1 changes but string2 remains unchanged, string1 has to be declared large enough to store the concatenation. Study the following example :

```
/*Program to demonstrate concatenation */
main()
{
        char str1[], str2[10];
        str1 [] = “String Function”
        str2[] = “Concatenation”;
        strcat(str1,str2);
        printf(‘\nString1 before concatenation :%s”, str1);
        printf(“\nString2 before concatenation :%s”,str2);
        printf(“\nConcatenated String : %s”, str1);
        printf(“\nString2 is unchanged :%s”, str2);
}
```

and the output is :

String1 before concatenation :String Function

String2 before concatenation :Concatenation

Concatenated String : String Function Concatenation

String2 is unchanged :Concatenation

strcat() function can also be used to append a string constant to a string variable.

eg. strcat(str1,”Hello”);

is a valid use of **strcat().** which will append “Hello” to str1

strcat() can also be nested as :

strcat(strcat(str1 ,str2),str3);

The inner **strcat()** will concatenate str1 and str2 in str1. The outer **strcat()** will concatenate the new str1 and str3 and the final string will be stored in str1.

Example : Compare two strings using **strcmp():**

The **strcmp()** functions compares two string to check if they are equal or not. If both the strings are equal it returns a value 0. If they are not equal it returns a value which is equal to the numeric difference between the ascii values of first non matching characters.

The general form of the **strcmp()** is **:**

strcmp(string1, string2);

string1 and string2 can be string variables or string constants. eg.

strcmp("Hello", str2);

strcmp(str1, "Good Morning");

strcmp(str1 ,str2);

strcmp("Good","god");

When the two strings are compared if a nonzero value is returned it means that the strings are not identical. Since the function returns an integer value you can save it in an integer variable and also use it as follows :

```
main()
{
        char str1 [25] = "Good Morning", str2[25] = "GoodMoming";
        char str3[25] = "Good Morning";
        int i, j;
        i = strcmp(str1 ,str2);
        printf("\nResult of comparison is %d",i);
        j = strcmp(str1,str3);
        printf("\nResult of comparsion is %d", j);
}
```

A sample run :

Result of comparison is -45

Result of comparison is 0

Note that in the above example str1 and str2 are not identical. Hence the first comparison will return a non zero value. In the second comparison since str1 and str3 are identical the value of the comparison will evaluate to 0. The value returned if the strings are not identical is actually the difference between the ascii values of the first set of characters in both the strings which do not match.

Example :

To copy one string into another using **strcpy()**:

This function is used to copy to contents of one string into another. The general form of **strcpy()** is :

strcpy(string1 ,string2);

The contents of string2 will be copied to string1. string2 can be a character array variable or a string constant. The size of string1 should be large enough to hold the contents of string2. eg.

strcpy(str1, "Hello");

strcpy(str1,str2);

The following example illustrates :

```
main()
{
        char str1[40], str2[40];
        printf("\nEnter string :");
        gets(str1);
        strcpy(str2, str1);
        printf("\nString into which copied:\n");
        puts(str2);
}
```

A sample output:

Enter string : A string copy program
String into which copied :
A string copy program

Thus making use of the standard string library functions, it is possible to write many useful programs which otherwise would have to be written on character by character basis as we saw in the previous section.

6.5.2 Check Your Progress.

1. Describe the use of the following string functions :

a) strlen()

.....

b) strcat()

.....

c) strset()

.....

d) strstr()

.....

2. Without making using of the standard string library functions write the following programs in C :

a) Copy the contents of one string of size 25 characters into another with the last character as the first character in the new string. i.e. in reverse order.

b) Replace the contents of a string of size 25 with the alphabet 'a'.

3. Make use of the standard string library functions and write the following programs :

a) Enter a string and convert it into uppercase.

b) Enter a string and reverse it.

c) Enter a string and set all its characters to Z

Write separate programs in C for all the above and print the result on the screen.

6.5.3 Two Dimensional Array of Characters :

We saw the use of a character array to represent strings. Now we can extend this concept to a two dimensional array of characters. This array can be used to store a list of names, places etc.

eg. Let us use a two dimensional array of characters to represent the following list of names :

John
jack
jill
jane

jimmy

In the above list each name itself is an array of characters and all the names together can be represented as a two dimensional array of characters as follows :

Each row represents a name. Five such names are there in this table. Hence this array can be specified as

char name[5][7];

where 5 indicates that a total of five names may be stored in this array, where each name can be upto seven characters long.

Initialising a 2-dimensional array of characters :

A 2-dimensional array of characters can be initialised as follows :

```
char city[4][10] =
                    {“Pune”, “Delhi”, “Chennai”, “Mumbai”};
or
char city[4][10]=   {
                    “Pune”,
                    “Delhi”,
                    “Chennai”,
                    “Mumbai”
                    };
```

Each indivisual city name is enclosed in double quotes and individual cities are separated by a comma.

Let us use the array of names declared as above to write a sample program to check whether a name we enter as input is found in the list or not:

Example:

```
#define FOUND 1
#define NOTFOUND 0
main()
{
char name[5][7] = {“john”, “jack”, “jimmy”, “jill”, “jane”};
char str1[7];
int fl, z, i;
i = 0;
fl = NOTFOUND;
printf(“\nEnter your name :”);
scanf(“%s”, str1);
for(i =0; i<5; i++)
{
z = strcmp(&name[i][0], str1);
if(z ==0)
{
        printf(“\Congratulations! Your name is in the list”);
        fl = FOUND;
        break;
}
}
if(fl==NOTFOUND)
printf(“\nSorry! Your name is not in the list”);
```

}

A sample run of the program :

Enter your name : jill

Congratulations! Your name is in the list

Enter your name :joseph;

Sorry! Your name is not in the list

In the two dimensional array of characters declared in the above program, note that the first subscript gives the total number of names in the list and the second gives the maximum number of characters of each element in the array. The symbolic constants FOUND and NOTFOUND are set to 0 and 1 respectively. Initially a flag f1 is set to NOTFOUND to indicate that we have not yet found a match. While comparing strings, the & is used to pass the base address of the array. We have made use of the **strcmp()** function to compare the string input with each of the names in the array. This function returns a zero value if a match is found otherwise it returns a nonzero value. As soon as a match is found the flaf fl is set to FOUND. The program also illustrates the use of **break,** to exit the for loop as soon as a match is found. Follow the program carefully for a thorough understanding.

6.5.3 Check Your Progress.

1. How will you declare the following lists as two dimensional arrays of strings:

a. abc, pqr, Imn, rst, uvw

.....

.....

b. Blue, Green, Red, Yellow, Purple

.....

.....

2. Write programs in C to input the above lists in two dimensional arrays of strings and print the lists.

.....

.....

6.6 SUMMARY

In this chapter we leant about the arrays of C.

An array is a group of related data items which share a common name. The complete set of values of similar quantities is called an array, whereas each individual value in the array is called an element. Array elements could be int, float, char etc.

Array Declaration : The array has to be declared before it can be used in a program. The general form of declaration of an array is :

type variable_name[size]

Thus

- An array is a collection of similar elements

- The individual values in the array are called as elements

- The first element in the array is numbered 0

- Each individual element in an array is accessed by its position in the array. This position is called the array subscript or index number.

- The array has to be declared for type and dimension before it can be used

- All the elements in an array are stored in contiguous memory locations

Strings : An array of characters is a string. Any group of characters within double quotation marks is a constant string. Thus a string constant is a one dimensional array of characters. A string variable is any valid C variable name and since a string is an array of char we always declare a string as an array.

The general form of a string variable is :

char string_name[size];

6.7 CHECK YOUR PROGRESS - ANSWERS

6.1 & 6.2

1. a) similar
b) subscript
c) static
d) comma

2. a) int arr1 [3] = {5,3,5};
b) int arr1 [3] = {5, 3,5};
c) char ch1[3] = {'A', 'B', 'C');
d) float num[4] = {40.45, 29.1, 90.21, 18.8};

3. a)

```
main()
{
int i, j, temp;
int num[10];
printf("\nEnter elements of array :");
for (i=0; i< 10; i++)
        scanf("%d", &num[i]);
for(i = 0; i< 9; i++)
{
        for(j = i+1;j< 10; j++)
        {
                if(num[i] < num[j])
                {
                        temp = num[i];
                        num[i] = num[j];
                        num[j] = temp;
                }
        }
}
printf("\nThe sorted array :\n");
for(i=0;i<10;i++)
printf("%d\t", num[i]);
}
```

b.

```
main()
{
```

```
float arr1 [15];
int pos = 0, neg = 0, i;
printf("Enter array elements :");
for(i = 0; i<15; i++)
        scanf("%f", &arr1[i]);
for(i = 0; i < 15; i++)
{
        if(arr1[i]<0)
                neg = neg + 1;
        else
                pos = pos + 1;
}
printf("\nNumber of positive elements = %d", pos);
printf("\nNumber of negative elements = %d",neg); }
```

c.

```
main()
{
float arr1[15], pos[15], neg[15];
int i,j, k;
float pos_sum, neg_sum;
printf("Enter array elements :\n");
for(i = 0; i<15; i++)
        scanf("%f', &arr1 [i]);
i=0;
j = 0;
k = 0;
for(i = 0; i < 15; i ++)
{
        if(arr1 [i] < 0)
        {
                neg[j]=arr1[i];
                j++;
        }
        else
        {
                pos[k] = arr1 [i];
                k++;
        }
}
pos_sum = 0;
neg_sum =0;
printf("\nArray of positive elements :\n");
for(i = 0; i < k; i++)
{
        printf(''%.2f\t", pos[i]);
        pos_sum = pos_sum+.pos[i];
}
printf("\nSum of positive elements = %.2f", pos_sum);
```

```
printf("\nArray of negative elements :\n");
for(i = 0; i < j; i ++)
{
        printf("%.2f\t", neg[i]);
        neg_sum = neg_sum + neg[i];
}
printf("\nSum of negative elements = %.2f", neg_sum);
}
```

6.3

1 a) int arr1[][3];

b) float num[1][2] = {1,1};

c) int arr1[2][2] = {{2,2}, {3,3}};

2.

a) A string constant is a group of characters or array of characters.

b) An array is a group of related data items which share a common name .

c) A particular element is indicated by writing a number in bracket after array name This number is called as the index number .

d) Matrix is defined by rows & columns which can be easily represented by 2-dimentional array.

e) String concatenation means appending one string at the end of the other.

3 a)

```
main()
{
        int arr1[5][4], i, j;
        int cs[4] = {0, 0, 0, 0};
        printf("\nEnter elements of array :\n");
        for(i=0;i<5;i++)
        {
                for (j = 0; j < 4; j++)
                        scanf("%d", &arr1[i][i]);
        }
        for(i = 0; i < 4 ; i ++)
        {
                for (j = 0; j < 5; j++)
                {
                cs[i] = cs[i] + arr1[j][i];
                }
        }
        for(i = 0; i < 5; i++)
        {
                for(j = 0; j < 4; j++)
                        printf("%d\t",arr1[i][j]);
                printf("\n");
        }
        for(i = 0; i < 4; i++)
                printf("%d\t", cs[i]);
}
```

b)
```
#define ROWS 5
#define COLS 5
main()
{
        int arr1[ROWS][COLS], arr2[ROWS][COLS], arr3[ROWS][COLS];
        int i, j, rows, cols;

        printf("Enter number of rows :");
        scanf("%d", &rows);
        printf("Enter number of columns :");
        scanf("%d", &cols);

        printf("Enter elements of array 1\n");
        for(i = 0; i< rows; i++)
        {
                for(j = 0; j < cols; j++)
                        scanf("%d", &arr1[i][j]);
        }
        printf("Enter elements of array 2 :\n");
        for(i = 0; i < rows; i++)
        {
                for(j = 0; j < cols; j++)
                        scanf("%d", &arr2[i][j]);
        }
        for(i = 0; i < rows; i++)
        {
                for(j = 0; j < cols; j++)
                        arr3[i][i] = arr1[i][j] + arr2[i][j];
        }
        printf("Array1\t\tArray2\t\tArray3\n");
        for(i = 0; i < rows; i++)
        {
                for(j = 0;j < cols; j++)
                        printf("%d ", arr1 [i][j]);
        printf("\t");
        for(j = 0;j < cols; j++)
                printf("%d ", arr2[i][j]); printf("\t");
        for(j = 0;j < cols; j++)
                        printf("%d ", arr3[i][j]);
        printf("\n");
        }
}
```

6.5 & 6.5.1

1 a) characters

b) '\0' character

c) whitespace

d) gets

2. a) 'a'+ 100

b) 7 - 'p'

c) char str1 [10] = “Good”;

d) char str[10] = “One”

e) str1[5] = {‘a’, ‘s’, d, f, g, ‘\0’);

3. a)

```
main()
{
        char str1[25], str2[10];
        int count;
        puts(“Enter string :”);
        gets(str1);
        for(count = 0; count <10; count++)
        str2[count] = str1 [count];
        str2[6ount] = ‘\0’;
        printf(“First string is :\n”);
        puts(str1);
        printf(“Copied string is :\n”);
        puts(str2);
}
```

b)

```
main()
{
         int i;
        char ch, ch1, ch2,ch3;
         i = ‘a’ + 10;
        ch = ‘z’-’c’;
        ch1 = 100 -’r’;
        ch2 = 100-’m’;
        ch3 = ch1 - ch2;
        printf(“i = %d\t”, i); printf(“ch = %c\n”, ch);
        printf(“ch1 = %c\tch2 = %c\tch3 = %c\n”, ch1, ch2,ch3);
}
```

6.5.2

1. a) strlen() : This function is used to determine the length of a string. It returns the length of the string as an integer value.

b) strcat(): Appends one string at the end of the other. It takes two parameters,both of which are strings and appends one string at the end of another.

c) strset() : This function is used to set all the characters of a string to a given character. Both the string and the character to which the string characters are to be set are the parameters provided to this function.

d) strstr() : This function finds the first occurence of a given string in another string. Thus it is useful to determine whether a string is a part of another string or not.

2. a)

```
main()
{
        char str1[25], str2[25];
        int i, j, count;
```

```
        printf("Enter a string :\n");
        gets(str1);
        count = 0;
        i = 0;
        while(str1 [i] != '\0')
        {
        count = count + 1;
        i = i + 1;
        }
        j = 0;
        for(i = count-1; i >= 0; i—)
        {
        str2[j] = str1[i];
        j++;
        }
        str2[j] = '\0';
        printf("String is :\n");
        puts(str1);
        printf("The reverse string is :\n");
        puts(str2);
    }
```

b)

```
    main()
    {
        char str1[25];
        int count, i;
        printf("Enter a string :\n");
        gets(str1);
        count = 0;
        i = 0;
        while(str1[i]!='\0')
        {
        count = count + 1;
        i =i + 1;
        }
        for(i = 0; i < count; i++)
        str1[i] = 'a';
        printf("The replaced string is :\n");
        puts(str1);
    }
```

3. a)

```
    main()
    {
        char str1[25];
        printf("Enter a string to convert to uppercase:\n");
        gets(str1);
        strupr(str1);
        printf("The converted string :\n");
        puts(str1);
    }
```

b)
```
main()
{
char str1[25];
printf("Enter string to reverse :\n");
gets(str1);
strrev(str1);
printf("The reversed string is :\n");
puts(str1);
}
```

c)
```
main()
{
char str1[25];
printf("Enter a string :\n");
gets(str1);
strset(str1 ,'Z');
printf("The set string is :\n");
puts(str1);
}
```

6.5.3

1. a) `char list1[5][3] = {"abc", "pqr", "Imn", "rst", "uvw"};`

b) `char colours[5][10] = {"Blue", "Green", "Red", "Yellow", "Purple"};`

2.
```
main()
{
char str1[5][10], str2[5][10]; int i;
printf("Enter list of alphabets :\n"); for(i = 0; i< 5; i++)
    scanf("%s", str1[i]);
printf("Enter list of colours :\n");
for(i= 0; i < 5; i++)
    scanf("%s", str2[i]);
printf("The list of alphabets :\n");
for(i = 0; i < 5; i++)
    printf("%s\n", str1 [i]);
printf("The list of colours :\n");
for(i = 0; i < 5; i++)
    printf("%s\n", str2[i]);
}
```

6.8 QUESTIONS FOR SELF- STUDY

1. Describe with examples how you will declare the followir

a) a one dimensional array of int

b) a 2-dimensional array of float

c) a 2-dimensional array of characters

2. Describe with example how a 2-dimensional array of int is solved in memory.

3. Write short notes on :

a) Any two standard string manipulation functions.

b) Arrays

c) Representation of a one dimensional array in memory.

d) Various methods of intializing strings.

4 Answer the following in 3-4 Sentences

a. Describe with example the various ways to intilise 2 dimentional array.

b. Describe how 2 -dimensonal arrays are stored in memory.

6.9 SUGGESTED READINGS

Exploring C : Yashwant Kanitkar

C for Beginners : Madhusudan Mothe

CHAPTER 7

DATA TYPES

7.0 OBJECTIVES

Friends, after studying this chapter you will be able to

- state the primary data types along with their variations
- discuss variations of primary data types
- explain typecasting of data
- discuss what is meant by storage classes

7.1 INTRODUCTION

In our previous discussion of primary data types in Chapter 2, we have seen the **int, float** and **char** data types. We also know that these primary data types themselves could be of several types. We shall study the variations of these primary data types here.

Sometimes we are forced to direct the compiler to explicitly convert the value of an expression to a particular data type. In such situations we are required to perform what is known as typecasting. We shall attempt to study this feature. In our earlier chapter, which introduced C tokens, we saw what are bitwise operators. Let us study operations on data using bitwise operators in this chapter.

7.2 INTEGER DATA TYPES

We know how to declare and use the integer data type. We have already used it in many of the programs written so far. There are further variations of the integer data type. Let us now study these variations.

Integer data types are further classified as : **Signed and Unsigned.**

In both the **signed** and **unsigned** we have the **short int** and **long int.**

int (signed integer)	**unsigned int (unsigned integer)**
short int (Plain)	**unsigned short int (Plain)**
long int (Plain)	**unsigned long int (Plain)**

Thus, C has three classes of integer storage : **short int, int** and **long int** in both

the **signed** and **unsigned** forms. We know that an **int** variable occupies 2 bytes i.e. one word in memory. A signed integer uses 1 bit for assigning the sign and 15 bits for the magnitude of the number. The highest order bit of the **int** is used to store the sign of the number. If it is 1, then the number is negative and if it is 0 then the number is positive. By default, the declaration of an integer assumes it to be a signed number. Hence, the use of the qualifier **signed** when declaring an **int** is optional. The range of integer values then lies between -32768 to + 32767 for the **int** data type.

On the other hand, if the integer is declared as **unsigned,** then it uses all the bits for the magnitude of the number and no bit is reserved for the sign. Hence an unsigned integer is always positive. The range of values then becomes double that of the signed. Thus on a 16-bit machine the range of integer values for unsigned integers is 0 to 65,535.

In C, bigger ranges for integer variables are available if you need them. This is offered by a variation of the **int** type and is called the **long int.** The **long int** occupies two words i.e. 4 bytes of memory as compared to the one word of the normal **int** i.e they occupy double the space in memory than an ordinary **int.**

The long variables used to declare **long** integers are declared as follows :

```
long int a;
long int val1;
```

The range of values of **long** integers varies from -2147483648 to + 2147483647.

As in the case of **int** you can also have **unsigned long int . unsigned long int** occupies four bytes in memory, has a range of values between 0 to 4294967295 and is always positive. Like the **long int** variation of **int,** we also have a **short int** variation. Note that the interpretation of a qualified integer data type varies from one compiler to another. Thus we may have a compiler where **short int** may require less storage as compared to the ordinary **int,** or may require the same storage as the ordinary **int,** but as a rule, its storage will never exceed the storage requirement of the ordinary **int.** Similarly in case of **long int,** the **long int** may require the same amount of memory or more memory as compared to the ordinary **int** but it will never require less memory than the ordinary **int.**

We can declare **short int** as follows :

```
short int a;
short int val1;
```

As is the case with **int** and **long int, short int** can also be **signed** or **unsigned.** By default, an **int, long,** or **short** is signed.

The following declarations are also valid in C :

short int a;	*can also be wriiten as*	short a;
long int p;	*can also be written as*	long b;
unsigned int i;	*can also be written as*	unsigned i;
unsigned long int z;	*can also be written as*	unsigned long z;
unsigned short int q;	*can also be written as*	unsigned short q;

C also allows the addition of a suffix ‘L’ or ‘l’ at the end of a number if we wish to give it more storage than the normal **int.** This situation may occur where the constant is small enough to be an **int,** but to give it additional storage of two bytes we use the suffix ‘L’. eg. and int whose value is 50 will occupy two bytes in memory whereas 50L will occupy four bytes in memory.

