IT-06/14/19

Space for learners notes

Gauhati University

Institute of Distance and Open Learning

IT- 02

INTRODUCTION TO PROGRAMMING

Contents:

Contributors:

Mrs. Pallabi Saikia Assistant Professor of Computer Science GUIDOL	Unit- 1
Mrs. Mazida Akhtara Ahmed Guest Teacher, Dept. of IT GUIDOL	Unit- 2
Dr. Ridip Dev Choudhury Assistant Professor of Computer Science GUIDOL	Unit- 3
Dr. Swapnanil Gogoi Assistant Professor of Computer Science GUIDOL	Units- 4 & 7
Dr. Khurshid Alam Borbora Assistant Professor of Computer Science GUIDOL	Unit- 5
Dr. Pranab Das Assistant Professor, ADU	Unit- 6
Mr. Hemanta Kalita Assistant Professor of Computer Science GUIDOL	Unit- 8

Course Coordination:

Prof. Amit Choudhury	Director, IDOL, Gauhati University
Dr. Anjana Kakoti Mahanta	HOD, Dept. of Computer Science, G.U.

Content Editor:

Prof. Kandarpa Kr. Sarma	HOD, Dept. Electronics & Communication Engineering, GUIST
Prof. Anjana Kakoti Mahanta	HOD, Dept. of Computer Science, G.U.
Dr. Abhijit Sarma	Associate Professor, Dept. of Computer Science, G.U.

Language Editor:

Dalim Chandra Das	Asst. Professor, Dept. of English, GUIDOL

Format Editor:

Dipankar Saikia	Editor, SLM, IDOL, Gauhati University

Cover Page Designing:

Bhaskar Jyoti Goswami	IDOL, Gauhati University

November, 2018
Re-Print : July, 2019

SYLLABUS

IT-02: Introduction to Programming

Space for learners notes

Unit 1: Introduction to C

Steps for problem solving, algorithm, analysis of algorithm efficiency, flowchart, pseudo code, program, programming languages, translators.

Unit 2: History of C, Variables Constants, Expressions and Operators in C

History of C, features of C, structure of a C program, writing a C program, compiling and run a C program, syntax and semantic errors, logical and runtime errors, execution process.

Variables and Constants: Character set, identifiers and keywords, rules for forming identifiers, data types and storage classes in C, variables, declaring variables, initializing variables, constants, types of constants. Expressions and Operators: Assignment statements, unary and binary operators, arithmetic operators, relational operators, logical operators, comma and conditional operators, type cast operator, size of operator, precedence of operators.

Unit 3: Control Statements, Decision Control Statements

The If statement, the switch statement. Loop control statements: The while loop, the do-while loop, the for loop, the nested loop, the goto statement, the break statement, the continue statement.

Unit 4: Arrays and Strings

Arrays: Definition, syntax of array declaration and initialization, subscript, processing the arrays, multi-dimensional arrays, declaration and initialization of two-dimensional arrays, processing of two-dimensional arrays, representation of matrix using two-dimensional arrays.

Strings: Character arrays, declaration and initialization of strings, array of strings. Library string functions: strlen, strcpy, strncpy, strcmp, strncmp, strcmpi, strnicmp, strcat, strncat, strlwr, strupr, strrev, strdup, strchr, strset, strnset, strstr.

Unit 5: Functions

Definition, structure of a function, function declaration, function definition, formal parameter, actual parameter, the return statement, function prototypes, recursive function. Function calling: Call by value, call by address.

Space for learners notes

Unit 6: Structures and Unions

Declaration and initialization of structures, accessing the members of a structure, structures as function arguments, structures and arrays, unions, initializing an union, accessing the member of an union.

Unit 7: Pointers

What is pointer, address and indirection operators, pointer type declaration and assignment, pointer to a pointer, null pointer assignment, pointer arithmetic, passing pointers to functions, arrays and pointers, arrays of pointers, pointers and-strings.

Unit 8: C Preprocessors and Command Line Arguments and Files

The C Preprocessor and Command Line Arguments: Definition, macros in C, #define, #include, #ifdef, other preprocessor commands, predefined names defined by preprocessor, command line arguments in C, structure of programs that use command line arguments, accessing command line arguments.

Files: Definition, file handling in C using file pointers, fopen(), fclose(), input and output using file pointers, character input and output in files, string input/output functions, formatted input/output functions, block input/output functions, sequential files, random access files, positioning the file pointer.

CONTENTS:

Space for learners notes

Space for learners notes

Space for learners notes

Space for learners notes

Space for learners notes

UNIT 1 INTRODUCTION TO PROGRAMMING

Space for learners notes

CONTENTS

1.0 INTRODUCTION

Computer is an electronic device which accepts inputs, processes them and then finally produces the output. Computer consists of two main parts – hardware and software. Hardware includes its tangible, physical devices and the set of instructions required to manage the hardware and to accomplish different tasks of a computer is called the software. Software is nothing but a set of programs. In other words, software is also known as computer program. The person who performs the task of programming is called a programmer and the process of developing a program is called programming. With the help of a computer many real world problems like complex arithmetic equations, engineering calculations etc can be solved. In order to solve a problem, it should be fed with the appropriate instructions to get the result. To accomplish different tasks a programmer needs to write various programs. Before writing a program, the programmer should understand what the problem is and how it is to be solved. The programmer should have clear idea about the logic of the program. For clear understanding one should know about the basic tools for developing a program which includes algorithm, flowchart and pseudocodes. In this unit you will

Space for learners notes

find brief introduction to different programming languages, and learn how to develop algorithms, flowcharts, preudocodes which will help a programmer to write programmes in futute. Moreover you will learn about the different translators required while writing programs in different programming language.

1.1 OBJECTIVES

After going through this unit student will able to:

- Learn how to solve a problem
- Understand what is an algorithm
- Understand about the efficiency of the algorithm
- Understand how to develop an algorithm
- Understand what is a flowchart and how to draw it
- Learn what is a pseudocode and how to write a pseudocode
- Learn about programme and about different programming languages
- Appreciate the importance of translators

1.2 PROBLEM SOLVING

A computer is used to solve complex problems. The technique used to solve a problem depends on the type of the problem to be solved and a single problem can be solved in various ways depending upon the type of approach used by the programmer. Any problem can generally be solved by following the steps given below:

- First we need to understand the problem
- Secondly we should plan a strategy to solve the problem
- After planning the strategy we should try to find out all the possible alternatives.
- Out of all the possible alternatives we should try to find the optimal one and finally implement the plan
- After implementation, the result should be verified for correctness.

1.3 ALGORITHM

An algorithm is a basic technique to solve a given problem in a finite number of steps. The word algorithm derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 AD to 850 AD. Algorithm is quite helpful because as a beginner we may not be able to find a solution to a problem ready. In fact, it gives us an idea as what approach should be used for solving a particular program. After that we can use the same approach while programming in a computer.

An algorithm must have the following properties:

- **Input:** An algorithm can have zero or many number of inputs.
- **Output:** An algorithm should definitely have an output.
- **Finiteness:** An algorithm must terminate after a finite number of steps and each step should take finite amount of time.
- **Definiteness:** The steps must be stated clearly and correctly, and there should be no ambiguity.
- **Effectiveness:** The steps of an algorithm must be simple enough to carry out using just a pen and a paper.

Let us go through some examples:

Example: An algorithm to print the line **"Welcome to IDOL"**

```
Step1: Start
Step 2: Print "Welcome to IDOL"
Step 3: End
```

Example: An algorithm for finding the sum of two numbers.

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read two values in two variables num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum = num1+num2
Step 5: Display sum
Step 6: Stop
```

Example: An algorithm to find the largest among three different numbers

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read three values in the variables a,b and c.
Step 4: If a>b
            If a>c
                    Display a is the largest number.
            Else
                    Display c is the largest number.
```

Space for learners notes

Space for learners notes

```
    Else
        If b>c
            Display b is the largest number.
    Else
        Display c is the greatest number.
Step 5: Stop
```

Example: An algorithm to find the sum of first 50 numbers

```
Step 1: Start
Step 2: Declare variable n, sum
Step 3: Initialize n=1, sum=0
Step 4: sum = sum+n
Step 5: n = n+1
Step 6: Repeat steps 4 and 5 until n>50
Step 7: Print sum
Step 8: Stop
```