Example :

Let us write a small program to demonstrate the use of these **int** data types :

```
main()
{
    unsigned int i;
    short  s;
    long l;
    unsigned long j;
    printf("\nEnter unsigned int:");
    scanf("%u", &i);
    printf("\nEnter short int:");
    scanf("%d", &s);
    printf("\nEnter signed long int:");
    scanf("%ld", &l);
    printf("\nEnter unsigned long int:");
    scanf("%lu", &j);
    printf("\nUnsigned int: %u", i);
    printf("\nshort int: %d", s);
    printf("\nsigned long int: %ld",l);
    printf("\nUnsigned long int: %lu", j);
}
```

Study the output of the above program for various integer values for the different integer types and compare the results.

7.2 Check Your Progress.

1. Answer in 1- 2 lines

a) What is meant by long int?

.....

.....

b) How will you describe signed and unsigned int?

.....

.....

2. How will you declare the following ?

(i) a and b as short signed :

.....

(ii) p as long unsigned

.....

(iii) z as unsigned int

.....

3. Write a program to enter a long unsigned int and a long signed int from the keyboard and print the numbers in the following formats :

(i) Padded with leading zeroes

(ii) Left justified

(Using the formatting features studied in input/output)

7.3 FLOAT DATA TYPES

In our previous discussions, we learnt that **float** data type occupies four bytes in memory and can have a range of values between -3.4e38 to +3.4e38. We also know that the **float** data type has a precision of 6.

C offers a variation to **float** data type, which is **double. double** occupies 8 bytes in memory, and has a range of values between -1.7e308 to +1.7e308. A **double** data type uses 64 bits with a precision of 14 bits. Such numbers are known as double precision numbers.

A **double** type variable can be declared as :

double a;

double var1;

For even large real numbers C offers the **long double** data type. The **long double** occupies ten bytes in memory and falls in the range between -1.7e4932 to +1.7e4932. Declaring a **long double** data type yields still higher precision than the **double** data type.

7.4 CHAR DATA TYPES

We already know that **char** data type occupies one byte in memory. By default char is signed. A **signed char** has a range of values between -128 to + 127, **unsigned char** on the other hand has a range from 0 to 255.

Let us see the results of unsigned char and signed char with the following :

Example:

```
main()
{
        unsigned char ch1; char ch2;
        ch1 = 224;
        ch2 = 125;
        printf("Character ch1 = %c\tAscii value = %d",ch1,ch1);
        printf("\nCharacter ch2 = %c\tAscii value = %d",ch2,ch2);
}
```

The output of the program :

```
Character ch1 = α Ascii value = 224
Character ch2 = }          Ascii value = 125
```

In this example, we have declared **char** ch1 as unsigned hence it can take a range of values between 0-255. **char** ch2 is signed and has range of values between -128 to +127. Note that we use the format specification %c to print the character and %d to print its Ascii value.

Example : Write a program to print the Ascii values and the corresponding characters for signed **char** data type.

```
main()
{
        unsigned char i;
        for(i = 0; i< 255; i++)
        {
                printf("Character = %c\tAscii value = %d\n",i,i);
        }
}
```

While writing the program be careful to specify the correct range of values for the

for loop. We have declared **char** i to be unsigned to print the characters and their corresponding ASCII values.

The following table gives the format specifiers for the input and output of all the various data types as well as their range and the number of bytes they occupy :

Data Type	Range	Bytes	Format Specification
Integer			
short signed	-128 to +127	1	%d or %i
short unsigned	0 - 255	1	%u
int	-32,768 to + 32767	2	%d
unsigned int	0 to 65535	2	%u
long signed	-2,147,483,648 to +2,147,483,647	4	%ld
long unsigned	0 to 4,294,967,295	4	%lu
Real			
float	3.4e-38 to3.4e+38	4	%f
double	1.7e-308 to1.7e+308	8	%lf
long double	-3.4e-4932 to 1.1e+4932	10	%Lf
character			
signed	-128 to+127	1	%c
unsigned	0 to 255	1	%c

Making use of the above format specifications, we can input and output these various data types. We have illustrated the use of the **int** and **char** specifications in the previous examples. The following example illustrates the use of **float.**

Example :

```
main()
{
        float f;
        double d;
        long double ld;
        printf("\nEnter float f:");
        scanf("%f", &f);
        printf("Enter double d :");
        scanf("%lf", &d);
        printf("\nEnter long double ld :");
        scanf("%Lf", &ld);
        printf("\nFloat f = %f", f);
        printf("\nDouble d = %lf," d)
        printf("\nLong Double ld = %Lf',ld);
}
```

Compile the program and test it with various sample data of type **float, double** and **long double.**

7.3 & 7.4 Check Your Progress.

1. Answer in 1-2 lines.

a) What is the difference between float and double data type?

.....

.....

b) How will you describe signed and unsigned char data type?

.....

.....

2. Write the format specifications for the following :

a) unsigned char

.....

b) long double

.....

c) float

.....

7.5 TYPE CASTING

7.5.1 Type Conversions in Expressions :

We already know that C allows mixing of variables of different types and constants in expressions. While evaluating these expressions, certain rules are followed. When performing an operation, the compiler considers two operands and the operator associated with them. If the operands differ in type, then first the lower type is automatically converted to the higher type and then the operation is performed. The result is of the higher type. The final result of the expression is converted to the type of the variable on the left side of the assignment operator, before the value is assigned to it. However, you should take a note of the following :

- if the variable on the left of the assignment operator is an **int** and the result you have obtained is **float,** then the fractional part is truncated.
- if double has been converted to **float,** the digits are rounded
- in case of conversion of **long int** to **int,** while assigning to the variable to the left of the assignment operator, the excess higher order bits are dropped.

7.5.2 Type Casting :

In some situations, it becomes necessary to force a type conversion which is different from the automatic conversion done by C. We are required to explicitly convert a value of a particular expression to a specific data type. This is where, local conversion of a data type is done which is known as type casting. The general form of casting a value is:

(type-name) expression

type-name is one of the standard data types of C. The expression can be any valid C expression viz., a constant, variable or expression. The parenthesis around the type-name are essential. The value of the expression, undergoes the type conversion due to type casting. Some examples of type casting :

a = (int) 9.2

With type casting the fractional part will be truncated and the result will be 9.

```
int i = 9, j = 2;
float p;
{
        p = (float) i/j;
}
```

Here since i and j are both **int,** the integer division would yield a result 4 since the fractional part will be truncated. However, when you cast the expression i/j to type float, it will cause the variable i to be first converted to type **float,** before performing the division.

Type casting is useful to round off values. It is always a good programming practice to explicitly force type conversions. It is safer. Rules of automatic conversion should never be assumed, when you are combining variables of different types.

Example : To use type casting

```
main()
{
        int a;
        float b;
        b = 6.54;
        a = (int) (b + 0.9);
        printf("Value of a = %d", a);
}
```

The output of the program :

Value of a = 7

In this example, the value of the expression undergoes type conversion before being assigned to a. Note that the expression being cast itself does not change.

7.5 Check Your Progress.

1. What will the following type casting result in ?

a) a = (int) 18.3

.....

b) z = (int) 4.7 / (int) 2.3

.....

c) z = (int) 12.8 + 13.7

.....

2. Explain what action will take place in the following type casting operations.

a) x = (int) 4.2 + (int)3.7

.....

b) (float) (p/q)

.....

c) 1/float(a);

.....

7.6 SUMMARY

C language provides a variety of data types.

- There are basically four types of data in C. They are char , int, float and double. The basic types can be further extended by applying different qualifiers.
- Integer data type is further classified as short integer and long integer.
- Float is used for floating point number.
- Character data type can be signed characters or unsigned characters.
- Type casting does not permanently change the type of a variable. It only temporarily presents the variable in the required data types.

7.7 CHECK YOUR PROGRESS - ANSWERS

7.2

1.a) Long int is a variation of the int type. The long int occupies two words i.e. 4 bytes of memory as compared to the one word of the normal int i.e they occupy double the space in memory than ordinary int. The range of values of long integers varies from -2147483648 to + 2147483647. Long ints are of two types : unsigned long int and signed long int. By default a long int is signed, unsigned long int has a range of values between 0 to 4294967295 and is always positive.

b) A signed integer uses 1 bit for assigning the sign and 15 bits for the magnitude of the number. The highest order bit of the int is used to store the sign of the number. If it is 1, then the number is negative and if it is 0 then the number is positive. By default, the declaration of an integer assumes it to be a signed number. The range of integer values then lies between -32768 to + 32767 for the int data type. On the other hand, if the integer is declared as unsigned, then it uses all the bits for the magnitude of the number and no bit is reserved for the sign. Hence an unsigned integer is always positive. The range of values then becomes double that of the signed. Thus on a 16-bit machine the range of integer values for unsigned integers is 0 to 65,535.

2. (i) short a, b;

(ii) unsigned p;

(iii) unsigned z;

3.
```
main()
{
    long unsigned u;
    long l;
    printf("Enter value of unsigned long int :");
    scanf("%lu", &u);
    printf("\nEnter signed long int :");
    scanf("%ld", &l);
    printf("Unsigned long padded with leading zeroes and left justified :\n");
    printf("%014lu\n%-14lu", u, u);
    printf("\nSigned long padded with leading zeroes and left justified : \n");
    printf("%014ld\n%-14ld", l, l);
}
```

7.3 & 7.4

1. a) For representation of real numbers we make use of the float and double data types. float data type occupies four bytes in memory and can have a range of values between -3.4e38 to +3.4e38. The float data type has a precision of 6. double is a variation to float data type, which occupies 8 bytes in memory, and has a range of values between -1.7e308 to +1.7e308. A double data type uses 64 bits with a precision of 14 bits. Such numbers are known as double precision numbers.

b) The char data type is used to represent a single character. This character can be an alphabet, a digit or a special symbol. char data type occupies one byte in memory. By default char is signed. A signed char has a range of values between -128 to + 127, unsigned char on the other hand has a range from 0 to 255. We use the format specification %c to print the character both in the signed and the unsigned form

2. a) %c

b) %Lf

c) %f

7.5

1. a) a = 18

b) z = 2

c) z = 25.7

2.

a) 4.2 will be converted to int and 3.7 will be converted to int and the expression evaluated as 4/2.

b) The result of p/q will be converted to float.

c) a will be cast to float and the reciprocal will be determined.

7.8 QUESTIONS FOR SELF-STUDY

1. Answer in two-three sentence :

a) What is the classification of integer storage in C?

b) What is meant by double precision numbers?

c) What is type casting?

d) Which are the locations where variable values are generally stored?

e) What are local variables and global variables?

2. Write short notes on :

1) Type casting

2) Integer Data Types

3) float data types

4) Type Conversion in expressions.

7.9 SUGGESTED READINGS

Let us C : Yashwant kanitkar

Spirit of C : Mullish cooper

CHAPTER 8

FUNCTIONS

8.0 OBJECTIVES

Friends, the study of this chapter will help you to

- describe what are functions in C.
- start writing user defined functions.
- explain the form of a function and catagories of functions.
- state advanced features of function and function nesting.
- write much more sophisticated programs using functions.

8.1 INTRODUCTION

In the introduction of C language, we have said that a C program is nothing but a collection of functions. A function is defined as a self contained block of statements that perform a specific task of some kind. In all the programs we have written so far, we have made use of functions. Some of the functions we have used so far are **main, printf, scanf** etc. Functions are of two types : **library functions** and **user defined functions.** Library functions are those functions which are not required to be written by us. We can directly use them in our programs. User defined functions are functions developed by the user while writing programs. Of course, a user defined function can become a part of a library function later. We know that **main()** is a special type of function in C. Every program must have a **main().** This is the point where the execution of a program starts. We can divide our program into smaller functional sub programs where each subprogram may be independently coded, debugged and tested. These subprograms can then be combined together into a single program. These subprograms are nothing but functions.

Advantages of functions :

- Functions facilitate top-down modular programming. The overall problem is solved first and the details of the various lower levels are solved later.
- Functions reduce the length of the source program.
- Functions can be independently tested and debugged
- Functions can be used by many other programs. This means that it is not necessary to write functions which have already been written. They can be directly used in the programs.

Let us now begin our study of functions, their catagories and their features.

8.2 WRITING USER DEFINED FUNCTIONS

Let us now begin writing user defined functions. We shall write a C program which makes use of a user defined function. With the help of this program we shall study the various concepts related to functions.

Example :

```
/*Program to demonstrate the use of a function */
void drline ();
main()
{
        drline();
        printf("The function prints stars on the terminal\n");
        drline();
}
void drline()
{
        int i;
        for (i=1;i<=50,i++)
                printf("*");
        printf("\n");
}
```

The output of this program is :

```
**************************************************
The function prints stars on the terminal
**************************************************
```

There are a number of things to be understood about functions. Let us study them with the help of the above program. The program contains two functions :

main()

drline()

The execution of the program begins with **main().** When executing the program, the statement **drline();** is encountered. **drline()** is nothing but a function which we have written. At this point in the program, the control is transferred to the function **drline().** The **drline()** function in itself is a complete block of statements. This function prints * on the screen. When the function is executed, program control is again transferred to **main(). main()** restarts program execution from the statement immediately following the function call. After executing **printf,** control is again transferred to function **drline()** since we have again called **drline(). drline()** draws one more line. The program control again passes back to **main().** Both the **main()** and **drline()** functions make use of library function **printf.**

The **main()** function calls the drline() function. Here **main()** is the calling function and **drline()** is the called function. Any function can call any function. Also, any function can be called any number of times. A called function can also call functions. There is no precedence, no rules of hierarchies for functions.

Functions can be placed in any order. A called function can be placed before or after a calling function. (However, in usual practice all the called functions are put at the end.) It is however important to note, that a function cannot be defined in another function.

Any C program will contain at least one function. If it contains only one function then it must be **main().**

Let us further understand these points with the following examples :

Example: The example illustrates how two functions have been written and called in main()

```
void drline ();
void drstar();
main()
{
        drline();
        drstar();
        drline();
}
void drline()
{
        int i;
        for(i=1; i<=50; i++)
        printf("-");
        printf("\n");
}
void drstar()
{
        int k;
        for(k=1 ;k<=50;k++)
        printf("*");
        printf("\n");
}
```

Here **drline()** is called twice in the program. The functions **drline()** and **drstar()** are written after the function **main().** Their order of precedence is not important. We could have first written **drstar()** and then **drline()** also.

Let us see how a called function itself can call other functions with the help of the following example :

Example :

```
void fn1();
void fn2();
main()
{
        printf("\nCalling Functions");
        fn1();
        fn2();
}
void fn1()
{
        printf("\nLeaming C");
}
void fn2()
{
        printf("\nThis function calls function 1");
```

```
    fn1();
}
```

And the output of the program is :

```
Calling Functions
Learning C
This function calls function 1
Learning C
```

Note carefully the sequence of execution of this program. **main()** first calls fn1(). fn1() gets executed. Then fn2() is called through **main().** fn2() itself calls fn1(). Here it is not important whether fn 1 () is written first or fn2() is written first.

8.2.1 The general form of a C function ;

```
function_name(list of arguments) argument declaration;
{
    local variable declaration;
    statement_1;
    statement_2;
    :
    :
    return(expression);
}
```

In the general form of a C function there are many parts which are not a must while writing a function. For example, the list of arguments and argument declaration is optional. As we have already seen **main()** itself is a function for which there have been no arguments or argument declaration so far. The function drline() which we have written above also had no arguments. The method shown above is one method of declaring arguments. Another method of declaring arguments is the one where the type of the arguments are declared in the function header itself eg.

```
sum(a,b)
int a, b ;
```

is the first method.

In the second method the arguments are declared as follows :

```
sum(int a, int b);
```

The first method is known as the Kernighan & Ritchie (K&R) method. The second method is more commonly used nowadays.

If a function is using variables, then they are declared. **Local variables** are those variables which are used only by that particular function in which they are declared. In our drline() function we have used the local variable i.

The executable statements of the function come next. A function can have any number of executable statements. On the other hand, a function may not include any executable statement either, i.e. we may have a function which does nothing and which does not have any executable statements.

The **return** statement is optional. If any value is to be returned to the calling function, it is returned with the help of the **return** statement. Thus a **return** statement is used if any value from the called function is to be returned to the calling function. We shall study the **return** statement in detail subsequently.

A few points to remember:

- The function name has to follow the same rules as those of the variable names in C. Also care has to be taken to avoid assigning the library routine names or operating system commands as function names.

- The argument list is the list of valid variable names separated by commas. The argument list should be enclosed in a pair of parenthesis and should be placed

immediately after the function name.

The use of the argument list : The argument list is used to pass values to the called function from the calling function. Thus it provides a means of data communication between the two functions, eg.

```
sq(a);
prod(a,b);
power(x.n);
```

All the variables in the argument list must be declared with their types immediately after the **function header** (function header is the definition of the function name followed by the declaration of the arguments) and before the opening braces of the function body, eg.

```
sq(a);
float a;
{
        -------
        -------
        -------
}
```

Here the variable a is the argument of the function sq and is declared for its type as **float** immediately after the function header.

```
power(x,n);
float x;
int n;
{
        -------
        -------
}
```

The function power has two arguments x and n where x is declared of type **float** and n is declared to be of type **int.** Both the arguments are declared for their type immediately after the function header.

8.1 & 8.2 Check Your Progress.

1. Answer the following :

a) What are the advantages of functions?

.....

.....

b) Describe the general form of a C function.

.....

.....

2. Fill in the blanks :

a) .....variables are those variables which are used only by that particular function in which they are declared.

b) If a C program contains only one function then it must be.....

c) A function..... is the definition of the function name followed by the declaration of the arguments.

d) The.....statement is used if any value from the called function is to be returned to the calling function.

8.3 FUNCTION CATAGORIES

We learnt the general form of a function, the meaning of argument list and **return statement.** On the basis of these argument lists and **return statement** the functions can be catagorised as:

- Functions with no arguments and no return values
- Functions with arguments but no return values
- Functions with arguments and return values.

Let us study the various catagories of functions.

8.3.1 Functions with no arguments and no return values :

As we have seen earlier, when a function has no arguments, no data is sent to the function from the calling function. Also when the function is not returning any value, the called function does not receive any data from the calling function, i.e. there is only a transfer of control and not data. The general form of such a function which receives no arguments from the calling function and returns no values :

```
fn1()
{
        -------
        -------;
        fn2();
        -------;
}
fn2()
{
        -------
        -------
}
```

Here fn2() is called by fn1 (). However, no arguments are passed by fn1 () to fn2(). Similarly fn2() also does not return any value to fn1 ().

Example : The following example will illustrate a function with no arguments and no return values.

```
void add();
main()
{
printf("\nFunction to add numbers");
add();
}
void add()
{
        int a, b, sum;
        printf("\nEnter value for a :");
        scanf("%d",&a);
        printf("\nEnter value for b :");
        scanf("%d",&b);
        sum = a + b;
        printf("The sum is : %d", sum);
}
```

A sample output:

```
Function to add numbers
Enter value for a : 100
Enter value for b : 200
The sum is :300
```

main() calls the function add() which takes values for a and b. It sums a and b and prints the sum. There is no **return** statement in the function. The **return** statement is optional if the function does not return anything.

Example : The following function is called in main() to read a string and print it.

```
void str_fn();
main()
{
        char ch1='Y';
        printf("\nEnter a string :");
        while(ch1 = getch() != 'N')
        {
                str_fn();
                printf("Do you wish to enter another string (Y/N) ? :");
        }
        }
        void str_fn()
{
        char str1[40];
        printf("\nEnter a string :");
        scanf("%s", str1);
        printf("%s", str1);
}
```

A sample output:

```
Enter a string :CFunctions
CFunctions
Do you wish to enter another string (Y/N) ? :Y
Enter a string :Noarguments
Noarguments
Do you wish to enter another string (Y/N) ? :N
```

In the program **main()** calls str_fn(). This function reads a string and prints it. The control again goes back to **main().** Here the user types a 'Y' if he wants to enter another string and a 'N' to stop. Thus the program will execute until the user enters a 'N' and the function str_fn() will be called repeatedly by **main().** This demonstates how a function can be called by a calling function a number of times. However, no arguments are passed to str_fn() and no return values to main() by str_fn().

8.3.2 Functions with arguments and no return values :

In calls to functions with arguments we can send the list of arguments from the calling function to the called function.