Example: An algorithm to interchange the value of two numbers

```
Step 1: Start
Step 2: Read two values in two variables a and b
Step 3: Declare third variable c
Step 4: c = a
    a = b
    b = c
Step 5: Print a and b
Step 6: Stop
```

STOP TO CONSIDER: Here a variable c is created to just store temporarily the values so that the content of the other two variables may get swapped

Example: An algorithm to find the area and perimeter of a square

```
Step 1: Start
Step 2: Read value for the variable length
Step 3: Declare variable area and perimeter
Step 4: area = length x length
Step 5: perimeter = 4 x length
Step 6: Print area and perimeter
Step 7: Stop
```

Example: An algorithm to find the square roots of a quadratic equation $ax^2 + bx + c = 0$

```
Step 1: Start
Step 2: Declare variables a,b,c, r1,r2,d
Step 3: Read the coefficients a,b,c
Step 4: Calculate $d = b^2 - 4ac$
Step 5: If d < 0, print "roots are imaginary" and go to step 7
    Else
            r1 = (-b + sqrt(d))/ 2 x a
            r2 = (-b + sqrt(d))/ 2 x a
Step 6: Print roots r1& r2
Step 7: End
```

Example: An algorithm to convert temperature from Fahrenheith to Celcius

```
Step 1: Start
Step 2: Declare variables F and C
Step 3: Read variable F
Step 4: Compute C = 5/9 x (F-32)
Step 5: Print C
Step 6: End
```

Space for learners notes

Space for learners notes

STOP TO CONSIDER: Here in step 4 we have used the defined formula of converting Fahrenheit to Celsius

Example: An algorithm to find the product of n numbers

```
Step 1: Start
Step 2: Read n
Step 3: Declare variables product, n, count.
Step 4: Initialize count=0, product=1,
Step 5: count = count+1
Step 6: product = count x product
Step 7: Repeat steps 4 & 5 until count>n
Step 8: Print product
Step 9: End
```

Example: An algorithm to find the factorial of a number.

```
Step 1: Start
Step 2: Declare variables n, fact, i
Step 3: Read the value of n
Step 4: Initialize fact=1, i=1
Step 5: Repeat steps 6 & 7 until i<=number
        Else go to step 9
Step 6: fact = fact x i
Step 7: i=i+1
Step 8: Print fact
Step 9: End
```

1.3.1 Analysis of Algorithm Efficiency

As we know that a problem can be solved in a number of ways using algorithms, the important thing is not just getting a solution to a problem but how to solve the problem efficiently. The efficiency of an algorithm depends upon a number of factors like speed of the processor, programming language, compiler as well as also the size of the input.

Space for learners notes

Apart from these factors, how the data is being organized and also which algorithm to choose in solving a particular problem has a huge impact on the efficiency of the program. For this before solving a problem we should study the behavior of the algorithm so that the correct algorithm is used to solve a particular problem. The efficiency of an algorithm determines how effectively it uses its resources. By resources it means time and space. Time refers to the time required by the algorithm to execute each step and space refers to the amount of memory utilized while executing the algorithm. Lesser the use of resources in algorithm, greater is its efficiency.

Time efficiency and space efficiency are also referred to as time complexity and space complexity respectively. In space efficiency the amount of memory used is analyzed whereas in time efficiency the running time of the algorithm is analyzed.

Space Efficiency: Space efficiency depends upon a number of components such as instruction space, data space and runtime stack space.

Time Efficiency: The running time of an algorithm depends upon factors like the type of compiler, speed of the computer, amount of data and the actual data. While computing time efficiency of an algorithm three cases are being considered:

- Worst case
- Average case
- Best case

Worst case: Suppose we are given a list of numbers and we need to find a particular number in that list using a particular algorithm. The worst case occurs if the number present in the list is in the last position or if it is not in that list. This case results in maximum number of operations as we need to traverse each and every element one by one. So it takes a lot of time and hence slow.

Average case: Considering the same example of the worst case, let us assume that the number of given elements is 20. While finding for a particular element in the list we got it in the 10th position then we can just say that the algorithm took average time.

Best case: If while searching for a particular element in the list, we got it in the first position itself then the algorithm is considered to be in the best case. Less number of operation in less amount of time.