The general form of a function with arguments but no return values :

```
fn1()
{
        - - - - - -
        - - - - - -
```

```
        fn2(actual argument list);
        - - - - -
        - - - - -
}
fn2(formal argument list)
{
        - - - - -
        - - - - -
}
```

Example : Let us modify our above example of adding two numbers and send the values of two numbers to the function to add(). This function will accept the arguments but will not return any value.

```
void add(int, int);
main()
{
        int i, j;
        printf("\nFunction to add numbers");
        printf("\nEnter value for i :");
        scanf("%d",&i);
        printf("\nEnter value for j :");
        scanf("%d",&j);
        add(i, j);
}
void add(int a, int b)
{
        int sum;
        sum = a + b;
        printf("The sum is : %d", sum);
}
```

A sample output:

```
Function to add numbers
Enter value for a : 100
Enter value for b : 200
The sum is :300
```

Actual arguments and formal arguments :

In this program **main()** passes the values of i and j to the function add(). Note that when the function add() is called from **main(),** i and j are mentioned in the parenthesis (add(i,j)). These values are collected in the variable a and b in the function add(). The variables i and j are called the **actual arguments** and a and b are called **formal arguments.** Note that the type, order and number of actual arguments and formal arguments in the calling and called functions must match. They are matched one by one starting with the first argument. In such functions there is a one way communication where the calling function sends arguments to the called function. The called function however returns no data to the calling function. The use of the **return** statement is therefore optional since the function does not return anything.

If actual arguments are more than the formal arguments they are ignored. On the other hand if the actual arguments are less than the formal arguments then the unmatched formal arguments may get initialised to garbage values. Remember that the compiler will not generate any error message. It is our responsibility to ensure that the list of actual arguments and formal arguments match in type, number and order.

One more important point to remember is that when a function call is made only a copy of the values of the actual arguments is passed into the called functions and not the actual arguments. Whatever happens inside the function has no effect on the values of the actual arguments. This means that if the values of the formal arguments are changed in the called function, the corresponding change does not take place in the actual arguments of the calling function.

Example : To illustrate that change in values of formal arguments has no effect on the values of the actual arguments.

```
/* Program to illustrate actual and formal arguments */
void fn1 (int);
main()
{
        int a;
        a = 20;
        fn1(a);
        printf("\nThe value of a is : %d", a);
}
void fn1 (int i)
{
        i = 100;
        printf("\nThe value of i is :%d", i);
}
```

The output of the program will be

The value of i is 100

The value of a is 20

From the above example you can see that the value of the formal argument **i** is changed in the function fn1(). The value of the actual argument **a** remains unchanged.

8.3.3 Functions with arguments and return values :

In the above examples, we saw that we could pass arguments to the function but the called function itself did not return any values. But in more practical situations, we may need the return values from the called functions for further processing. Since a functions may be called by a number of other functions each function may require the return values to be output in different forms also. Thus a self contained function should be one which receives a predefined form of input and outputs a desired value. This will effect a two way communication by way of passing values between functions. The general form functions with arguments and return values could be depicted as:

```
fn1()
{
        - - - - -
        - - - - -
        fn2(actual arguments list)
        - - - - -
}
fn2(formal argument list)
{
        - - - - - -
        - - - - -
        return(result);
```

Example : Let us modify the above example to send values of a and b to the function sum() from **main().** The function sum() shall return the value of the addition to main().

```
int add(int, int);
main()
{
        int i,j, sum;
        printf("\nEnter first number:");
        scanf("%d",&i);
        printf("\nEnter second number:");
        scanf("%d",&j);
        sum = add(i,j);
        printf("\nThe sum is : %d", sum);
}
int add(int a, int b)
{
        int c;
        c = a + b;
        retum(c);
}
```

A sample run of the program :

```
Enter first number :10
Enter second number :40
The sum is : 50
```

The **return** statement is used to return a value to the calling function. The **return** statement returns the value of the addition of the two numbers to **main().** Note here that the sum of the numbers is output from **main()** and not from the function add().

8.3.1, 8.3.2 & 8.3.3 Check Your Progress.

1. Write true or false :

a) Every function has a return statement.

b) A function can be called any number of times by another function.

c) You can write a function which has arguments but no return values.

d) Both the values of actual arguments and formal arguments change when the function is executed.

e) In calls to functions with arguments we can send the list of arguments from the calling function to the called function.

2. Write C programs for the following functions :

a) Enter a 5 digit number and find the sum of digits in a function.

b) Write a function to calculate the sum of n even numbers starting from the number a (where n and a are input by the user).

c) Write a function to generate and print the first n elements of a Fibonacci series and find their sum. In a Fibonacci series every number is the sum of the preceding two numbers.

eg. 1 1 2 3 5 8 11 You may start with the first number of the series as 1.

8.3.4 Return Values and their types :

Now that we have introduced the **return** statement let us study it in more detail. We have seen that the **return** statement is used to return a value to the calling function.

The **return** statement can take one of the following forms :

return;

or

return(expression);

The first form of **return** does not **return** any value to the calling function. The second form of **return** returns a value to the calling function. In both the cases, the control is transferred back to the calling program when the return is encountered. Thus the **return** statement

- on execution transfers control back to the calling function

- returns the value of the expression in the parenthesis to the calling function.

eg, return(sum);
return (x + y +z);
retum(p);
return (a*b);

(i) A calling function can pass any number of values as arguments to the called function. However, the called function can return only value per call to the calling function.

Thus the statements

return(a, b);

or

return(a,10);

are not valid.

(ii) There can be more than one **return** statement in certain situations eg.

```
if(x <0)
        return(1);
else
        return (0);
if (ch = ‘A)
        return(a + b);
else
{
        if(ch = ‘M’)
                return(a *b);
        else
                return(a - b);
}
```

(iii) All functions be default return data of type **int.** If we want the function to return a particular type of data other than **int** then we are required to specify the data type in the function header:

eg. float prod(a,b);

double sq_rt(p);

We shall study how to make a function return a value other than **int** in the section advanced features of functions.

8.3.5 Scope Rule of Functions :

This section includes a description on the scope of variables of a function. The default scope of a variable is local to the function in which it is defined. This means that the presence of a variable is known only to the function in which it is declared and not to any other function. Thus if a variable a is defined in fn1(), then a is known only to fn1() and not to any other function. Hence the scope of a is said to be local to fn1 ().The following example will illustrate :

Example : To demonstrate scope rule of functions :

```
int sq(int);
main()
{
        int i, z;
        printf("\nEnter number to square :");
        scanf("%d", &i);
        z = sq(i);
        printf("\nThe square is %d", z);
}
int sq(int p);
{
        int q;
        q = p * p;
        return(q);
}
```

A sample run of the program :

Enter a number to square :5

The square is 25

Here the variables **i** and **z** are known only to the function **main().** Similarly variable **q** is local only to the function sq(). It is not known to **main().** Thus the value of i is not known to sq(). We send it as an argument to make it available to sq().

8.3.4 & 8.3.5 Check Your Progress.

1. What will the following functions return :

a)
```
int i = 0, j = 5;
j = fn(i);
fn(a)
{
        return(a);
}
.....
```

b)
```
int p,q;
q=fn(p);
fn(a)
```

```
{
if(a < 0)
        return(a * a);
else
        return (0);
}
```

for the value of q = 0

.....

2. Write true or false :

a) The scope of a variable is local to the function in which it is defined.

b) Every function has to return a value to the calling function.

c) return(a - b + c); is not a valid return statement.

d) Functions by default return a value of type int or float.

e) Functions can return only int values.

3. Write C programs for the following functions to return values as expected :

a) Two numbers are input. The function returns whichever of the two is greater.

b) A 4 digit number is input. The function returns the number with the digits reversed.

c) Write a function to return a 0 if a number is divisible by 3 and 1 otherwise.

8.4 ADVANCED FEATURES OF FUNCTIONS

8.4.1 Function Declaration and Prototypes :

We know by now that all functions be default return data of type **int.** If we want the function to return a particular type of data other than **int** then we specify the data type in the function header:

Example : Let us write a program to illustrate the use of return to return a value other than int.

```
float fn1 (float, float);
main()
{
        float fn1(); .
        int i, j ;
        float div;
        printf("\nEnter value of i:");
        scanf("%d", &i);
        printf("\nEnter value of j :");
        scanf("%d", &j);
        div = fn1(i,j);
        printf("\nThe value of a/b is %f:", div);
}
float fn1 (p, q)
{
float r;
```

```
r = p/q;
return(r);
}
```

Note here that the function fn1() is declared **float** in **main().** The function header is itself written as :

float fn1(),

which means that the function will return a **float** value.

In other situations it may be that we do not want the function to return any value. To make this possible we make use of the keyword **void** in the function header. When a function is declared **void** it means that the function will return nothing. eg.

Example : To illustrate void

```
void fn1();
main()
{
        void fn1();
        fn1();
}
void fn1()
{
        printf("\nFunctions in C");
        printf("\nThe function to illustrate that it is void");
}
```

The output of the program will be :

Functions in C

The function to illustrate that it is void

8.4.2 Call by Value and Call by reference :

We have studied that when we pass actual arguments to functions from calling function, a copy of these arguments is collected in the formal arguments in the called function. Uptil now we have seen several examples where we have passed values to function in this manner.

Arguments are passed to functions in two ways :

- sending the values of the argurments

- sending the addresses of the arguments

In all the functions used so far we have always passed the values of the variables (arguments) to the called functions. Such function calls are called **'calls by value'.** The examples illustrated above are calls by value.

eg. sum(ij);

fn1(a,b);

Similarly, there is another method by which we can pass the address of the variable to a function. (Variable address is the memory location of the variable). When we pass the address of the variable to a function it is referred to as **'call be reference'.** We have to make use of pointers for this purpose. Therefore we shall defer the discussion of **call by reference** till we study pointers.

8.4.3 Recursion :

In C, it is possible for a function to call itself. If a statement in the body of a function calls the function itself then the function is known as **recursive function.** We shall study recursion with the help of the following program which calculates the

factorial of a number.

The factorial of a number is the product of all integers from 1 through to the given number. Thus factorial of 5 = 1x2x3x4x5

Example : First let us write a simple program using a function to calculate the factorial :

```
/*Program to find the factorial */
int fact (int);
main()
{
        int n, factorial;
        printf("\nEnter a number to find its factorial :");
        scanf("%d", &n);
        factorial =fact(n);
        printf("\nThe factorial is : %d", factorial);
}
int fact( int a)
{
        int i, prod;
        prod = 1;
        for(i = a; i >= 1 ; i—)
        {
                prod = prod * i;
        }
        return(prod);
}
```

A sample run of the program :

```
Enter a number to find its factorial : 5
The factorial is : 120
```

Here the **for** loop executes from i = 5 to i = 1 by making use of the decrement counter. The product is multiplied by the value of i everytime. Thus till i becomes 1, we obtain 5 x 4 x 3 x 2 x 1 = 120. The value 120 is stored in prod and returned to **main().** The factorial value is then printed using printf().

Example : The same function can be written in a recursive way as follows :

```
/*Program to find the factorial using recursion */
int fact (int);
main()
{
        int n, factorial;
        printf("\nEnter a number to find its factorial :");
        scanf("%d", &n);
        factorial = fact(n);
        printf("\nThe factorial is : %d", factorial);
}
int fact( int a)
{
         int i;
        if(a==1)
                return(1);
```

```
        else
                i = a*fact(a-1);
        return(i);
}
```

A sample run of the program :

Enter a number to find its factorial: 4

The factorial is :24

Let us understand the working of the program carefully :

main() first reads a number to find its factorial. It is then passed to the function fact(). If the number is 1, the value is returned as 1 else the statement i = a * fact(a-1) again calls fact() by passing the value of a as 3. In the above example i = 4 x fact(3). Since fact() is called again it executes in the same way and i = 4 x 3 x fact(2). The next call is 2 x fact(1). fact(1) returns 1 and thus

i = 4 x fact(3)

= 4 x 3 x fact(2)

= 4 x 3 x 2 x fact(1)

= 4 x 3 x 2 x 1

Thus fact(4) returns 4 x fact(3)

which returns 3 x fact(2)

which returns 2 x fact(1)

which returns 1

While using recursive functions care has to be taken to have an **if** statement to provide a way to make the function **return** without giving a recursive call, otherwise it will fall in an infinite loop. An example of a recursive function falling in an infinite loop is shown as follows :

Example :

```
main()
{
        printf("\nAn example of recursion");
        main();'
}
```

Here **main()** calls **main()** continuously and the **printf** statement will keep on executing. The program will never come out of the loop. The execution will have to be terminated abruptly. Such situations should be avoided while using recursion. Recursive functions should have a **if** statement to force the function out of recursion.

8.4 Check Your Progress.

1. Answer the following :

a) How do you make a function return a value other than int?

.....

.....

b) What is a void function?

.....

.....

c) What is meant by call by value and call by reference?

.....

.....

d) Explain recursion.

.....

.....

2. Write C programs for the following :

a) Use recursion to find the sum of the first n numbers.

b) Write a function to find the product of two float numbers and another function tofind their quotient. Call both these functions from main and make them return valuesof type float.

8.5 NESTING OF FUNCTIONS

C allows nesting of functions. This means that a function which is being called can itself call another function, which itself may call another function and so on.

We have already made use of a number of standard library functions in our functions. This means that the function which we are calling from another function is itself calling other functions. Let us write a program to illustrate how a called function itself calls other functions.

The program converts a distance entered in km to cm. The function tom() converts km to m, but tom() itself calls tocm() which converts the distance in metres to cm.

Example : To convert distance entered in km to cm using nesting functions.

```
float tom(float);
float tocm(float);
main()
{
        float km, cm ;
        printf("\nEnter distance :");
        scanf("%f", &km);
        cm = tom(km);
        printf("\nThe distance in cm : %6.2f", cm);
}
float tom(i)
{
        float j, k;
        j = i * 1000;
        k = tocm(j);
return(k);
}
float tocm(a)
{
        float b;
        b = a * 100;
        return(b);
}
```

A sample run :

```
Enter distance :34.585
```

The distance in cm :3458500.00

Note that both the functions tocm() and tom() return non integer values. Therefore the word **float** is to be written before the function name in the function header. Also both the functions have to be explicitly mentioned in the calling function **main()** as functions whose return type is **float.** Follow the program carefully to understand how the function tom() calls tocm() to convert the distance in metres to cm.

C also allows nesting of function calls. Thus add(add(a,b), c) is possible. Let us see how to write a program to demonstrate nesting of function calls:

Example:

```
int add(int, int);
main()
{
        int a, b, c, sum;
        printf("\nEnter value for a :");
        scanf("%d", &a);
        printf("\nEnter value for b :");
        scanf("%d", &b);
        printf("\nEnter value for c :");
        scanf("%d", &c);
        sum = add(a, add(b,c));
        printf("\nThe sum is : %d", sum);
}
int add(int i, int j)
{
        int k;
        k = i+j;
        retum(k);
}
```

A sample output of the program :

```
Enter value for a :20
Enter value for b :40
Enter value for c :80
The sum is : 140
```

Here we have a function add() which adds two numbers. Now if we wish to use the function add() to add three numbers we can do so as shown above by making use of nesting function calls. We call the function add() with the values as a and the secondargument as a function call to add() with the arguments b and c. add(b,c) is first executed. It returns the value of a + b. This then is the second argument to the outer add() which now adds the sum to a by calling add() and thus the three numbers get added.

8.6 PASSING ARRAYS AS ARGUMENTS TO FUNCTIONS

Just as we can pass values of simple variables of type **int, float** etc to functions it is also possible to pass values of an array to a function. The following form is used to pass an array to a called function :

```
avg(num,n)
```

Here num is the array and n is the total number of elements in the array a. Note that when passing an array as an argument, only the array name without any subscripts is listed, followed by the size (number of elements) of the array. The called function should also be appropriately defined.

eg.

avg(num,n) is a function call to the function avg() which calculates the average of the elements of the array num. Let us assume that the array num contains values of type **float.** Then the called function will be defined as :

```
float avg(num, n)
float num[];
int n;
```

Thus avg takes two parameters (arguments) the array name and its size. The declaration

```
float num[];
```

tells the compiler that num is an array of numbers of type **float.** Note that it is not necessary to declare the size of the array here.

Having understood this let us now write an actual program to calculate the average of the elements of the array.

Example:

```
float avg (float [], int);
main()
{
int j;
float average;
float num[5] = {8.2, 7.8, 3.3, 9.2, 8.1};
for(j = 0; j <5; j++)
printf("\nNumber = %6.2f", num[j]);
average = avg(num, 5);
printf("\nAverage is : %6.2f", average);
}
float avg(float num [], int n)
{
int i;
float sum = 0;
for (i = 0; i < n; i++)
sum += num[i];
sum = sum/n;
return(sum);
}
```

The output of the program will be :

```
Number = 8.2
```

Number = 7.8

Number = 3.3

Number = 9.2

Number = 8.1

Average is : 7.32

Follow carefully the working of the program. The array num[] is passed as an argument to function avg(). Note that avg() returns a **float** value, hence its type has to be explicitly declared in both the called and calling functions. The array is passed in the specific format as explained above, and the function average is also appropriately defined. The function first calculates the sum of all the elements of the array and then averages it. The average is returned to the calling function **main()** and printed.

8.5 & 8.6 Check Your Progress.

1. Write C programs for the following :

a) Pass an array of int numbers to a function and find the smallest number of the array and return it to the calling function.

b) Pass an array of characters to a function and print the elements of the array in the reverse order in the function.

c) Write a function mul() to find the product of two numbers. Make use of nesting of function calls to find the product of 4 numbers with the help of mul().

d) Write a function to find the average of n numbers. This function should call another function which will calculate the sum of these numbers. The calling function will then calculate the average with the sum returned and return the value to main.

8.7 SUMMARY

In this chapter we learnt about one of the most important aspects of Programming - Function.

A function is defined as a self contained block of statements that perform a specific task of some kind. Functions are bifurcated in to two types : library functions and user defined functions. Library functions are those functions which are not required to be written by user. We can directly use them in our programs. User defined functions are functions developed by the user while writing programs.

Categories of Functions: On the basis of argument lists and return the functions can be categorized as :

" Functions with no arguments and no return

" Functions with arguments and no return values

" Function with argument and return values

The call to the function can be by value or reference - Passing the value of the variables to the called function implies that function calls are "call by value". When we pass the address of the variable to the function the it is referred to as " call by reference".

Recursion : If a statement in the body of a function calls the function itself then the function is known as recursive function.

8.8 CHECK YOUR PROGRESS - ANSWERS

8.1 & 8.2

1.a) There are a number of advantages of functions :

(i) Functions facilitate top-down modular programming. The overall problem is solved first and the details of the various lower levels are solved later.

(ii) Functions reduce the length of the source program.

(iii) Functions can be independently tested and debugged

(iv) Functions can be used by many other programs. This means that it is not necessary to write functions which have already been written. They can be directly used in the programs.

b) The general form of a C function is :

```
function_name(list of arguments)
argument declaration;
{
        local variable declaration;
        statement_1;
        statement_2;
        :
        return(expression);
}
```

The function name is followed by the list of arguments and argument declaration. Arguments can also be declared in the function header itself. If a function is using any variables, then they are declared in the function. The executable statements of the function come next. A function can have any number of executable statements. On the other hand, a function may not include any executable statement either. The return statement is optional. A return statement is used if any value from the called function is to be returned to the calling function.

2. a) local

b) main

c) header

d) return

8.3.1, 8.3.2 & 8.3.3

1. a) False

b) True

c) True

d) False

e) True

2. a)

```
void fn sum (int);
main()
{
        int num;
        printf("Enter a five digit number:");
        scanf("%d", &num);
        fnsum(num);
}
```

```
void fnsum(j)
{
        int k, add;
        add = 0;
        for(k = 1; k <=5; k++)
        {
                add = add + j%10;
                j=j/10;
        }
        printf("\nThe sum is : %d", add);
}
```

b)
```
void fnsum (int, int);
main()
{
        int n, a;
        printf{"Enter first even number : ");
        scanf("%d", &a);
        printf(" \nEnter total number of numbers :");
        scanf("%d", &n);
        fnsum(a,n);
}
void fnsum(i,j)
{
        int add, k;
        add = 0;
        for(k = i; k < (i + 2*j); k = k+2)
        add = add + k;
        printf("The sum is : %d", add);
}
```

c)
```
void f1bo(int);
main()
{
        int n;
        printf("Enter the value of n :");
        scanf("%d", &n);
        fibo(n);
}
void fibo(num)
{
        int i,n1,n2,no, add;
        n1 = 1;
        n2 = 1;
        add = n1 + n2;
        printf("The Fibonacci Series\n%d\t%d\t", n1, n2);
        for(i= 3;i<=num; i++)
        {
```

```
            no = n1 + n2;
            add = add + no;
            printf("%d\t", no);
            n1=n2;
            n2= no;
        }
        printf("\nThe sum is :%d", add);
    }
```

8.3.4

1. a) 0

b) 0

2. a) True

b) False

c) False

d) False

e) False

3. a)

```
int fnbig(int, int);

main()
{
    int n1, n2, big;
    printf("Enter values for n1 and n2 :");
    scanf("%d%d", &n1, &n2);
    big = fnbig(n1,n2);
    printf("\nThe greater number is :%d", big);
}
int fnbig(i,j)
int i, j;
{
    if(i>j)
        return(i);
else
    return(j);
}
```

b)

```
int fnrev (int);
main()
{
    int num, newnum;
    printf("Enter a 4 digit num :");
    scanf("%d", &num);
    newnum = fnrev(num);
    printf("The number reversed is : %d",newnum);
}
int fnrev(int, n1)
{
```

```
        int i,k, new;
        i = 1000;
        new = 0;
        for(k = 1; k <= 4; k++)
        {
                new = new + n1%10 * i;
                i = i/10;
                n1 =n1/10;
        }
        return(new);
}
```

c)
```
int fncheck (int);
main()
{
int n, result;
printf("Enter a number:");
scanf("%d", &n);
result = fncheck(n);
printf("The value returned is : %d", result);
if(result == 0)
printf("\nThe number is divisible by 3");
else
        printf("\nThe number is not divisible by 3");
}
int fncheck(int, n1);
int n1;
{
        if(n1%3==0)
                return(0);
        else
        return(1);
}
```

8.4

1.a) If we want the function to return a particular type of data other than **int** then we specify the data type in the function header eg. if we want a function to return a data type of float we specify it as :

float fn1(),

which means that the function will return a **float** value.

b) A function when declared **void** means that it will return nothing.To make this possible we make use of the keyword **void** in the function header, eg. void fn1 (); will not return anything to the calling function.

c) When we have pass the values of the variables (arguments) to the called functions then such function calls are called **'calls by value'.**

eg. sum(i,j);

When we pass the address of the variable to a function it is referred to as **'call by reference'.** (Variable address is the memory location of the variable). We have to make use of pointers for this purpose.

d) In C, it is possible for a function to call itself. If a statement in the body of a function calls the function itself then the function is known as recursive function. While using recursive functions care has to be taken to have an if statement to provide a way to make the function return without giving a recursive call, otherwise it will fall in an infinite loop. Recursive functions should have a if statement to force the function out of recursion.

2. a)

```
int fact (int);
main()
{
        int n, sum;
        printf("Enter the value of n :");
        scanf("%d", &n);
        sum = fact(n);
        printf("The sum is : %d", sum);
}
int fact(int, n);
int n;
{
        int add ;
        add = n;
        if(n==1)
                return(1);
        else
                add = add+ fact(n -1);
        return(add);
}
```

b)

```
float mul(float, float);
float quo(float, float);
main()
{
        float a, b, prod, div;
        printf("Enter values of a and b :");
        scanf("%f%f", &a, &b);
        prod = mul(a,b);
        div = quo(a,b);
        printf("\nThe product is : %.2f", prod);
        printf("\nThe quotient is : %.2f", div);
}
        float mul(float p, float q);
        {
                float n;
```

```
        n = p*q;
        return(n);
    }
    float quo(float i, float j)
    {
        float k;
        k = i/j;
        return(k);
}
```

8.5 8.6

1. a)

```
int fnsmall (int [], int);
main()
{
    int arr1[10];
    int i, j;
    printf("Enter elements of the array :");
    for(i = 0; i < 10; i++)
        scanf("%d", &arr1 [i]);
    j = fnsmall(arr1, 10);
    printf("\nThe smallest number is : %d", j);
}
int fnsmall(int num [], int k)
{
    int min,I;
    min = num[0];
    for(I = 1; I < k; I++)
    {
        if(min > num[I])
            min = num[I];
}
return(min);
}
```

b)

```
#include "stdarg.h"
#include "stdio.h"
void fnrev (char[], int);
main()
{
    char arr1[25];
    int i,k;
    printf("Enter elements of character array :\n");
    i = 0;
    while((arr1[i] = getchar()) != '\n')
        i++;
    arr1[i] = '\0';
    fnrev(arr1,i);
}
```

```
fnrev(char, charr[], int);
char charr[];
int len;
{
        int I;
        for(I = len-1; I>= 0; I—)
                putchar(charr[I]);
}
```

c)
```
int mul (int, int);
main()
{
        int a, b, c, d, prod;
        printf("\nEnter values of a,b,c,d :");
        scanf("%d%d%d%d", &a, &b, &c, &d);
        prod = mul(a,mul(b, mul(c,d)));
        printf("\nThe product is : %d", prod);
}
int mul(int p, int q)
{
        return(p * q);
}
```

d)
```
float fnsum (float [], int);
float fnavg (float [], int);
main()
{
        float fnavg(), fnsum();
        float num[10], avg;
        int i;
        printf("Enter numbers to average :\n");
        for(i=0; i< 10; i++)
                scanf("%f", &num[i]);
        avg = fnavg(num, 10);
        printf("\nThe average is : %.2f", avg);
}
float fnavg(n1 ,j)
float n1[];
int j;
{
        float sum;
        sum = fnsum(n1,j);
        return(sum/j);
}
float fnsum(arr,k)
int k;
{
```

```
        int I;
        float z;
        z = 0;
        for(I = 0; I< k; I++)
                z = z + arr[I];
        return(z);
}
```

8.9 QUESTIONS FOR SELF-STUDY

1. Answer the following :

a) What is a function? What are the types of functions?

b) Describe the methods of declaring function arguments.

c) What is meant by a called function and a calling function? Explain with example.

d) What are formal arguments and actual arguments?

e) What are the function catagories? Explain the general form of a function with arguments and no return values.

2. Write short notes on :

a) The return values & their types

b) Scope rules of function

3. How can you pass array as arguments to functions?

4. What is meant by nesting of functions?

5. Describe the advanced features of functions.

3. Write C programs for the following :

a) Write a function to calculate the area of a circle and return it to the calling function.

b) Pass an array of type int to a function which sets the positive and negative elements in two separate arrays.

c) Write a function to print all numbers divisible by 3 between 300 to 400.

d) Write a function to find the length of a string and call it from main. The function should return the length to main.

8.10 SUGGESTED READINGS

Programming in ANSI C : Balguruswamy

Exploring C : Yashwant Kanitkar

C for Beginners : Madhusudan Mothe

CHAPTER 9

POINTERS

9.0 OBJECTIVES

Friends,

The study of this chapter will help you to

- state what pointers are
- explain how to declare pointers and their usage to access variable values
- state pointer arithmetic with examples
- describe the usage of pointers with functions, arrays and strings
- discuss pointers to pointers
- explain dynamic memory allocation and C functions related to dynamic memory allocation.

9.1 INTRODUCTION

Pointers Overview :

Pointers are an important feature of the C language. To understand pointers let us revise certain points about computer memory. You already know that computers store the data and instructions in memory. The computer's memory is a sequential collection of storage cells. Each cell is known as a **byte** of memory. Each cell also has a unique address associated with it. This address is a number. Generally the addresses given to memory locations are numbered sequentially.

Whenever a variable is declared in a program, the system allocates memory to hold the value of the variable. Each byte has a unique memory address, therefore each variable also has a unique address associated with it. eg.

int i = 10;

This declaration reserves space in memory to store the value of the variable i. The name i gets associated with that particular memory address. The value 10 which has been assigned to the variable i gets stored in that particular memory location. We can represent the location of the variable in memory as follows :

Variable_name	**i**
value	**10**
address	**3245**

Let us assume that the computer has allocated the address location 3245 to store the value of the integer variable i. Thus, the variable i gets an address associated with it to store its value. It is possible for us to determine the address of a variable in memory. It can be done as follows :

Example : To determine the address of a variable in memory :

```
main()
{
        int i = 10;
        printf("\nValue of I :", i);
        printf("\nAddress of i:" &i);
}
```

The output is:

Value of I: 10

Address of i :3245

It is clear from the above example that the address of i is obtained with the expression **&i**. We make use of the **address operator (&)** to determine the address of the variable i. Thus the memory address of a variable is a number which is always positive. Since the memory address is a number we can as well store it in memory like any other variable. This is where **pointers** come into picture. In all the programs we have written so far we have made use of the address operator & when using **scanf** to read the values of the variables. The same address operator can be used to determine the address of the corresponding variable in memory.

Thus, pointers are nothing but variables used to hold memory address. A pointer is a variable which contains the address of another variable. It is therefore possible to access the value of a variable either with the help of the variable name or with its address.

Example : Let us write a program to further understand the concept of pointers :

```
main()
{
        int a;
        char ch1;
        float b;
        a = 100;
        ch1 = 'Z';
        b = 40.85;
        printf("\nThe value of a is %d:", a);
        printf("\nThe address of a is :", &a);
        printf("\nThe value of ch1 is %c:", ch1);
        printf("\nThe address of ch1 is :", &ch1);
        printf("\nThe value of b is %f:", b);
        printf("\nThe address of b is :", &b);
}
```

The output of the program is :

The value of a is : 100

The address of a is : 6540

The value of ch1 is : Z

The address of ch1 is : 3400

The value of b is : 40.85

The address of b is : 5284

It is important to note here that the addresses of the variables that are output here may not match with the output you get and that every time you run this program, the compiler may assign different memory locations to the variables and you may get different addresses. We can also make use of the **%u** operator to print addresses since addresses are **unsigned integers.**

Since the address of a variable is a number we can as well store it in another variable. Let us use a variable j to store the address of the variable i.

The address of i can be stored in j as

j = &i;

But every variable in C has to be declared before we can use it. Since we are using j for the purpose of storing the address of i we make use of the operator **'*'** to declare it. This is called the **value at address** operator.

We declare j as follows :

int*j;

This declaration tells the compiler that j is a variable used to store the address of an integer value. i.e. j points to an integer.

j itself is a variable and so will have its own unique address associated with it. The value at j is the address of the variable i. This can be depicted as follows :

Variable	i	j
Value	10	3245
Address	3245	4000

Declaring a pointer variable :

The general form of declaring a pointer variable is as follows :

*data type *pointer_name;*

In this declaration -

the * means it is a pointer variable

pointer_name is the name given to the pointer variable and it being a variable needs space in memory.

The data type indicates the type of the data to which the pointer variable points.

eg.

int *a;

This declares that the pointer variable points to an integer data type.

char *ch1;

Here the pointer variable points to a character data type.

float *p;

declares p as a pointer to a floating point variable.

When you declare a pointer variable you have to initialise it by assigning to it the address of a variable eg.

int *p. i;

p = &i;

Thus p is initialised and now contains the address of i. Pointers should not be used before they are initialised. When the type of a pointer is declared it will hold the address of that data type only.

```
eg.   int *i, j;
      float p;
      i = &p;
```

is invalid since p is declared **float** and the data type in the pointer declaration is declared to be of type **int.**

A pointer variable can also be initialised at the time of declaration also as follows:

```
float a, *z = &a;
```

Here a is declared to be of type **float.** Pointer variable z is also declared to hold address of data type **float** and hence can hold the address of a. (Note that a has to be first declared before assigning its address to z i.e. the statement float *z = &a, a; is invalid. Absolute value cannot be assigned to any pointer variable. Thus the following is invalid :

```
int *p;
p = 100;
```

Having seen how to declare and initialise pointer variables let us now see how to make use of the pointer variable to access the value of a variable.

Example : To determine the value of a variable using a pointer.

```
main()
{
      int i, *ptr, val;
      i = 100;
      ptr = &i;
      val = *ptr;
      printf("\nThe value of i is %d", i);
      printf("\nThe value of i is %d", *ptr);
      printf("\nThe value of i is %d", *&ptr);
      printf("\nThe value of i is %d", val);
      printf("\nThe address of i is %u", &i);
      printf("\nThe address of i is %u", ptr);
}
```

The output of the program will be :

```
The value of i is 100
The value of i is 100
The value of i is 100
The value of i is 100
The address of i is 65496
The address of i is 65496
```

The program demonstrates the various ways which can be used to determine the value of a variable. The statement val = *ptr; returns the value of the variable whose address is stored in ptr to val. Thus *ptr returns the value of the variable i.

```
ptr = &i;
val = *ptr;
```

can be combined and written as :

```
val = *&i;
```

Remember that val will have the same value as the value of the variable i. Study thoroughly the concept of pointers before proceeding to the further topics.

9.1 Check Your Progress.

1. What will be the output of the following :

a)
```
int i = 10, *j;
j = &i;
printf("%d\t%u", i, &i);
```
.....

.....

b)
```
float f = 15.3, *ptr = &f;
printf("%u\t%f", &f, f);
```
.....

.....

2. Are the following valid ?

a)
```
int *p;
p = 100;
```
.....

.....

b)
```
float *j;
int i;
j = &i;
```
.....

.....

c)
```
int i, *j = &i;
```
.....

.....

d)
```
char ch1, *cptr;
int i;
cptr = i;
```
.....

.....

9.2 POINTER ARITHMETIC

Pointer variables can be used in expressions

eg.

```
float x, p, q, z, *ptr1, *ptr2;
ptr1 = &p;
ptr2 = &q;
```

Here ptr1 and ptr2 are pointer variables which point to variables of type **float**. They are therefore initialised with the addresses of **float** variables p and q respectively.

then

(i)
```
x = *ptr1/ *ptr2;
*ptr1 = *ptr2 – 10;
z = *ptr1 x 10;
```
are valid.

Example:

```
main()
{
        float a, b, *p1 = &a, *p2 = &b;
```

```
        float z;
        a = 100;
        b = 21.8;
        printf("\nThe value of a is %6.2f", a);
        a = *p1 * 10;
        printf("The new value of a is %6.2f", a);
        z = *p1/*p2;
        printf("The value of z is %6.2f", z);
        z = *p1-*p2;
        printf("The new value of z is %6.2f", z);
}
```

The output of the program will be:

The value of a is 100.00

The new value of a is 1000.00

The value of z is 45.87

The new value of z is 978.20

Note : *When using '/' (division operator) in pointer arithmetic remember to have a space between the/and * else /* will be mistaken as a comment. Thus write *ptr1/ *ptr2 and not *ptr1/*ptr2. With the above example it is clear that with the use of pointers we have been able to manipulate the values of variables.*

(ii) Pointers can also be incremented or decremented. Thus

ptr1 ++ or

ptr2 -- are valid in C.

In this case, it is important to understand what happens when pointers are incremented or decremented. ptr++ will cause the pointer ptr1 to point to the next value of its type. Thus if a is an integer and ptr1 is a pointer to a, then when ptr is incremented its value will be incremented by 2 since an integer is stored in 2 bytes of memory. Thus if the pointer ptr1 has an initial value 4820 then ptr++ will cause its value to be 4822 i.e. its value is incremented by the length of the data type to which it points. This length is called the scale factor.

For the purpose of revising let us once again see the various data types and the number of bytes they occupy in memory

int	2 bytes
char	1 byte
float	4 bytes
long int	4 bytes
double	8 bytes

Example : To demonstrate increment and decrement of pointers

```
main()
{
        int a, *ptr1;
        float b, *ptr2;
        ptr1 = &a;
        printf("\nptr1 is : %u", ptr1);
        ptr1++;
        printf("\nnew ptr1 is %u", ptr1);
        ptr2 = &b;
        printf("\nptr2 is : %u", ptr2);
```

```
        ptr2—;
        printf("\nnew ptr2 is %u", ptr2);
}
```

The output of the program is :

ptr1 is : 65492

new ptr1 is : 65494

ptr2 is : 65494

new ptr2 is : 65490

In this program ptr1 points to an integer variable whereas ptr2 points to a **float** variable. Incrementing ptr1 causes its value to increase by 2 (since int occupies 2 bytes in memory). Decrementing ptr2 causes its value to be decremented by 4 since a **float** occupies 4 bytes in memory.

(iii) Point (ii) can be extended as follows :

C also allows us to add integers to pointers. eg.

ptr2 + 4

ptr1+10

C also allows to subtract integers form pointers :

ptr1 -10

ptr2 - 2

Subtraction of pointers is allowed in C

p1-p2

Example : To add and subtract integers from pointers :

```
main()
{
        int a, b, *ptr1 = &a, *ptr2 = &b;    int q = 10, b = 20;
        printf("\nThe value of ptr1 is %d", ptr1);
        *ptr1 = *ptr1 + 10;
        printf("\nThe new value of ptr1 is %d", *ptr1);
        printf("\nThe value of ptr2 is %d", *ptr2);
        *ptr2 = *ptr2 - 40;
        printf("\nThe new value of ptr2 is %d", *ptr2);
        *ptr2 = ptr2 - ptr1;
        printf("The new value of ptr2 is %d", *ptr2);
}
```

Check the output of this program and study what happens to the values of ptr1 and ptr2 when integers are added to and subtracted from them.

(iv) Pointers can also be compared as :

ptr1 > ptr2

ptr2 < ptr1

ptr1 == ptr2

ptr1 != ptr2 and so on. It is however important to note that pointer variables can be compared provided both variables point to objects of same data type.

(v) **Remember that you cannot** -

- add two pointers

ptr1 + ptr2 is invalid

- use pointers in multiplication

ptr1 * ptr2 is invalid
ptr2 * 10 is invalid

- use pointers in division
ptr1/20 is invalid
ptr2/ptr1 is invalid

9.2 Check Your Progress.

1. Write True or False :

a) Pointers cannot be used in expressions.

b) Pointers can be added to integers.

c) Multiplication of pointers is valid in C.

d) Pointers can be assigned negative values.

2. Answer in 1-2 sentences :

a) What cannot be done with pointers in pointer arithmetic?

.....

.....

b) What happens when pointers are incremented?

.....

.....

9.3 POINTER AND FUNCTIONS

Having obtained an overview of pointers and learnt pointer arithmetic let us now learn the use of pointers in functions. We had deferred our discussion of **call be reference** in functions till we studied pointers. Let us now see this aspect of functions.

Example : Program to exchange the values of i and j:

Let us write the program to swap the values of i and j first by using the **call by value** method and then study the **call by reference** method :

Call by value:

```
main()
{
        int i,j;
        i = 10;
        j = 50;
        printf("\ni = %d\tj = %d", i, j);
        swap(i,j);
        printf("\ni = %d\t j = %d", i,j);
}
swap(int a, int b)
{
        int z;
        z = a;
        a = b;
        b = z;
        printf("\na = %d\t b = %d", a, b);
```

```
}
```

The output of the program is :

```
i = 10          j = 50
a = 50          b = 10
i = 10          j = 50
```

Note that in this function the values of **i** and **j** remain unchanged, only the values of a and b get interchanged. As we have already studied, the values of the actual arguments merely get copied into the corresponding formal arguments of the called function. Thus changes made to the formal arguments have no effect on the values of the actual arguments.Thus even though you manipulate the formal arguments the actual arguments remain unchanged.

9.3.1 Call by reference :

In call by reference we pass not the values but the addresses of the actual arguments to the formal arguments of the called functions. The formal arguments are declared to be pointer variables to accept the actual arguments. This implies that with these addresses of the actual arguments we can have access to the actual arguments and be able to manipulate them. We can thus change the values of the actual variables with this technique. The following example rewrites the above program by making use of call by reference and thus actually changing the values of i and j. It illustrates how to pass the addresses as arguments to the called function :

Example : To actually swap i and j using call by reference

```
void swap(int *, int *)
main()
        {
        int i,j;
        i = 10;
        j = 50;
        printf("\ni = %d\tj = %d", i, j);
        swap(&i,&j);
        printf("\ni = %d\t ,j = %d", i, j);
}
void swap(int *ptr1, int *ptr2)
{
        int k;
        k = *ptr1;
        *ptr1 = *ptr2;
        *ptr2 =k;
}
```

The output of the program willbe :

```
i = 10 j = 50
i = 50 j = 10
```

Here, the addresses of the variables i and j are copied into the formal arguments *ptr1 and *ptr2 of the called function. With the help of these addresses we have access to the actual values of i and j. Thus the values of i and j are exchanged.

Another example is given below which changes the actual value of a variable. Study it carefully.

Example :

```
/* Program to demonstrate call by reference */
main()
{
        int i;
```

```
        i = 10;

        printf("\nValue of i is :%d", i);
        fn1(&i);
        printf("\nNew value of i is : %d", i);
}

void fn1(p)
int *p;
{
        *p = *p + 10;
}
```

The output of the program is :

```
Value of i is :10
New value of i is :20
```

In this example, when the function fn1() is called the address of i (and not the value of i) is passed to fn1(). In the function fn1() the variable ptr is a pointer variable which points to data type **int** and so it receives the address of i.

```
*p =*p +10;
```

means that 10 gets added to the value which stored at the address p. Hence 10 gets added to i. This means that call by reference actually allowed you to change the values of the variables of the calling functions.

We have studied that the **return** statement can return only one value from the called function. However we can make a function return more than one value by making use of call be reference method. Let us see how this can be done :

Example : To make a function return more than one value :

```
void fn1(int, int, int *, int * );
main()
{
        int a, b, sum, prod;
        printf("\nEnter a :");
        scanf("%d", &a);
        printf("\nEnter b:");
        scanf("%d", &b);
        fn1(a,b, &prod, &sum);
        printf("\nThe product of a and b is %d", prod);
        printf("\nThe sum of a and b is %d", sum);
}
void fn1(int i, int ,j, int *p, int *s)

{
        *p = i*j;
        *s = i + j;
}
```

A sample output:

```
Enter a : 50
Enter b : 10
```

The product of a and b is : 500

The sum of a and b is : 60

The above example makes efficient use of passing addresses to functions so that the function returns the sum and product of the numbers a and b. We pass the addresses of the variables prod and sum as parameters to fn1(). fn1() collects these addresses in the pointer variables p and s. The values at these variables are then calculated as the product and sum respectively and are then available in the calling function. Thus two values are returned to the calling function from the called function.