STOP TO CONSIDER:

The analysis of algorithm is a very large topic that is covered in a different subject which is beyond the scope of this course. The above explanation is just as an introduction.

CHECK YOUR PROGRESS :

1. What is an algorithm?
2. Write an algorithm to find whether a number is odd or even.

Space for learners notes

1.4 FLOWCHART

Flowchart is a diagrammatic representation of an algorithm. There may be different kinds of flowcharts but in case of programming, program flowcharts are used. Flowchart represents the flow of data within the control of the program. Flowcharts are sometimes also termed as Process Flowchart, Process Map and Functional Flowchart. It helps us in understanding the programming approach to solve a problem. It is the step next to algorithm to elucidate the concept of programming. Flowcharts can be easily drawn with a pen and a paper. Flowcharts can also be drawn using different computer software. Although there are different types of flowchart, we will be discussing only the program flowchart as it is relevant to this course. Program flowchart is used for developing the structure of a program that shows the logical flow and the operations performed. A program flowchart includes inputs, operations, computation and calculations, decision making conditions, looping and branching sequences and various outputs.

HISTORY

Frank Gilbreth explained the first flow chart to the members of ASME in 1921 to introduce his presentation "Process Charts - First Steps in Finding the One Best Way". It soon made into the field of industrial engineering. During the early 1930s, Allan H. Mogensen (industrial engineer) trained business people to use some of the industrial engineering tools at the Work Simplification Conference in New York. A graduate, Art Spinanger, developed their Deliberate Methods Change Program. Another graduate, Ben S. Graham, used flowcharting to process information with the use of multi-flow process chart to exhibit many documents and their connection with others. In this way flowcharts were used first by engineers for designing different components. Because it was easy to develop and maintain a flowchart, it came into use in many fields. It became a very important tool in the field of computer science as it is the basic tool for developing programs.

Advantages of flowchart:

- It depicts the sequence of a program. It can be broken down further for study and analysis.
- It is easier to explain the logic of a program to a beginner using a flowchart.
- Flowchart does not follow any programming syntax, so anyone can develop it.
- Flowchart can be developed in parts. This concept can be used while developing software, as many programmers are involved, each one can develop his/her flowchart of the module assigned.
- Developing a flowchart before writing a program results in error-free program and takes less amount of time.

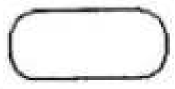
Disadvantages of flowchart:

- It is difficult to solve a complex problem using flowchart as the process becomes cumbersome and the user has to spend a lot of time developing it.

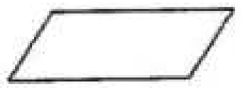
- If modifications are required the user has to erase the earlier flowchart and do it again.

Space for learners notes

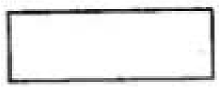
Some standard symbols are used while designing a flowchart. Flowchart symbols have been standardized by the American Standard Institute. The symbols along with the terms are given below:



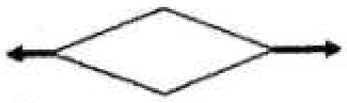
Start or End of the program symbol. This symbol is used in the beginning of the flowchart to indicate the start of the program.



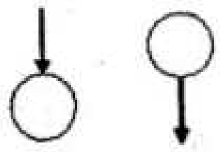
Input/Output symbol: This symbol is used to read the values of the inputs or variables that are required to perform a particular task. The same symbol is used also for output which gives the final result after carrying out all the operations.



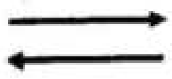
Process symbol: This symbol is used to carry out the operations after which the final result is given.



Decision making and branching symbol: This symbol is used to decide which path to follow based on a certain condition. For example, if the condition is true then path1 is selected and if not, path2.



This symbol is used for joining two parts of a flowchart when it has a large and complex structure this symbol is rarely used generally.



Connector or Flow line: This symbol is used to connect the symbols to show the flow of the program.