Example : To make a function return more than one value :

```
void fn1 (int, float*, float*);
main()
{
        int radius;
        float circum, area;
        printf("\nEnter radius :");
        scanf("%d", &radius);
        fn1 (radius, &area, &circum);
        printf("\nArea is : %6.2f", area);
        printf("\nCircumference is : %6.2f", circum);
}
void fn1(int r, float *a, float *c)
{
        *a = 3.14*r*r;
        *c = 2*3.14*r;
}
```

A sample run of the program :

Enter radius : 7

Area is : 153.86

Circumference is : 43.96

In this example, our function calculates both the area and circumference of the circle. We are passing the radius and along with it the addresses of the variables **area** and **circum** to the function fn1 (). Therefore area and circum are calculated and their values can be obtained with the help of their addresses.

9.3.1 Check Your Progress.

1. Answer the following 1- 2 lines

a) Compare call by value and call by reference.

.....

.....

2. Write C programs for the following :

a) Use call be reference method to calculate the sum, product, difference and quotient of two numbers a and b and make the function return all these values.

b) Use call by reference method to write a function to calculate the value of a^b. Print the value from main().

9.3.2 Pointers to Functions :

Functions also have an address location in memory. Hence, we can declare a pointer to a function. A pointer to a function is declared as follows:

*type (*fnptr)();*

fnptr is a pointer to a function which returns a value of *type.* The pointer is to be enclosed in a pair of parenthesis.

A function pointer can be used to point to the specific function by assigning the name of the function to the pointer.

eg.

```
float (*fnptr)();

float add();

fnptr = add;
```

will declare fnptr as a function pointer and intialised it to point to function add. Now we can use fnptr to call the function add as :

```
(*fnptr)(a, b);
```

(Note that there is a parenthesis around *fnptr).

This is as good as calling function add():

```
add(a,b);
```

9.3.2 Check Your Progress.

1. Write in short on pointers to functions.

.....

.....

9.4 POINTERS AND ARRAYS

We have already seen that when we declare an array the elements of the array are stored in contiguous memory locations. In pointers we have studied that whenever we increment a pointer it points to the next memory location of its type. Using this, let us now study pointers and arrays.

When an array is declared, the compiler immediately allocates a base address and sufficient memory to hold all the elements of the array in contiguous memory locations. The base address is the address of the first element (i.e. 0th index) of the array. The compiler also defines the array name as a constant pointer to the first element of the array.

9.4.1 Pointers to one dimensional arrays :

Let us study about pointers and one dimensional arrays with the help of the following example:

Suppose you declare an array as follows :

```
int arr[5] = { 10, 20, 30, 40, 50};
```

arr is an array of type **int** whose size is five and the elements of the array are initialised in the declaration. Let us assume that the base address of arr is 2400. Then since each element of the array is of type **int** and there are five elements in the array the five elements will be stored as follows

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	← Array Elements
10	**20**	**30**	**40**	**50**	← Values
2400	2402	2404	2406	2408	← Addresses of Elements

arr is a constant pointer which points to the first element arr[0]. Thus the address of arr[0] in our example is 2400.

We can declare an **int** pointer ptr to point to the array arr as

ptr = arr;

or

ptr = &arr[0];

We already know that when we use the increment operator with the pointer, its value gets increased by the length of the data type to which it points. Here ptr points to data type **int,** therefore incrementing ptr will cause its value to increase by 2 and hence it will point to the next element of the array which is arr[1]. Thus it is possible to obtain the addresses of all the elements of arr[] as follows :

ptr	=&arr[0]	=2400
ptr + 1	=&arr[1]	=2402
ptr + 2	=&arr[2]	=2404
ptr + 3	=&arr[3]	=2406
ptr + 4	=&arr[4]	=24

Thus you can obtain the address of the element as :

address of n^{th} element = base address + *(n x scale factor of data type)*

In our case we can determine the address of the 4th element as :

address of the 4th element = 2400 + (4 x scale factor of int)
= 2400 + (4 x 2)
= 2408

We can use pointers to access the elements of an array. Thus we can access arr[3] as *(ptr+3), arr[2] as *(ptr + 2) and so on. The pointer accessing method is very fast as compared to accessing by the index number as arr[2], arr[4] etc.

Example : A program to obtain the addresses of all elements in the array:

```
main()
{
        int arr[5] = {10,20, 30, 40, 50};
        int i, *ptr;
        ptr = &arr[0];
        for (i = 0; i <5; i++)
        {
                printf("Element: %d\tAddress : %u", *ptr, ptr);
                ptr++;
        }
}
```

A sample run would give the following :

Element: 10	Address : 65488
Element: 20	Address : 65490
Element: 30	Address : 65492
Element: 40	Address : 65494
Element: 50	Address : 65496

In the above program note that we have not used indexing to access the elements of the array. Instead we have incremented the pointer ptr everytime so that it points to the next memory location of its type. Accessing array elements with pointers is always faster as compared to accessing them by subscripts. This method can be very effectively used if the elements are to be accessed in a fixed order according to some definite logic. If elements are to be accessed randomly, then using subscripts would be easier though not as fast as accessing them using pointers.

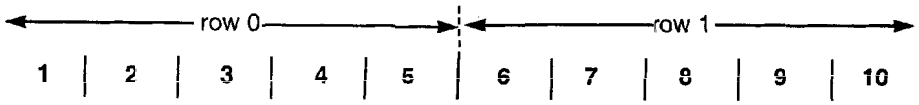
9.4.2 Pointers to 2 dimensional arrays :

We already know that elements of a two dimensional array are stored row wise. Thus the base address of a two dimensional array arr[] is the address of the arr[0][0]th element which will be obtained as &arr[0][0]. The compiler then allocates contiguous memory locations to the array elements row wise i.e first all the elements of row 1 are stored then all elements of row 2 and so on. Let us see this representation with the help of the following example :

```
int arr[2][5] = {(1,2,3,4,5),
                 (6,7,8,9,10)
                 };
```

The elements will be stored in memory rowwise as :

If we want to access the element arr[1][2] we can do it as :



Representation of elements of a 2-dimensional array

```
arr[1][2]    = *(ptr + 5 x1 +2)
             = *(ptr + 7)
             = 8
```

where ptr points to the base address of the array.

Thus to access an element arr[i][j] the formula would be :

a[i][j] = *(ptr + no. of cols x i + j)

Hence it is essential to define the number of columns i.e. size of each row when declaring a two dimensional array so that the compiler can determine the storage mapping for the array elements.

9.4.3 Pointers and strings :

A string is an array of characters which is terminated by the null character “\0'. Thus the concept of pointers and one dimensional arrays can be extended to array of characters. Let us write a program to determine the values of the elements of the character array with the help of pointers.

Example : To access elements of a string with pointers

```
main()
{
        char str1[25] = “Pointers”;
        char *cp;
        cp = &str1[0];
        while(*cp != ‘\0’)
        {
                printf(“\nCharacter :%c\tAddress : %u”, *cp, cp);
                cp++;
        }
}
```

The output of the program will be :

Character: P Address : 65472

Character: o Address : 65473

Character: i Address : 65474

Character: n Address : 65475

Character: t Address : 65476

Character: e Address : 65477

Character: r Address : 65478

Character: s Address : 65479

Since characters require one byte of storage in memory, incrementing pointer cp will increment its value by 1 and it will point to the next character in the string.

The concept of single dimension array of characters i.e. string can be extended to the table of strings. When we declare a two dimensional array of strings, each string will be allocated equal length as specified in its declaration. However, in practice the all strings of the table are rarely equal in length. Hence instead of making each row of a fixed number of characters we can make each row a pointer to a string of varying lengths.

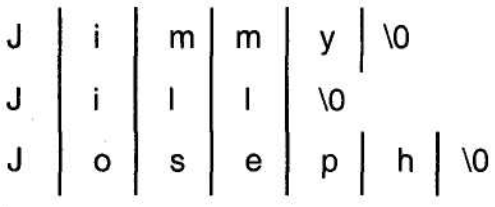
```
eg. char *name[3] =
                    {
                        "Jimmy";
                        "Jill";
                        "Joseph"
                    };
```

The above declaration declares name to be an array of three pointers where each pointer points to the particular name. This can be shown as follows :

```
name[0]---------->  Jimmy
name[1]---------->  Jill
name[2]---------->  Joseph
```

Had we declared name to be a two dimensional array of strings as name[3][10] it would have reserved 30 bytes for the array name, where each name would be allocated 10 bytes. But when we declare name to be an array of pointers, where each element points to a name, the total memory allocated would be 18 bytes as follows

:



In order to access the jth character of the ith row :

*(name[i] + j) would be useful. Note that we first select the row, then the jth element of that particular row and then determine value at address.

We can print the names in the array as shown in the following example:

Example : To demonstrate an array of pointers to strings.

```
main()
{
```

```
        int i;
        char *name[3] = {
                        "Jimmy",
                        "Jill",
                        "Joseph"
                    };
        printf("\nNames in the array :");
        for(i=0; i<3; i++)
            printf("\n%s", name[i]);
}
```

9.4 Check Your Progress.

1. Write the formulae for accessing the following with pointers:

a) The nth element of a one dimensional array of float:

.....

.....

b) The ith character of a string

.....

.....

c) The a[i][j]th element of the array two dimensional array a[][].

.....

.....

2. Write programs in C for the following using pointers :

a) Find the average of elements of array a[5] of type int.

b) Reverse the string "Pointers&Arrays" using pointers to strings.

c) Write a program using pointers to input elements to an integer array and print them in the reverse order.

9.5 POINTERS TO POINTERS

We know that a pointer variable contains the address of a data type. Since the addresses are always whole numbers the pointers will always contain whole numbers. The declaration char *ch does not imply that ch contains a data type **char.** It implies that ch is a pointer to a data type **char,** i.e ch contains the address of a **char** variable, i.e. the value pointed to by ch is of type-**char**.

The concept of pointers can thus be extended further. A pointer contains the address of a variable. This variable itself can be a pointer. We can have a pointer to contain the address of another pointer.

eg.

```
int. i, *j,**k;
j = &i;
k = &j;
```

i is an **int** data type and j is a pointer to i. k is a pointer variable which points to the integer pointer j. The value at k will be the address of j. In principle, there is no limit how far you can extend the concept of pointers to pointers. You can further have another pointer to point to k and so on.

Example : Let us write a small program to illustrate pointers to pointers:

```
main()
```

```
{
        char ch, *ch_ptr;
        int **ptr;
        ch = 'A';
        ch_ptr = &ch;
        ptr = &ch_ptr;
        printf("\nCharacter is : %c :", ch);
        printf("\nAddress of ch is : %u", ch_ptr);
        printf("\nValue of ch_ptr is : %u", ch_ptr);
        printf("\nAddress of ch_ptr is : %u", ptr);
        printf("\nValue of ptr is : %u", ptr);
        printf("\nCharacter is :%c", *ch_ptr);
        printf("\nCharacter is :%c", **ptr);
}
```

Carefully follow the program to see how to obtain addresses and values at addresses with the help of the above program. Also note that although ch is of type **char** and ch_ptr is a pointer to ch, the pointer ptr to ch_ptr is declared of type **int,** since it is going to hold the address of ch_ptr which is always going to be a whole number.

9.5 Check Your Progress.

1. Write the declaration and initialisation for the following :

a) A pointer to contain the address of a data type float:

.....

.....

b) A pointer to contain the address of a data type int and another to contain the address of this pointer:

.....

.....

c) A pointer to contain the address of a data of type char.

.....

.....

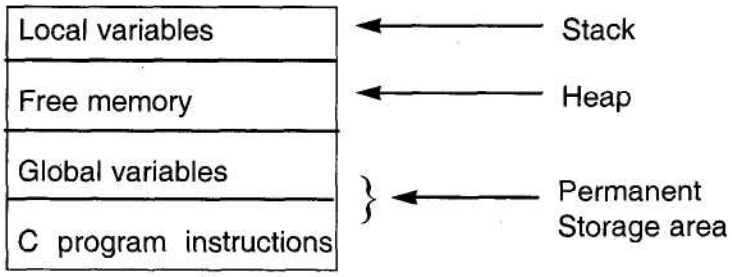
9.6 DYNAMIC MEMORY ALLOCATION

In real life programming situations, many times it so happens that the number of data items keeps on changing during program execution, eg. if you are processing a list of travellers, new passengers may get added to the list as also passengers may be cancelled. Such data which is subject to change during program execution is said to be dynamic in nature. In order to handle such type of data efficiently and easily we make use of dynamic memory management. Structures which are dynamic in nature provide flexibility to add, delete and rearrange data items at run time. They make is possible to free unwanted memory and allocate new memory as and when required. This section will introduce the dynamic storage management functions in C.

The process of allocating memory to data items during run time is called as dynamic memory allocation. C has four library routines which are useful for allocating and releasing memory during the program execution. These functions make intelligent use of memory. These functions are :

Function		Use
malloc()	-	Allocates required memory in bytes and returns a pointer to the first byte of the space allocated.
calloc()	-	Allocates space for an array of elements, initialises them to zero and returns the pointer of the first byte of memory.
free	-	frees previously allocated memory
realloc	-	modifies the size of the previously allocated memory.

The following figure shows how a C program is stored in memory :



From the figure, it is understood that the global variables and the C program instructions are stored in what is known as the permanent storage area. The local variables are stored in an area called the stack. In between these two, free memory available is called as the heap. This is the area which is used for dynamic memory allocation. The heap size keeps on changing during program execution. When local variables are created and they die, the memory is occupied and gets freed respectively.

If it so happens that the heap becomes full then the memory allocation functions return NULL.

Let us now study these functions one-by one :

malloc():

Memory can be allocated using the **malloc()** function.

The general form of malloc() is :

ptr = (cast-type)malloc(bytes);*

malloc() reserves a size of memory equal to bytes and returns a pointer of type void i.e. we can assign this pointer to any type of pointer. Thus ptr is a pointer of cast-type, eg.

ptr1 = (int*)malloc(25 * sizeof(int));

This declaration will allocate a memory equivalent to twenty five times the size of **int** in bytes, ptr1 is a pointer of type **int** and the address of the first byte of memory allocated will be assigned to ptr1.

ch_ptr = (char*)malloc(10);

will allocate 10 bytes of space for the pointer ch_ptr of type **char.**

The storage which is allocated dynamically has no name, hence it can be accessed only through pointers. **malloc()** can be used not only for simple data types but also complex data types like structures.

malloc() allocates contiguous bytes of memory. In the event there is not enough memory, it returns a NULL. Care should therefore be taken during program writing to ensure whether enough memory is available or not.

Example:

Let us write a program to dynamically allocate memory to a list of integers.

main()

```
{
        int *ptr, *p, n;
        printf("\nEnter number of elements :");
        scanf("%d", &n);
        ptr = (int*)malloc(n * sizeof(int));
        if (ptr == NULL)
        {
                printf("Not enough memory to allocate");
                exit(1);
        }
        printf ("Enter values :");
        for(p = ptr; p< ptr + n; p++)
        scanf("%d", p);
        printf("\nThe values input are :\n");
        for(p = ptr; p<ptr + n; p++)
        printf("Value = %d\tAddress = %u\n", *p, p);
}
```

A sample run:

```
Enter number of elements ; 3
Enter values :
100
200
300
The values input are :
Value = 100          Address = 2456
Value = 200          Address = 2458
Value = 300          Address = 2460
```

The for loop in the above program initialises pointer p to the address of the first element of the list and accepts elements till the number specified. Another for loop prints the elements of the list and their corresponding addresses.

calloc():

This function is generally used to allocate memory for derived data types like arrays and structures. **calloc()** allocates multiple blocks of storage, each of the same size. It then sets all the bytes to zero.

The general form of **calloc()** :

ptr = (cast-type) calloc(n, element-size);*

With this declaration, n contiguous blocks each of size element-size are allocated and all the bytes are set to zero. If there is not enough memory, a NULL pointer is returned.

eg.

```
int arr1[10];
int *ptr;
ptr = (arr1*)calloc(10, sizeof(arr1));
-------
-------
```

Here we have defined an array arr1 having ten elements of type **int**. We use the **calloc()** function to reserve 10 blocks of size each equal to the size of arr1. The pointer is returned to ptr of type **int.** It is important to check whether space has been made available or not before further execution.

eg.

```
if (ptr == NULL)
{
        printf("\nInsufficient Memory");
        exit();
}
```

free():

To release the used space in dynamic memory allocation, when it is not required can be done with the **free()** function. When we release the block of memory which we have allocated dynamically and no longer need it, it can become available for future use.

The general form of **free()** is :

free(ptr);

where ptr is a pointer to the memory block which has been dynamically allocated using **malloc()** or **calloc().**

realloc():

In situations, where the previously allocated memory is not sufficient and there is additional memory requirement, it is possible to reallocate memory for more data. Alternatively, it is also possible, that the memory allocated is much larger than required and we need to make free the excess memory which has been dynamically allocated previously. **realloc()** helps us to alter the original allocation of memory.

The form of **realloc()** is :

ptr = malloc(size);

is the statement with which we have dynamically allocated memory. To reallocate this memory we use the realloc as follows :

ptr = realloc(ptr, newsize);

Remember that **realloc()** will also return a pointer to the first byte of the reallocated memory block. The newsize will be the memory space allocated to the pointer. newsize may be larger or smaller. Also, it is important to note that the new block may or may not begin at the same place as the old block. If contiguous additional space is not available, then **realloc()** will create an entirely new memory space and move the contents of the old block to the new block. A NULL will be returned if the function fails to locate additional space and the original block is lost.

Example : To first allocate memory for five integers in a list and then use **realloc()** to make additional space for 3 more integers :

```
#include "stdlib.h"
main()
{
        int *ptr, *p;
        ptr = (int*)malloc(5 * sizeof(int));
        if (ptr == NULL)
        {
                printf("No space");
                exit(1);
        }
        printf("Enter values :");
        for(p = ptr; p< ptr + 5; p++)
                scanf("%d", p);
        for(p = ptr; p < ptr+5; p++)
                printf("Value = %d\tAddress = %u\n", *p, p);
```

```
        ptr = (int*) realloc(ptr, 8 *sizeof(int));
        if (ptr == NULL)
        {
                printf("Reallocation Failed");
                exit(1);
        }
        printf("New block reallocated succesfully");
        printf("\nNew block contains :\n");

        for(p = ptr; p < ptr+5; p++)
                printf("Value = %d\tAddress = %u\n", *p, p);

        printf("'Enter new values for the reallocated block");
        for(p = ptr; p < ptr + 8; p++)
                scanf("%d", p);
        printf("New values in the list \n");
        for(p = ptr; p< ptr + 8; p++)
        printf("Value = %d\tAddress = %u\n", *p, p);
}
```

First we make use of **malloc()** to dynamically allocate memory for a list of integers which contains 5 values. If the function fails it will return NULL and the program will exit. On the other hand, if it is able to allocate memory it will prompt the user to input the list of five integers. We have made use of the integer pointer p to read the elements of the list. We then print the elements and their corresponding addresses. Now we make use of **realloc()** to allocate memory to hold such 8 data items. If reallocation is successful, the old data will remain intact and we print it. If reallocation fails it returns NULL and the program is exited and data lost. We can now read in new values in the newly reallocated block. These are then entered and printed. Remember that you can allocate more memory or less memory using **realloc().** Test the program for various sample data items.

9.6 Check Your Progress.

1. Write the use of the following functions :

a) realloc():

.....

.....

b) free():

.....

.....

2. Write True or False :

a) There is no way of knowing whether malloc() has allocated memory or not.

.....

b) malloc() and realloc() both allocate memory dynamically.

.....

c) The free() function is used to release memory allocated dynamically when it is no longer required.

.....

d) A dynamically allocated storage can be given a name.

.....

e) Dynamically allocated storage need not store data in contiguous memory locations.

.....

9.7 SUMMARY

In this chapter we studied the new and powerful concept of pointer. Pointers are particularly important in system programming.

Pointers is contains the addresses of another variable. It is therefore possible to access the value of a variable either with the help of variable name or with its addresses.

Pointers can be used in all places where arrays are used.

We can invoke functions using pointers

Dynamic allocation is allocating memory to data items during run time. C has four library functions which are useful for allocating memory. They are malloc( ) , calloc( ), free( ), realloc( ). These functions make intelligent use of memory.

9.8 CHECK YOUR PROGRESS - ANSWERS

9.1

1. a) 10 65486

b) 65388 15.3

(Note : You may get different addresses than the above ones.)