The rules to be followed for drawing flowcharts:

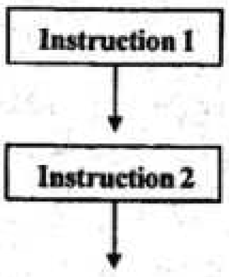
1. Flow lines are used to connect all the symbols of the flowchart.
2. Flow lines enter the top of the symbol and exit out at the bottom, except for the Decision symbol, which can have flow lines exiting from the bottom or the sides.
3. The flow of a program in a flowchart is always from top to bottom and never bottom to top.
4. Flowchart should always begin with the start symbol and end with the end symbol. In both the cases same symbol is used.

1.4.1 Basic Control Structure

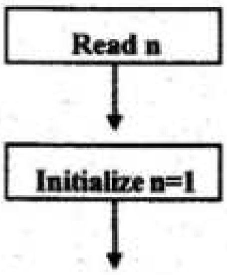
There are some standard ways of connecting the symbols in flowcharts. This forms the three basic control structures. Each of the three basic structures has a single entry flow and a single exit flow.

Space for learners notes

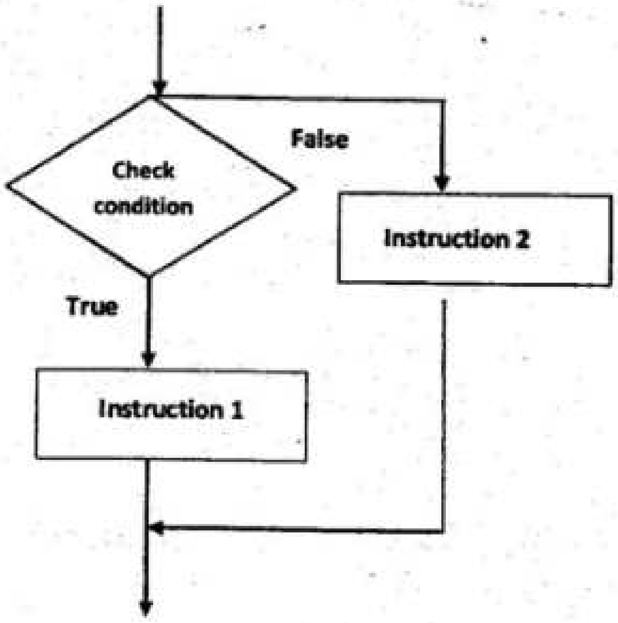
a) Sequence- In sequence control structure the steps are executed one after another and are represented by symbols that follow each other top to bottom or left to right. Top to bottom is the standard.



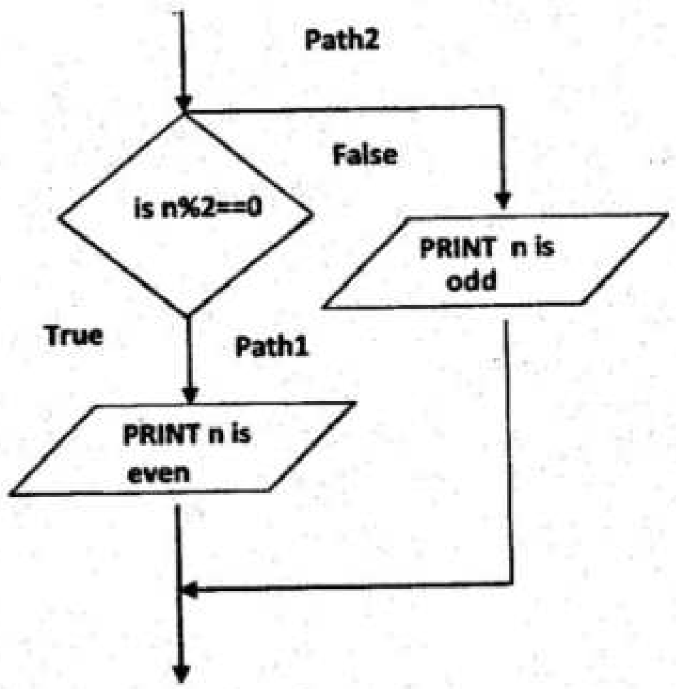
Suppose we are required to read a value and to initialize that value. These two statements should follow one another sequentially. In such a situation, we use sequence control structure.



b) Selection- In selection control structure, a condition is evaluated and depending upon whether the condition is true or false, the control may follow a certain path.