2. a) Invalid

b) Invalid

c) Valid

d) Invalid

9.2

1. a) False

b) True

c) False

d) False

2. a) In pointer arithmetic it is not possible to

(i) add two pointers

(ii) use pointers in multiplication

(iii) use pointers in division

b) When a pointer is incremented the pointer points to the next value of its type i.e. its value is incremented by the length of the data type to which it points. This length is called the scale factor.

9.3.1

1. a) In call by value, we pass the values of the arguments to the called function from the calling function. Here the values of the actual arguments merely get copied into the corresponding formal arguments of the called function. The changes made to the formal arguments have no effect on the values of the actual arguments.Thus even though you manipulate the formal arguments the actual arguments remain unchanged.

In call by reference we pass not the values but the addresses of the actual arguments to the formal arguments of the called functions. The formal arguments are declared to be pointer variables to accept the actual arguments. The implies that with these addresses of the actual arguments we can have access to the actual arguments and be able to manipulate them. We can thus change the values of the actual variables with this technique.

2. a) main()

```
{
        int a, b;
        float add, mul, div, diff;
        printf("Enter value of a :");
        scanf("%d", &a);
        printf("\nEnter value of b :");
        scanf("%d", &b);
        fn1(a, b, &add, &mul, &div, &diff);
        printf("\nThe sum is : %.2f, add);
        printf("\nThe product is : %.2f", mul);
        printf("\nThe quotient is : %.2f", div);
        printf("\nThe difference is : %.2f", diff);
}
fn1(int i, int j, float *sum, float *prod, float *quo, float *sub)
{
        printf("i = %d\tj = %d\n", i, j);
        *sum = i + j;
        *prod = i * j;
        *quo = i/j;
        *sub = i - j;
        printf("\nsum = %.2f", *sum);
}
```

b)

```
main()
{
        int a, b;
        int z;
        printf("Enter value of a :");
        scanf("%d", &a);
        printf("Enter value of b :");
        scanf("%d", &b);
        power(a,b,&z);
        printf("The value of a to the power b is : %d", z);
}
power(int a, int b, int *p)
{
        int i,j;
        j = 1;
        for(i = 1; i<=b; i++)
        j = j * a;
        *p = j;
}
```

9.3.2

1. Functions have an address location in memory. Hence, it is possible to declare a pointer to a function. A pointer to a function is declared as follows :

*type (*fnptr)();*

where *fnptr* is a pointer to a function which returns a value of *type.* The pointer is to be enclosed in a pair of parenthesis. A function pointer can be used to point

to the specific function by assigning the name of the function to the pointer.

eg. float (*fnptr)();

float add();

fnptr = add;

will declare fnptr as a function pointer and intialised it to point to function add. Now fnptr can be used to call the function add() with the following statement:

(*fnptr)(a, b);

9.4

1. a) n^{th} element = *(base address + (n * scale factor of float))

b) ith character of a string = *(base address + (i * 1));

c) value at a[i][j] = *(base address + no. of columns * i + j)

2. a)
```
main()
{
        int arr1[5], *ptr, sum, i;
        float avg;
        printf("Enter elements of array :");
        for(i = 0; i< 5; i++)
                scanf("%d", &arr1[i]);
        sum = 0;
        ptr = &arr1;
        for(i= 0; i < 5; i++)
        {
                sum = sum + *ptr;
                ptr++;
        }
        avg = (float)sum/5;
        printf("The average is :%.2f", avg);
}
```

b)
```
main()
{
        char str1[25],*p = &str1;
        strcpy(str1, "Pointers And Arrays");
        p = p + strlen(str1);
        printf("\nReverse string :\n");
        while (p >= &str1)
        {
        printf("%c", *p);
        p—;
        }
}
```

c)
```
main()
{
        int num[5], *p= &num, count;
        printf("\nEnter array elements :");
        count = 0;
        while(count < 5)
        {
```

```
            scanf("%d",&num[count]);
            count++;
            P++;
        }
        p—;
        while(p >= &num)
        {
            printf("%d\t", *p);
            p—;
        }
    }
```

9.5

1. a) float f, *p;
P = &f;

b) int i, *p, **k;
P= &i;
k = &p;

c) char ch, *ptr;
ptr = &ch;

9.6

1. a) The use of **realloc( )** is in situations, where the previously allocated memory is not sufficient and there is additional memory requirement. Alternatively, it is also possible, that the memory allocated is much larger than required and we need to make free the excess memory" which has been dynamically allocated previously. realloc() can also be used in such situations. Thus realloc() helps us to alter the original allocation of memory.

b) To release the used space in dynamic memory allocation, when it is not required we can use the **free( )** function. Thus, when we no longer need the block of memory which we have allocated dynamically we can release if with the function free() and it can become available for future use.

2. a) False

b) True

c) True

d) False

e) False

9.9 QUESTIONS FOR SELF-STUDY

1. Answer in 3-4 sentences:

a) What is a pointer?

b) What is the value at address operator?

c) What happens when a pointer is decremented?

d) What type of data does a pointer hold?

e) What is the base address of an array?

f) What does calloc() do?

g) What is dynamic memory allocation?

2. Write short notes on :

a) Pointers and strings
b) malloc()
c) Call by reference
d) Pointer arithmetic

9.10 SUGGESTED READINGS

Spirit of C : Mullish cooper

Exploring C : Yashwant Kanetkar

The C Programming language : Kernigham & Ritche

CHAPTER 10

STRUCTURES

10.0 OBJECTIVES

After, the study of this chapter you will be able to

- describe structures
- state the structure template to create arrays of structures
- expalin the use of structures with functions
- describe the use of pointers with structures
- explain the user defined data types in C

10.1 INTRODUCTION

We have seen that arrays can be used to store elements of the same data type such as **int** or **float.** If we wish to represent data items of different types arrays are not useful. This is where structures come into picture. Structures are derived data types. We make use of structures to represent a collection of data items of different types. Structures are useful for handling logically related data items. For example, the personal data of people like names, addresses, phone numbers etc., or data of students like their roll numbers, names, marks, grades etc.

We shall study how to define and use structures. An array of structures provides a very useful tool to maintain records. We can also pass structures as arguments to functions. These features shall also be studied with examples. How to handle structures with the help of pointers is also discussed.

10.2 DEFINING A STRUCTURE

A structure contains a number of data types grouped together. The data types can be similar or of different types. The structure definition and creation of the structure variable will be best understood with the help of an example. Let us consider an example of a database of students academic record. This database will consist of the

student roll no, his name, his marks out of 2 subjects and his percentage. A structure to represent this information can be declared as follows :

```
struct stud
{
        int roll;
        char name[30];
        int marks1;
        int marks2;
        float per;
};
```

The keyword **struct** declares the structure with the name stud. This name is called the **structure tag.** The structure stud is declared to consist of five fields the roll number of type **int,** the name of type array of **char,** marks1 and marks2 of type **int** and per of type **float.** These fields are called as **structure members** or **structure elements.** As you can see from the above declaration the members can be of different data types.

Thus the general form of a structure definition is

```
struct tag_name
{
        data_type member_1;
        data_type member_2;
        :
        :
};
```

Note that this declaration of a structure simply describes a **format** or a **template** of the structure. Once you have defined this structure data type you can declare one or more variables to be of that type. Thus we can now declare a structure variable or structure variables for the stud structure type as follows :

struct stud_db1, stud_db2, stud_db3;

This declaration will declare stud_db1, stud_db2, stud_db3 as three variables of data type **struct** stud. Each of these variables will have four members as specified in the template. Thus the entire declaration of a structure is represented as :

```
struct stud
{
        int roll;
        char name[30];
        int marks1;
        int marks2;
        float per;
};
struct stud stud_db1, stud_db2, stud_db3;
```

Another method is to combine the structure type declaration and structure variables in a single statement as shown below :

```
struct stud
{
        int roll;
        char name[30];
```

```
        int marks1;
        int marks2;
        float per;
} stud_db1, stud_db2, stud_db3;
```

or

```
struct
{
        int roll;
        char name[30];
        int marks1;
        int marks2;
        float per;
} stud_db1, stud_db2,stud_db3;
```

In the first declaration we have given a tag to the structure viz. stud, in the second declaration the tag has not been given to the structure; the variables are declared directly in the same statement as the declaration. This means that the use of the tag name is optional in structure declaration.

It is important to remember that the structures members themselves are not variables. The structure declaration does not tell the compiler to allocate any memory. It merely defines the form of the structure.

When we declare variables of type **struct,** the compiler allocates memory to each individual member. All the members are stored in contiguous memory locations. In the following declaration :

```
struct num
{
        int i; char ch;
        float f;
};
struct num num1
```

7 bytes of storage space will be made available to hold the structure members : 2 bytes for i, 1 byte for ch and 4 bytes for f.

When declaring a structure :

- The declaration is enclosed within a pair of braces and the closing brace must be followed by a semicolon.

- Each member has to be declared independently for its type and name in a separate statement in the structure template.

- The tag name of the structure is then used to declare structure variables in the program.

Usually in practice if structures are being used they are declared at the beginning of a program before even defining any variables or functions. They can also be declared outside **main**(). This makes the structure definition global and it can then be used by other functions also.

10.2.1 Initialising Structures :

Like the primary variables and arrays, structure variables too can be initialised at the time of declaration. We can initialise structure variables as follows :

```
struct stud
{
```

```
        int roll;
        char name[30]
        int marks1;
        int marks2;
        float per;
};
struct stud stud_db1 = {101, “John”, 60, 50, 55.0};
struct stud_db2 = {102, “Jane”, 80, 60, 70.0};
```

Let us determine the addresses of these structure elements (since we have studied that structure members are stored in contiguous memory locations) with the help of the following program :

Example:

```
main()
{
        struct  stud
{
        int roll;
        char name[30];
        int marks1;
        int marks2;
        float per;
};
struct stud stud_db1 = {101, “John”, 60, 50, 55.0};
printf(“\nAddress of roll number: %u”, &stud_db1 .roll);
printf(“\nAddress of name : %u”, &stud_db1.name);
printf(“\nAddress of marks1 : %u”, &stud_db1.marks1);
printf(“\nAddress of marks2 : %u”, &stud_db1 .marks2);
printf(“\nAddress of percentage : %u”, &stud_db1 .per);
}
```

A sample output of the program :

```
Address of roll number: 65456
Address of name : 65458
Address of marks1 : 65488
Address of marks2 : 65490
Address of per: 65492
```

Depending upon the number of bytes required for each data type the structure elements will be stored. Follow the output of the program carefully.

10.2.2 Accessing Structure Elements :

How to access the elements of a structure? In order to do this, we make use of the **dot operator** or a **period operator (.).**

eg. if we wish to refer to marks 1 of stud_db1 we do so as :

stud_db1 .marks

Similarly we access the name of stud_db2 as :

stud_db2.name

Note the syntax for accessing the structure elements. The structure variable should be followed by the dot and then the structure element.

struct_ var.struc_ele

10.2.3 Assigning values to individual members of the structure :

Individual structure members can also be assigned values as :

stud_db1 .roll = 101;

strcpy(stud_db1 .name, “Johnny”);

stud_db1 .per = 87.5

stud_db1 .marks1 = 80;

stud_db1 .marks2 = 95;

stud_db.marks2 represents the marks2 of stud_db1.

Thus we have given values to the members of structure variables stud_db1. We can also make use of **scant** to read values for structure variables through the keyboard. The following example will illustrate :

Example : To read values to members of a structure variable

```
main()

{
        struct person
        {
                char name[30];
                float sal;
        };
        struct person per_1;
        printf(“\nEnter values for name and salary :”);
        scanf(“%s%f”, &per_1 .name, &per_1 .sal);
        printf(“\nName = %s \tSalary = %.2f”, per_1 .name, per_1 .sal);
}
```

A sample run of the program :

```
Enter values for name and salary Joseph 3500.80
Name = Joseph            Salary = 3500.80
```

In this example, we have declared a structure person and a structure variable per_1. Through the keyboard we input the values of the structure variables of per_1 and then output them on the screen.

10.1 & 10.2 Check Your Progress.

1. Fill in the blanks:

a) The elements of a structure can be accessed using the .....operator.

b) The keyword **struct** declares the structure with a name which is called the .....

c) Declaration of a structure simply describes a..... of the structure.

d) The fields of a structure are called as structure .....

2. Write True or False :

a) Every structure member has to be declared separately for its type.

b) Structures have to initialised at the time of declaration.

c) There is no way to access structure members separately.

d) Structures can hold data of different types.

e) When a structure is declared a compiler immediately allocates memory for the structure.

f) The use of the tag name is optional in structure declaration.

3. Declare the following structures and intialise them with sample data :

a) Structure to hold the data of books in a library, which consist of the book code, book title, number of copies, number of copies issued.

.....

.....

b) Structure of a personal record which includes name, telephone number and email.

.....

.....

c) Structure of a record of hospital doctors which indicate their name, registration number, area of specialisation and visiting hours.

.....

.....

10.3 ARRAY OF STRUCTURES

In the above examples, we have declared a number of structure variables for a particular structure. But in actual practice when we are required to handle a large data for such structures, then we would have to declare numerous structure variables for the same structure, eg. if we wish to use the above structure for representing marks of 100 students then we will have to declare 100 structure variables of that type. A more convenient way of doing this is to declare an array of structures. Each element of the array will represent a structure variable. This can be done as illustrated in the following declaration :

```
struct stud student[100];
```

Here we have declared an array student of structure type stud. Each element of the array represents a structure variable. Each element will of the type stud. Thus taking the structure template of stud of our previous example we can declare an array of structure type stud as :

```
struct stud
{
```

```
        int roll;
        char name[30];
        int marks1 ;
        int marks2;
        float per;
};
struct stud stud_db1 [2];
```

Each element of the array will be a structure element of type stud. The elements of the array will be referenced using subscripts as :

stud_db1[0].name
stud_db1[1].marks1
stud_db1[0].marks2 and so on.

In an array of structures all the elements of the array are stored in contiguous memory locations. Each element of the array is a structure and structures are also stored in contiguous memory locations.

Example : Declaring an array of structures :

```
main()
{
        struct student
        {
                char name[30];
                float sal;
        };
        struct student stud_db[10];
        int i;
        for (i = 0; i < 10; i++)
        {
                printf("\nEnter values for name and salary :");
                scanf("%s%f", &stud_db[i].name, &stud_db[i].sal);
        }
        printf("\nName\tSalary\n");
        for (i = 0; i < 10; i++)
        {
                printf(" %s\t %.2f\n", stud_db[i].name, stud_db[i].sal);
        }
}
```

The above program declares an array stud_db of structure type student. Note the use of the subscript and dot operator to read the values of the structure members in the array. Run the program for sample values of data items.

Arrays within structures :

It is possible to use arrays within a structure. We have already seen this when we defined **char** array in the above examples. In a similar way we can use single or multidimensional arrays of type **int** or **float** within structures, eg. We may use a separate array for storing the marks obtained in various subjects by a student and then include this array in the structure declaration as shown below :

```
struct stud
{
        char name[25];
```

```
    int marks[3];
};
```

Here the structure contains two elements name and marks. name is an array of characters whereas marks is an array of type **int.** The individual elements should be accessed using appropriate subscripts eg. If stud_1 and stud_2 are structure variable of type stud then you can access the elements as :

```
stud_1.marks[2];
stud_2.marks[1] etc.
```

Also if you have defined an array student of type stud as

```
struct stud student[10];
```

then you can refer the structure elements as :.

```
student[2].marks[0];
student[3],marks[1] etc.
```

Example : To demonstrate use of arrays within an array of structure:

```
main()
{
    struct stud
    {
    char name[25]; int
    marks[5];
    };
    struct stud st[2];
    int i, j;
    printf("\nEnter name and marks\n");
    for(i = 0; i< 10; i ++)
    {
        scanf("%s", st[i].name[i]);
        for (j = 0; j < 5; j ++)
        {
            scanf("%d", &st[i].marks[j]);
    }
    printf(\^nName\tMarks1\tMarks2\tMarks3\tMarks4\tMarks5\t\n");
    for(i = 0; i< 10; i ++)
    {
        printf("%s", st[i].name[i]);
        for (j = 0; j < 5; j ++)
        {
            printf("\t%d", st[i].marks[i]);
        }
        printf("\n");
    }
}
```

Carefully follow the program and see how the structure members are accessed. The marks are input in an array of integers where the subscript j is used. The name is a string which uses the same subscript i as that of the structure array.

The sizeof operator: In order to determine the size of the structure we can use the sizeof operator as :

sizeof(struct struc1)

This operator will determine the number of bytes required to hold all the members of the **struct** struc1. eg.

if struc_1 is a variable of type **struct** then

sixed1(struc_1)

will give the size of the structure in bytes.

On the other hand if struc_1 is an array of structures then

sizeof(struc_1) would give the total number of bytes the array struc_1 requires.

Example : Determine the size of a structure variable :

```
main()
{
        struct stud
        {
                int roll;
                char name[20];
};
struct stud st, std[5];
int i j;
i = sizeof(st);
j = sizeof(std);
printf("\nSize of structure st = %d\nSize of array std of structure = %d");
}
```

The output of the program will be :

Size of structure st = 22

Size of array std of structure = 110

10.3 Check Your Progress.

1. **Declare the following structures as arrays and illustrate how their sizes can be determined using the sizeof operator.**

a) Structure to hold the data of books in a library, which consist of the book code, book title, number of copies, number of copies issued.

.....

.....

b) Structure of a record of hospital doctors which indicate his name, registration number, area of specialisation and visiting hours.

.....

.....

10.4 ASSIGNING VALUES OF ONE STRUCTURE VARIABLE TO ANOTHER

The values of a structure variable can be assigned to another structure variable of the same type using the **assignment operator (=).**

Example : The following example will illustrate assignment of values of one structure variable to another.

```
main()
{
        struct stud
        {
```

```
        char name[25];
        int marks1;
        int marks2;
        };
        struct stud stud_1 = {"John", 75, 82};
        struct stud stud_2, stud_3;
        stud_2 = stud_1;
        stud_3 = stud_1;
        printf("\nElements of stud_1 :%s\t%d\t%d",
                stud_1 .name, stud_1 .marks1, stud_1 .marks2);
        printf("\nElements of stud_2 :%s\t%d\t%d",
                stud_2.name, stud_2.marks1, stud_2.marks2);
        printf("\nElements of stud_3 :%s\t%d\t%d",
                stud_3.name, stud_3.marks1, stud_3.marks2);
}
```

The output of the program would be :

```
Elements of stud_1 John     75     82
Elements of stud_1 John     75     82
Elements of stud_1 John     75     82
```

Thus the elements of structure stud_1 are copied to stud_2 and stud_3,

An alternative method to copy structure elements is shown below :

```
strcpy(stud_2.name,stud_1.name);
stud_2.marks1 = stud_1 .marks1;
stud_2.marks2 = stud_1.marks2;
```

As you can see in this method we are copying individual structure members to the corresponding members of the other structure. The first method is much easier to use as compared to the second one. Use the second method and rewrite the above program to copy the structure members one by one.

10.4.1 Comparison of Structure variables :

We have seen that it is possible to assign the values of one structure variable to another. Similarly, structure variables of the same type can be compared for equality or inequality.

eg. If book1 and book2 are two structure variables of type book then the following operations are valid :

book1 == book2

compare all members of book1 and book2. It returns a value 1 if all members are equal else returns 0.

book1 != book2

will compare all members of book1 and book2. Returns a 1 if the members are not equal else returns a 0.

It may be noted that all compilers may not support this feature of comparison although indivisual members can be compared.

10.4 Check Your Progress.

1. What values will the following return for the structure variables declared as shown below :

a) i = lib1.sr_no == lib2.sr_no

.....

.....

b) j = lib1 .author != lib2.author;

.....

.....

c) k = ((lib3.author == lib4.author) && (lib1 .sr_no == lib2.sr_no));

.....

.....

d) i = lib3.copies == lib4.copies;

.....

.....

```
struct library
        {
                int sr_no;
                char author[30];
                int copies;
        };
        struct library lib1 = {2,"abc", 5}, lib2 = {2, "abc", 5};
        struct libr
        {
                int sr_no;
                char name[25];
                char author[25];
                int copies;
        };
        struct libr lib3 = {100, "abc", "Imn", 4}, lib4 = {100, "Imn", "abc", 4};
```

10.5 NESTING OF STRUCTURES

One structure can be nested in another structure. This means that you can have a structure within a structure. Let us see how to nest a structure within another structure with the example given below :

We have a structure emp which contains the **information about an employee like** his name, department, age, basic salary, dearness allowance and house rent allowance. A single structure to represent this information can be declared as :

```
struct emp
{
        char name[25];
        char dept[20];
        int age;
        float basic_sal;
        float da;
        float hra;
} emp_rec;
```

On the other hand we can declare the items related to the salary part in another

substructure as:

```
struc
{
        float basic_sal;
        float da;
        float hra;
} salary;
```

and then nest this substructure within the outer structure as :

```
struct emp
{
        char name[25];
        char dept[20];
        int age;
        struct
        {
                float basic_sal;
                float da;
                float hra;
} salary;
} emp_rec;
```

The emp structure contains a member salary which itself is a structure. The members of the inner structure variable salary can be accessed as :

```
emp_rec.salary.da
emp_rec.salary.basic_sal
emp_rec.salary. hra
```

Thus to access the innermost member in a nested structure all the structure variables must be chained from the outermost to innermost variables with the **dot operator.**

Structures can also be nested by making use of structure tags. The above declaration can also be done in the following manner:

```
struct sal
{
        float basic_sal;
        float da;
        float hra;
};
struct emp
{
        char name[25];
        char dept[20];
        int age;
        struct sal salary;
};
```

The template sal is defined outside the emp template. Then it is used to define the structure salary within the emp template.

The concept of structure nesting can be extended to nest more than one type of structure within a structure.

Example : To use structure nesting :

```
main()
{
        struct sal
        {
                float basic_sal;
                float da;
                float hra;
} ;
struct emp
{
                char name[25];
                char dept[20];
                int age;
                struct sal salary;
};
        struct emp emp_1;
        printf("Enter members of emp_1\n");
        printf("Enter name :");
        scanf("%s", emp_1.name);
        printf("\nEnter department:");
        scanf("%s", emp_1.dept);
        printf("\nEnter age:");
        scanf("%d", &emp_1.age);
        printf("\nEnter Basic Salary :");
        scanf("%f", &emp_1 .salary.basic_sal);
        printf("\nEnter Dearness Allowance :");
        scanf("%f", &emp_1 .salary.da):
        printf("\nAn Enter House Rent Allowance :");
        scanf("%f", &emp_1 .salary.hra);
        printf("\nAn Data for emp_1");
        printf("\nName\tDepartment\tAge\n");
        printf("%s\t%s\t%d\n", emp_1.name, emp_1.dept, emp_1.age);
        printf("\nBasicSal\tDearness\tHouse Rent\n");
        printf("%.2f\t%.2f\t%.2f\n", emp_1 .salary.basic_sal, emp_1 .salary.da, emp_1
        .salary.hra);
}
```

10.5 Check Your Progress.

1. Write C program for the following :

a) A database of a library where the outer structure will hold the code and name of the book and the inner structure will contain the fields viz. the name of the author, the number of copies, the number of copies issued and year and month of latest edition (as string). Make use of structure nesting. In the first part of the program declare the structure and define an array of structures to hold the data for 50 such books. Input a sample data of 4-5 books in this data base.

Then access records from this structure and print them in a properly formatted manner.

10.6 PASSING A STRUCTURE VARIABLE TO A FUNCTION

C supports passing of structure values as arguments to functions. The following methods are used to pass structure variables to a function :

- Passing each member of the structure as an actual argument when calling a function. In this method, as the number of structure elements goes on increasing, the method becomes unmanageable and also cumbersome to use.

Example : To pass individual elements of a structure to a function.

```
void (char*, int);
main()
{
        struct emp
        {
                char name[25];
                int age;
        };
        struct emp e1 = {“Jimmy”, 35};
        fn1(e1.name, e1.age);
}
fn1(char *ch, int a)
{
        printf(“\nName is : %s\nAge is : %d\n”, ch, a);
}
```

The output of the program :

```
Name is : Jimmy
Age is : 35
```

In this example, since name is an array of characters we have to pass the base address to the called function. On the other hand since age is of type **int** we pass its value. Thus this example illustrates use of call by value and call by reference both.

- In the second method, we pass a copy of the structure to the called function. We have already learnt that though the formal arguments may-change in the called function, it has no effect on the actual arguments of the calling function. Hence when the structure is passed in this manner, any changes to the structure members will not be effected to the members in the calling function. This means that we will have to return the entire structure back to the calling function. This method of passing an entire structure as an argument to the called function may not be supported by all compilers.

- Alternatively we can use pointers to pass the address of the structure to the called function. Using pointers proves to be much more efficient as compared to the above method. We shall study this method in the next section.

Let us now study how to pass a structure to the called function :

Passing a structure variable to a function :

The general form for passing a structure variable to the function is :

function_name(structure variable name)

It is important to note the following points when passing a structure variable to a function :

- The function must be declared for its type depending upon the data type it is supposed to return. If the function is supposed to return the entire structure then it has to be declared as type **struct** with the appropriate tag name.

- Both the structure variables used as the actual and formal arguments must be

of the same **struct** type.

- If the function is going to return a structure then the calling function should declare an identical type to accept the value returned.

Example : To pass a structure variable to a function

```
void fn1(struct emp);
struct emp
{
        char name[25];
        int age;
};
main()
{
        struct emp e1 = {“Jimmy”, 35};
        fn1(e1);
}
fn1 (struct emp emp_1)
{
printf(“\nName is : %s\nAge is : %d\n”, emp_1 .name, emp_1 .age);
}
```

This program will produce the same output as the previous example. Here we have declared the structure outside **main()** so that it becomes available to both **main()** and fn1(). In this program we have not made the called function return anything to the calling function. Let us write another example for the same.

Example : To demonstrate passing a structure variable to a function and making the function return a structure.

```
struct num
{

        float a;
        float b;
        float mul;
        float cliff;
        float sum;
};
struct num num1;
struct num fn();

main()
{
        printf(“\nEnter value for a :”);
        scanf(“%f”, &num1 .a);
        printf(“Enter value for b :”);
        scanf(“%f”, &num1 .b);
        num1 =fn(num1, num1.a, num1.b);
```

```
        printf("The sum is : %.2f", num1 .sum);
        printf("\nThe difference is : %.2f", num1.diff);
        printf("\nThe product is :%.2f", num1.mul);
        printf("\nThe quotient is :%.2f", num1.quo);
}
struct num fn(no, i,j)
struct num no;
float i;
float j;
{
        no.sum = i + j;
        no.diff = i - j;
        no.mul = i *j;
        no.quo = i/j;
        return(no);
}
```

In the above example we have used function fn() to calculate the sum, difference, product and quotient of two numbers a and b. The structure num1 stores the two numbers and all these four values.W send the structure and the values of a and b as parameters to the function fn(). The function calculates the values and returns the data of type **struct** num back to the calling function. These values are assigned to the structure variable num1 in the calling function. Note that the structure num is declared outside **main()** to make it globally accessible to all the functions.

10.6 Check Your Progress.

1. Write in about 5-6 sentences the various methods to pass structure variables to functions.

   .....
   .....
   .....
   .....

2. **Write true or false** :

a) It is possible to pass structure members as arguments to a function.

b) When a structure is passed as an argument to a function the function cannot return anything.

c) It is possible to pass the entire structure to a function and to make the function return a data of type **struct.**

10.7 POINTERS AND STRUCTURES

In the above example, we have used the **call by value** method to pass arguments. Another way to do the same is by using the **call by reference** method of functions. This method as you know makes use of pointers. Therefore before using the call by reference method with structures let us first study a few basic concepts about pointers and structures :

We can declare a pointer to data objects of type structure eg.

```
struct stud
{
        char name[25];
        int roll;
        float per;
} st, *ptr;
```

Here, st is a structure variable structure type stud and pointer ptr is a pointer to point to data object of structure type stud.

When we initialise the pointer as :

```
ptr = st;
```

it will assign the address of the first member of st to ptr. When using pointers we cannot make use of the (.) dot operator to access the elements of the structure variable, since the pointer is not a structure variable but a pointer to the structure. Therefore C provides the **arrow operator** '->' to refer to the structure elements. To access the elements of st using pointers we have to write it as follows :

```
ptr->name
ptr->roll
ptr->per
```

Thus we can modify the above program and pass the address of the structure to a function.

Example : This method will be best discussed with an example :

```
struct num
{
        int a;
        int b;
        float mul;
        float quo;
        int diff;
        int sum;
};
struct num num1;
fn1();

main()
{
        printf("\nEnter value for a :");
        scanf("%d", &num1 .a);
        printf("Enter value for b :");
        scanf("%d", &num1.b);
        fn1(&num1);
        printf("\nSum = %d, Prod = %.2f Diff = %d, Quotient = %.2f",
        num1.sum, num1.mul, num1.diff, num1.quo);
}
fn1 (struct num *no)
{
        no->mul = no->a * no->b;
        no->quo = no->a/no->b;
        no->sum = no->a + no->b;
        no->diff = no->a - no->b;
```

```
}
```

In this program we have been able to compute all the operations in one function. We make use of the arrow operator '->' to assign the values of the results to the corresponding members of the structure. Follow the program carefully.

Using pointers with array of structures : When we have declared an array of structures how do we use pointers? Let us see how this is done with the help of the following example :

```
struct stud
{
        char name[25];
        int roll;
        float per;
} st[5], *ptr;
```

Here we define st as an array of 5 elements each of type **struct** stud. When we assign the pointer ptr as :

```
ptr = st;
```

the address of the zeroeth element of st is assigned to ptr. Thus ptr Will now point to st[0]. Then we can access the members as :

```
ptr->name
ptr->roll
ptr->per
```

Here when the pointer ptr is incremented by one it will point to st[1] i.e. the next element of the array st[]. We can access the individual members of all the elements of the array st[5] as :

```
for(ptr = st; ptr < st + 5; ptr++)
        printf("%s\t%d\t%f\n", ptr->name, ptr->roll, ptr->per);
```

Alternatively we can also use the notation :

```
(*ptr).name
(*ptr).roll
(*ptr).per
```

to access the members. Note the syntax for accessing the members. Parenthesis are essential around *ptr because the . **operator** has a higher precedence then the **operator.**

10.7 Check Your Progress.

1. Fill in the blanks :

a) C provides the ..... to refer to the structure elements when using **pointers.**

b) * operator has a ..... precedence than the . (dot) operator.

c) We can pass the ..... of a structure to a function using pointers.

2. Write a C program for the following :

a) Declare an array of structures which holds the length and breadths of 10 rectangles. Pass this using pointers to a function which computes the areas and perimeters of these rectangles.

10.8 SUMMARY

Structure is a collection of data items of different data-types under single name for convenient handling.

Structures are derived data types.

Structures are defined at a global level.

The array of structures is the most convenient way of handling the large amount of data.

Nested structures means C allows us to define a structure within a structure.

Structures also can be passed to a function as a parameter and can also return from function.

When large structures are to be passed to function it is to pass pointers to structures.

10.9 CHECK YOUR PROGRESS - ANSWERS

10.1 & 10.2

1. a) dot
b) structure tag
c) template
d) members

2. a) True
b) False
c) False
d) True
e) False
f) True

3. a)
```
struct lib
{
        int code;
        char name[20];
        int copies;
         int issued;
};
        struct lib bk1 = {“101, “CProgramming”, 10, 7};
```

b)
```
struct per
{
        char name[20];
        long unsigned ph;
        char email[25];
};
struct per p1 = {“John”, 7908352, “mymail@yahoo.corn”};
```

c)
```
struct doc
{
        char name[20];
        int regno;
        char special;
        char hrs[20];
};
```

```
struct doc d1 = {“Dr.Stevens”, 1204, “Surgery”, “2.00to5.00”};
```

10.3

1 a)

```
struct lib
{
        int code; char
        name[20];
        int copies;
        int issued;
};
struct lib bk1 [10];
int i;
i = sizeof(bk1);
```

b)

```
struct doc
{
        char name[20];
        int regno;
        char special;
        char hrs[20];
};
struct doc d1 [50];
int i;
i = sizeof(d1);
```

10.4 1**.** a) 1

b) 1

c) 0

d) 1

10 5

1.

```
#include “stdarg.h”
#include “stdio.h”
main()
{
        struct bk
        {
                char author[25];
                int copies;
                int issued;
                char edition[20];
        };
        struct book
```

```
{
        int code;
        char name[30];
        struct bk b1;
};
struct book b[50];
int i;
printf("Enter book data :\n");
for(i = 0; i< 5; i++)
{
        printf("Enter code :");
        scanf("%d", &b[i].code);
        fflush(stdin);
        printf("Enter name:");
        scanf("%s", b[i].name);
        printf("Enter author:");
        scanf("%s", b[i].b1.author);
        printf("Enter copies :");
        scanf("%d", &b[i].b1 .copies);
        printf("Cppies issued:");
        scanf("%d", &b[i].b1 .issued);
        printf("Enter latest edition :");
        scanf("%s", b[i].b1 .edition);
}
printf("\nCode\tName\t\tAuthor\t\tCopies\tIssued\tLatest Edition\n");
for(i = 0; k 5; i++)
printf("%d\t%s\t%s\t%d\t%d\t%s\n", b[i].code, b[i].name,
b[i].b1.author,
b[i].b1 .copies, b[i].b1 .issued, b[i].b1 .edition);
      }
```

10.6

1. C supports passing of structure values as arguments to functions. The following methods are used to pass structure variables to a function:

(i) Passing each member of the structure as an actual argument when calling a function. In this method, as the number of structure elements goes on increasing, the method becomes unmanageable and also cumbersome to use.

(ii) Passing a copy of the structure to the called functioh.In this method however, when the structure is passed to a function, any changes to the structure members will not be effected to the members in the calling function. Therefore we will have to return the entire structure back to the calling function. This method of passing an entire structure as an argument to the called function may not be supported by all compilers.

(iii) In the third method, we can use pointers to pass the address of the structure to the called function. Using pointers proves to be much more efficient as compared to the above method.

2. a) True

b) False

c) True

10.7

1. a) arrow operator
b) lower
c) address

2.
```
struct rect
{
        float len;
        float breadth;
        float area;
        float peri;
};
main()
{
        struct rect r[10];
        int i;
        for(i= 0; i< 5; i++)
{
        printf("Enter length and breadth :");
        scanf("%f%f", &r[i].len, &r[i].breadth);
}
fn(&r);
}
fn(r1)
struct rect *r1;
{
        struct rect *ptr;
        ptr = r1 ;
        while(ptr < r1 +5)
{
        ptr->area = ptr->len * ptr->breadth;
        ptr->peri = 2 * (ptr->len + ptr->breadth);
        printf("Area = %.2f\tPerimeter = %.2f\n", ptr->area, ptr->peri);
        ptr++;
        }
}
```

10.10 QUESTIONS FOR SELF - STUDY

1. What are structures? Describe the terms structure tag, structure members. Explain with an example how to define a structure.
2. Why are arrays of structures used? Describe with example how to declare an array of structures.
3. Write short notes on passing a structure variable to a function.
4. Write a note on Pointers and Structures.

10.11 SUGGESTED READINGS

Let us C : Yashwant Kanitkar

C for Beginners : Madhusudan Mothe

CHAPTER 11

FILE MANIPULATION

11.0 OBJECTIVES

Friends, the study of this chapter will help you to

- state high level and low level File Input/Output Operations
- learn the purpose of opening of files viz. read, write, append etc.
- discuess how to open, close a file
- write programs to perform formatted high level file Input/Output using the standard library functions in C
- explian what is random file access and the functions available for random access
- Cover what are command line arguments and write programs for the same
- explain how to take care of errors during file input/output
- discuss what is the C preprocessor and learn the preprocessor directives viz. macro substitution and file inclusion.

11.1 INTRODUCTION

Uptil now we have studied console input and output functions for input and output and written a number of programs with the help of these functions. However, in real life applications data volumes to be handled are very large and console input/output does not become a convenient method. This is where we make use of files to store data on disksand read it from the disks whenever required. Thus a **file** is a place on a disk where related data is stored. C has a number of standard library functions to perform basic file operations.

These functions include :

- opening a file
- reading data from a file
- writing to a file
- closing a file
- naming a file.

Functions to perform input/output operations on files are broadly classified as :

- **Low level File I/O** functions also called as System Input/Output functions.

- **High Level File I/O** functions also called as standard or stream Input/Output functions. The standard I/O library of C has a number of functions to perform high level file I/O. High level I/O functions are much more easier to use than low level disk I/O. However low level disk I/O functions are much more efficient in terms of operation and amount of memory used by the program.

The high level disk input/output operations are further classified as text and binary. The basic difference between these two modes lies in the way in which a file is opened. In both these modes we have both the formatted and unformatted functions. Text and binary files handle the following areas in different ways :

- How new lines are handled
- How end of file is represented
- How numbers are stored
- Handling of new lines

In this chapter our primary focus will be on high level disk input/output operations in the text mode.

11.2 HIGH LEVEL INPUT/OUTPUT FUNCTIONS

Before we study the high level input/output functions on files in detail, let us first know a few things related to opening, closing and purpose of opening the file.

Opening a File :

Before reading from a file or writing to it the first thing to be accomplished is to open the file. Once a file is opened a link is established between the operating system and the program. The operating system has to know certain things about the file viz :

File name : File name is a string which makes a valid filename depending upon the operating system, eg. hello.c, abc.out etc.

Data Structure : The data structure of the file is defined as FILE in the standard I/O function definition. This structure has been defined in the header file stdio.h (standard input/output header file). This header file is always required to be included in our programs when we wish to perform operations on files. All files should therefore be declared of type FILE before they are used. FILE is a defined data type.

Purpose : When we open the file we have to mention the purpose of opening the file i.e whether we want to read a file, write to an already existing file, append new contents at the end of a file etc.

Declaring and Opening a File :

Every file which we open has its own FILE structure which contains information about the file like its size, its current location in memory etc. The FILE structure contains a character pointer which points to the first character that is about to be read.

The format for declaring and opening a file is :

```
FILE *fp;
fp = fopen("filename", "mode");
```

fp is declared to be a pointer to the data type FILE. fp contains the address of the structure FILE which has been defined in the standard I/O header file stdio.h. The second statement opens the file whose name is filename. Note that both the filename and mode are strings and therefore enclosed in double quotes. The **mode** indicates the **purpose** of opening the file. The mode can be one of the following :

"r"	searches for the file. If it exists, it is loaded in memory and the pointer is set to the first character in the file. If the file does not exist it returns NULL. *reading from file is possible.*
"w"	searches for the file. If it exists, its contents are overwritten. If the file does not exist, a new file is created. If the file cannot be opened returns NULL. *writing to the file can be done.*
"a"	searches a file. If it exists, it is loaded in memory and a pointer is set to point to the first character in the file. If it does not exist, a new file is created. If unable to open a file returns NULL. *appending new contents at the end of the file is possible*
"r+"	Searches for the file. If its exists it is loaded into memory and a pointer is set to point to the first character in the file. If the file does not exist returns NULL. *It is possible to read, write new contents, modify existing contents*
"w+"	Searches for file. If found its contents are destroyed. If the file is not found a new file is created. If unable to open file returns NULL. *writing new contents, reading them back, and modifying existing contents is possible*
"a+"	Searches for file. If it exists it gets loaded into memory and a pointer is set to point to the first character in the file. If it does not exist, a new file is created. Returns NULL if unable to open the file. *reading existing contents, appending new contents is possible. Cannot modify existing contents*

Multiple files can be opened and used at a given time. The exact number however is dependent on the system which we are using.

Closing a File :

When we have finished the operations on a file, the file must be closed. This ensures that all the outstanding information associated with the file is removed from the buffers and all links to the file are broken. There are a number of other reasons for which the file has to be closed. They include:

- Misuse of the file is prevented.

- We might also be required to close a file in order to open it in some other mode

- There is a limit to the number of files that can be kept open at a particular time.

In such cases, unwanted files may be closed.

The function to close a file is :

fclose(filepointer);

This function will close the file associated with the FILE Pointer *file pointer.* Closing a file deactivate the file and the file is no longer accessible. As soon as a file is closed, the file pointer associated with it may be used for another file.

11.1 & 11.2 Check Your Progress.

1. Answer in one sentence :

a) What is a file?

.....

.....

b) What is the first thing to be done when handling file input/output?

.....

.....

c) Give any one reason why a file should be closed?

.....

.....

d) What are the different purposes for which a file should be opened?

.....

.....

2. Describe the following file opening modes.

a) "r" -

.....

.....

b) "w" -

.....

.....

11.2.1 Unformatted High Level File Input Output (Text Mode):

fgetc and fputc functions :

The simplest file Input/Output functions from the standard I/O routines are **fgetc** and **fputc.** These functions can handle one character at a time.

If a file is opened in the "r" mode with the file pointer fp, then

fgetc(fp);

reads a character from the file whose pointer is fp.

If a file is opened in the "w" mode, with the file pointer fp, then

fputc(ch, fp);

will write the character contained in ch to the file associated with FILE pointer fp.

fgetc and **fputc** make the file pointer move ahead by one character for every operation. The reading of the file should be stopped when the EOF (end of file is encountered).

Let us write a program to open a file and display its contents on the monitor to illustrate the use of the **fgetc.**

Example : To read a file and display its contents

```
#include "stdarg.h"
#include "stdio.h"
main()
{
        FILE *fp;
        char i;
        fp = fopen("array1.c", "r");
        if(fp == NULL)
```

```
    {
        printf("Cannot open source file");
        exit();
    }
    i = fgetc(fp);
    while((i = fgetc(fp)) != (char)EOF)
        printf("%c", i);
    printf("%c", i);
    fclose(fp);
}
```

In the above example, if the file cannot be opened successfully print the message "Cannot open source file" is printed and the program is exited. If the file has been successfully opened the data in the file will be read character by character till end of file is encountered. Every time we read a character we display it on the screen. Once the entire file has been read, the file should be closed with **fclose.**

fgetc and **fputc** can be used together in order to copy contents of one file to another. The following example will illustrate the use of the **fgetc** and **fputc** to read characters from a file and write them to a new file.

Example : To read a file and copy its contents to another file

```
#include "stdarg.h"
#include "stdio.h"
main()
{
    FILE *fp, *fp1;
    char ch;
    fp = fopen("myfile", "r");
    fp1 = fopen("copyfile", "w")
    if(fp == NULL)
    {
        printf("Cannot open source file");
        exit();
    }
    if(fp1 == NULL)
    {
        printf("Cannot open target file");
        exit();
    }

    ch = fgetc(fp);
    while((ch = fgetc(fp)) != (char)EOF)
    {
        fputc(ch, fp1);
    }
    fclose(fp);
    fclose(fp1);
    printf("File copying successful!");
}
```

In this example, we have opened myfile in the read mode to copy its contents to the copyfile. Remember that copyfile is to be opened in the w mode. If the source

file (myfile in our case) is not found it returns NULL. Once the file has been successfully openend the contents of myfile are copied to copyfile character by character till EOF of myfile occurs. Remember that when copyfile is opened in w mode, if the file does not exist a new file is created to write, but if the file does exist its contents are overwritten.

The getw and putw functions :

getw and **putw** are similar to **fgetc** and **fputc.** They are used to read integer values. These functions are useful when you are dealing with only integer data. The general form of **putw** is :

putw(integer, fp);

to put an integer into the file and

getw(fp);

to read an integer from the file.

Example : To illustrate **getw** and **putw**

```
#include “stdarg.h”
#include “stdio.h”
main()
{
        int i;
        FILE *fp;
        fp = fopen(“Intfile”, “w”);
        if (fp == NULL)
        {
                printf(“Unable to open file”);
                exit(1);
}
printf(“Enter integer values to file (-1) to finish :\n”);
whi!e(i!=-1)
{       scanf(“%d”, &i);
                putw(i.fp);
        }
        fclose(fp);
        fp = fopen(“Intfile”,”r”);
        while((i = getw(fp)) != EOF)
                printf(“%d\t”,i);
        fclose(fp);
}
```

The program reads in integer values till you enter -1 to indicate end of data entry and prints them again on the screen by retrieving them from the Intfile using getw.

String I/O in Files :

In this section, let us study the functions that are capable of handling strings. The functions to read and write strings from and to a file are **fgets()** and **fputs().**

Let us write a program to write strings to a file by making use of the function**fputs()** and to display them on the screen by opening the file and using **fgets().**

Example : To write strings to a file and read them back

```
#include “stdarg.h”
#include “stdio.h”
main()
```

```
{
        FILE *fp;
        char str1[80];
        fp = fopen("Sample.txt", "w");
        if (fp == NULL)
        {
                printf("\nUnable to open file");
            exit();
        }
        printf("\nEnter strings for file :\n");
        while(strlen(gets(str1)) >0)
        {
            fputs(str1, fp);
            fputs("\n", fp);
        }
        fclose(fp);
        printf("\nLet us write strings back from the file :\n");
        fp = fopen("Sample.txt", "r");
        while(fgets(str1,79,fp) != NULL)
            printf("%s",str1);
        fclose(fp);
}
```

A sample output:

```
Enter strings for file :
Mary had a little lamb
Its fleece was white as snow
Let us write strings back from the file :
Mary had a little lamb
Its fleece was white as snow
```

There are a number of things to note in this program. str1 is defined as an array of characters i.e. a string of size 80 (width of the screen). The function **fputs()** writes the contents of str1 to the file pointed to by fp. The file is opened in the "w" mode to write. Once again note that "w" will create a new file if it does not exist and will overwrite the file

if it does exist. **fputs()** does not automatically insert a new line character at the end of the line. Therefore, the second **fputs()** is used to enter a newline character to the file after every string input. When inputting strings, each string input is terminated by pressing the Enter key. In order to terminate entering strings press Enter as the first character on a new line. This implies that the string is of zero length and the condition strlen(gets(str1) >0) will become false. The file is then closed.

The second part of the program now opens the file in the read mode to read its contents and display them. For this purpose we make use of the **fgets()** function. **fgets()** takes three arguments, the first is the address where the string is stored, second the maximum length of the string and third is the pointer to the structure FILE. A NULL will be returned by **fgets()** when all the lines have been read and the program will end.

11.2.1 Check Your Progress.

1. Write True or false :

a) fgetc(fp) is used to write a character to a file whose file pointer is fp.

b) We make use of fputc() function for inputting strings to a file.

c) To read from a file it has to be opened in the "r" mode.

2. Explain the following functions :

a) fgets():

.....

.....

b) fputc():

.....

.....

c) fputs():

.....

.....

3. Write a program to write characters to a file using fputc(). Terminate input by typing Z'. Read the characters back from the file and print them on screen.

11.2.2 Formatted Disk I/O Functions :

The two functions for formatted disk I/O to write characters, strings, integers, floats are the **fscanf()** and the **fprintf(). fscanf()** and **fprintf()** are identical to **printf** and **scanf** except that they work on files.

The general form of the **fprintf()** is :

fprintf(fp, "control string", list);

Here fp is the file pointer associated with the file to which we are writing. The control string contains the output specifications for the items in the list as in the case of **printf.** The list can include variables, constants and strings.

The general form of **fscanf()** is

fscanf(fp, "control string", list);

where fp is the file pointer associated to the file from which we are reading. The control string contains the specifications for reading the items from the list.

Let us see how to make use of the formatted disk I/O functions with the help of the following program: :

Example : To use **fscanf()** and **fprintf()** for formatted file Input/Output

```
#include "stdarg.h"
#include "stdio.h"
main()
{
        FILE *fp;
        char name[20], email[20];
        long unsigned ph;
        char more;

        fp = fopen("Person", "w")
        if (fp== NULL)
        {
                printf("Cannot open file");
```

```
                exit();
        }
        more = 'y';
        while (more == 'y')
        {
                printf("Enter name, email, phone:\n");
                scanf("%s %s %lu", name, email, &ph)
                fprintf(fp, "%s\n%s\n%lu\n", name, email, ph);
                printf("\nEnter more data :");
                fflush(stdin);
                scanf("%c", &more);
                }
        fclose(fp);

        fp = fopen("Person", "r");
        while(fscanf(fp, "%s %s %lu", name, email, &ph) != EOF)
                printf("%s %s %lu\n", name, email, ph);
        fclose(fp);
}
```

In this program we have made use of the **fflush()** function. The **fflush()** function removes any data remaining in the buffer. The argument that **fflush()** takes is the buffer which we want to flush out. Here our buffer is the **stdin** which is the buffer related to the standard input device which in this case is the keyboard.

The **fflush()** is required in this program, because you prompt the user with the statement "Enter more data :" But the user has typed the Enter key after completing his entries of the previous name, email and ph and the program takes the Enter key as an answer to Enter more data ? and hence will stop reading further. To avoid this we first empty the buffer and then ask the user whether he wishes to enter more data.

This program has been used to write and read dissimilar data types to a file. Therefore we can make efficient use of structures while reading and writing such records to a file. This program illustrates how to make use of structures for formatted file I/O.

Example :

```
#include "stdarg.h"
#include "stdio.h"
main()
{
        FILE *fp;
        struct per
        {
                char name[20];
                char email[20];
                long unsigned ph;
        };
        struct per person;
        char more = 'Y';
        fp = fopen("Person.dat", "w");
        if(fp == NULL)
```

```
        {
                printf("\nCannot open file");
                exit();
        }
        while(more == 'Y')
                {
                printf("\nEnter name, email and phone :\n");
                scanf("%s %s %lu", person.name, person.email, &person.ph);
                fprintf(fp, "%s\n%s\n%lu\n", person.name, person.email, person.ph);
                printf("\nEnter another record (Y/N) ?");
                fflush(stdin);
                more = getche();
        }
        printf("\n");
        fclose(fp);
        fp = fopen("person.dat", "r");
        if(fp == NULL)
        {
                printf("\nCannot open file\n");
                exit();
        }
                while(fscanf(fp, "%s %s %lu", person.name, person.email, person.ph)
        != EOF)
        printf("%s %s %lu\n", person.name, person.email, person.ph);
        fclose(fp);
}
```

11.2.2 Check Your Progress.

1. **Write a program** using the formatted file input/output functions to input data to an inventory file. The data consists of the following elements : the itemcode, the item name, unit price and quantity. Make use of a structure to store the data. Enter data for 10 such items to the file and print out the same.

11.3 RANDOM FILE ACCESS

In our above discussion we have seen the various methods by which we can access data sequentially for reading and writing. However, in practical usage there are numerous situations when we are interested in accessing a particular part of a file and not the other parts. The standard C library provides functions for such random access. These functions are **fseek, ftell** and **rewind.**

ftell: This function takes a file pointer as its argument. It returns a long integer value which corresponds to the current position in the file. This function is useful to save current position of the file for later use. The **ftell** takes the following form :

n = *ftell(fp);*

where n is a long integer, n gives the relative offset from the current position (in bytes) which implies that n bytes have been read (or written) so far.

rewind : rewind takes the file pointer as its argument and resets the position

to the start of the file. eg. rewind(fp);

will set the file position to the beginning of the file.The first byte in the file is numbered 0, the second 1 and so on. We can use **rewind** to read or write to file again, without having to close and reopen it.

fseek is a function which is used to move the file position to the required location. The form of **fseek** is :

fseek(filepointer, offset, position);

where filepointer is a pointer to the file, offset is the value of type **long** and the position is an integer. The offset specifies the number of bytes to be moved from the location specified in position. Position can take one of the following three values :

Position	Value
0	Beginning of file
1	Current position
2	End of file

If the offset is positive, the position is moved forward, if it is negative, the position is moved backward. eg.

fseek(fp, m, 1);	will move the position forward by m bytes from current position
fseek(fp, -m, 2);	will move the position backward by m bytes from end of file
fseekfp, m, 0)	will move forward by m bytes starting from beginning of file.

If the operation is successful **fseek** returns a zero. In the event that we attempt to read beyond the limits of the file, **fseek** returns a value -1 and an error occurs. Always be sure to check for errors when using **fseek.**

Let us create a file in a sample program to make use of these functions:

Example :

```
#include “stdarg.h”
#include “stdio.h”
main()
{
int ch;
        FILE *fp;

        fp = fopen(“charfile”, “w”);
        printf(“\nEnter characters for file “);
        while ((ch = getche()) != ‘Z’)
        {
                fputc(ch, fp);
                printf(“%c”, ch);
        }
        fclose(fp);
        fp = fopen(charfile”, “r”);
        fseek(fp,2,0);
        printf(“\nPosition of character : %ld\tCharacter is : %c”, ftell(fp), fgetc(fp));
        fseek(fp, -6, 2);
        printf(“\nPosition of character : %ld\tCharacter is : %c”, ftell(fp), fgetc(fp));
        printf(“\nCurrent position %ld”, ftell(fp)); rewind(fp);
```

```
        printf("\nThe first position is %ld\tCharacter is :&c", ftell(fp), fgetc(fp));
}
```

The program uses the random file access function to determine the character at various positions by going forward, rewinding etc. Follow the program carefully and try to seek more characters using the random file access functions.

11.3 Check Your Progress.

1. Explain the following functions :

a) ftell:

.....

.....

b) rewind :

.....

.....

2. What will be the result of the following ?

a) fseek(fp, 8, 0):

.....

.....

b) fseek(fp, -16, 2):

.....

.....

c) fseek(fp, 10, 1) :

.....

.....

11.4 COMMAND LINE ARGUMENTS

A command line argument is a parameter which is supplied to a program when the particular program is invoked eg. it may be a filename of a file which is to be processed by that command.

Let us understand command line arguments by using the example of creating a file to copy the source file to the target file. So if you want to copy a file named source to a file named target then we may use a command like:

C>flcopy source target

where flcopy is the program which is the executable file (An executable file is one which has the .exe extension and can be executed as the DOS prompt).

How do we send the source and target filenames as parameters to flcopy? It is possible for us to pass the source filename and the target filename to **main()** to make

these parameters available to the program. Up till now, we have been using **main()** in all our programs and this is the place where our program execution starts. However, we have not yet passed any parameters to **main().** Actually **main()** can take two parameters **argc** and **argv.** Information contained in the command line is passed to the program through these two arguments **argc** and **argv** whenever **main()** is invoked.

argc is an **int** which counts the number of arguments on the command line. **argv** is an array of pointers to strings(which are the command line arguments). Thus **argc** is an integer whose value is equal to the number of strings to which **argv** points. When the program is executed, the strings on the command line are passed to **main(). argv** is an array called the **argument vector.** It is an array of character pointers that

points to the command line arguments. Thus in our example

flcopy source target

the value of **argc** is 3 (which is the number of arguments on the command line) and the array of character pointers to strings **(argv)** is :

argv[0] contains the base address of the string “flcopy”
argv[1] contains the base address of the string “source”
argv[2] contains the base address of the string “target”

When the command line arguments are to be passed to **main(),** we have to declare the **main()** function as follows :

```
main(argc, argv)
int argc;
char *argv[];
or
main(int argc, char *argv[]);
```

Note the order of arguments in passing them to **main(). It** is is **argc** and then **argv.** The first parameter in the command line is always the program name. This implies that argv[0] is always the name of the program.

Having understood this, let us make use of the command line arguments to write a program to copy the source file to the target file.

Example : Use command line arguments to copy source to target. #include “stdarg.h”

```
#include “stdio.h”
main(int argc, char *argv[])
{
        int i;
        FILE *fs, *ft;
        if(argc !=3)
        {
                printf(“Incorrect arguments”);
                exit();
        }
        fs = fopen(argv[1], “r”);
        if (fs == NULL)
        {
                printf(“Unable to open source file”);
                exit();
        }
        ft = fopen(argv[2], “w”);
        if (ft == NULL)
        {
                printf(“Unabkle to open target file”);
                exit();
        }
        while (i = fgetc(fs)) != EOF)
                fputc(i,ft);
        fclose(fs);
        fclose(ft);
}
```

Here we have used the same logic of copying source to target as in our previous program. Save this file with the name flcopy.c. After successful compilation the exe file flcopy.exe will be generated. This file can now be executed at the command prompt.

Now you can run the flcopy file at the command prompt as :

c> flcopy myfile yourfile

This program will copy the contents of myfile to yourfile. You have now successfully created a program that will copy the source file to the target file at the DOS prompt.

Advantages of using argc and argv :

The advantages of using **argc** and **argv** are :

- we can execute this program at the command prompt, there is no need to compile the program every time we want to run it.

- In our previous program we had to either specify the filename of source and target in the program itself or prompt the user the every time to enter the filenames during execution.

11.4 Check Your Progress

1. Answer the following in 1-2 lines

a) What is meant by argc and argv?

.....

.....

b) What are the advantages of using argc and argv?

.....

.....

c) What is a command line argument?

.....

.....

11.5 SUMMARY

A file is a permanent place on a disk where related data is stored. C provides number of functions for opening a file , reading data from a file, writing to a file, closing a file.

File can be bifurcated on the basis of the way it is opened i. e Binary mode or text mode.

Random file- Random files read the records Randomly. One can modify, Update, delete files directly without accessing all the records.

Arguments can also be passed to function main, using special parameters. Since we pass the arguments to main at the command prompt they are called command line parameters.

11.6 CHECK YOUR PROGRESS - ANSWERS

11.1 & 11.2

1. a) A file is a place on a disk where related data is stored.

b) The first thing to be done while handling input/output to files is to open the file.

c) We might be required to open the file in some other mode and therefore the file should be closed.

d) The different purposes for which a file should be opened are reading a file, writing to an already existing file, appending new contents at the end of a file.

2. a) r - This mode searches for the file. If it exists, it is loaded in memory and the pointer is set to the first character in the file. If the file does not exist it returns **NULL. It is** possible to read from the file.

b) w - This mode searches for the file. If it exists, its contents are over written. If the file does not exist, a new file is created. If the file cannot be opened returns **NULL. It** is possible to write to the file.

11.2.1

1. a) False

b) False

c) True

2. **a)** fgets() is a standard C library function to read strings from a file**.** It takes three arguments, the first is the address where the string is stored, second the maximum length of the string and third is the pointer to the structure **FILE.**

b) fgetc is a standard input/output function capable of reading one character at a time. These functions handle one character at a time. When you open a file in the "r" mode fgetc(fp); reads a character from the file whose pointer is fp. fgetc makes the file pointer move ahead by one character for every operation.

c) The function fputs() writes the contents of a string t6 the file pointed to by fp. The file is opened in the "w" mode to write. fputs() function does not automatically insert a new line character at the end of the line.

3.
```
#include "stdarg.h"
#include "stdio.h"
main()
{
        FILE *fp;
        char ch1;
        int i;
        fp = fopen("newfile", "w");
        printf("\nEnter contents for file :");
        'while((ch1 = getche()) != 'Z')
                fputc(ch1, fp);
        fputc(ch1, fp);
        printf("\n");
        fclose(fp);
        fp = fopen("newfile", "r");
        if(fp == NULL)
        {
                printf("\nCannot open file");
                exit();
        }
        while((ch1 = fgetc(fp)) != 'Z')
                putchar(ch1);
}
```

11.2.2

1.

```
#include “stdarg.h”
#include “stdio.h”;
main()
{
        struct inventory
        {
                int cd;
                char name[20];
                float unit;
                 int qty;
        };
        struct inventory in;
        FILE *fp;
        int i;
        fp = fopen(“Invent.dat”, “w”);
        if(fp == NULL)
        {
                printf(“Unable to open file”);
                exit();
        }
        printf(“\nEnter item code, name, unit price, qty\n”);
        for(i = 0; i < 2; i++)
        {
                scanf(“%d %s %f %d”, &in.cd, in.name, &in.unit, &in.qty);
                fprintf(fp, “%d\n%s\n%f\n%d\n”, in.cd, in.name, in.unit, in.qty);
        }
        fclose(fp);
        fp = fopen(“Invent.dat”, “r”);
        if(fp == NULL)
        {
                printf(“Cannot open file”);
                exit();
        }
        printf(“Item Code Name Unit Price Quantity\n”);
        while(fscanf(fp, “%d %s %f %d”, &in.cd, in.name, &in.unit, &in.qty) !=
        EOF)
                printf(“%d %s %f %d\n”, in.cd, in.name, in.unit, in.qty); fclose(fp);
}
```

11.3

1. a) **ftell**: This function is useful to save the current position of the file This function, takes a file pointer as its argument and returns a long integer value corresponding to the current position in the file, ftell takes the following form :

n = ftell(fp);

where n is a long integer, n gives the relative offset from the current position (in . bytes) which implies that n bytes have been read (or written) so far.

b) **rewind** : rewind takes the file pointer as its argument and resets the position to the start of the file. eg. rewind(fp);

will set the file position to the beginning of the file.The first byte in the file is numbered 0, the second 1 and so on. We can use rewind to read or write to file again, without having to close and reopen it.

2. a) Will move the position forward by 8 bytes starting from the beginning of the file.

b) Will move the position backward by 16 bytes from the end of the file.

c) Will move the position forward by 10 bytes starting from the current position.

11.4

1. a) The function **main()** can take two parameters **argc** and **argv.** These two arguments are used to pass the information contained in the command line whenever **main()** is invoked. **argc** is an **int** which counts the number of arguments on the command line, **argv** is an array of pointers to strings(which are the command line arguments). Thus **argc** is an integer whose value is equal to the number of strings to which **argv** points, **argv** is an array called the **argument vector.** It is an array of character pointers that points to the command line arguments.

b) The advantages of using **argc** and **argv** are :

- we can execute a program at the command prompt, there is no need to compile the program every time we want to run it.

-It is not necessary to either specially the filenames (parameters) every time in the program and prompt the user to input them during execution. They can be passed directly at the command prompt.

c) A command line argument is a parameter which is supplied to a program when the particular program is invoked eg. it may be a filename of a file which is to be processed by that command**.**

11.7 QUESTIONS FOR SELF - STUDY

1. Describe the various file opening modes.
2. What happens when a file is closed? What are the reasons for closing a file?
3. Describe the unformatted high level file I/O functions?
4. Which are the functions provided in the standard C library for random file access?
5. Describe argc & argv with example.

11.8 SUGGESTED READINGS

Lets us C : Yashwant Kanitkar

The C Programming Language : Kernighan & ritche