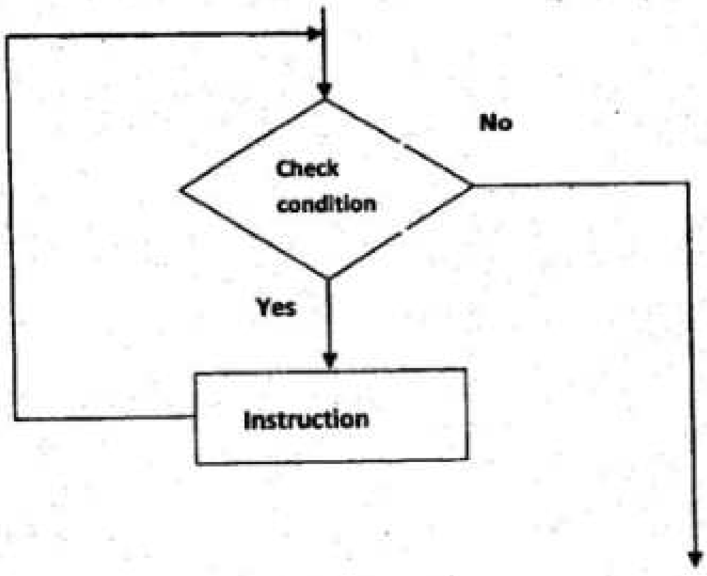


For example we have to find whether the given number is odd or even. There may be two possible cases: True or False. If it is true then path1 should be selected and if not, path 2.



STOP TO CONSIDER: In the above example, % operator is used. % operator returns the remainder.

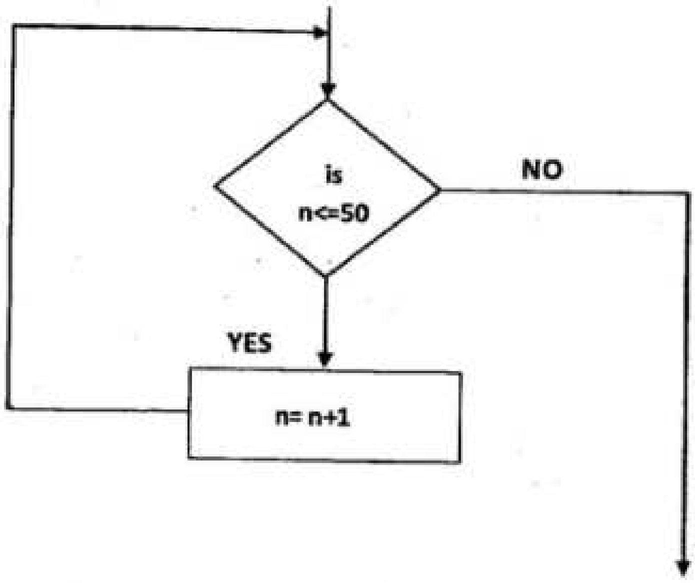
c) **Repetition**- In repetition control structure, a group of statements are executed repeatedly until it keeps on satisfying a given condition.



Space for learners notes

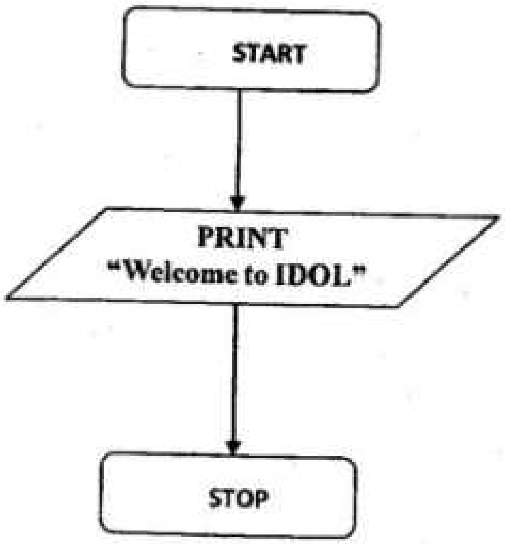
Space for learners notes

For example, we need to execute a particular statement until a number is less than or equal to 50. Until the condition is true the same statement is repeated, after which statement following the decision symbol is repeatedly executed.



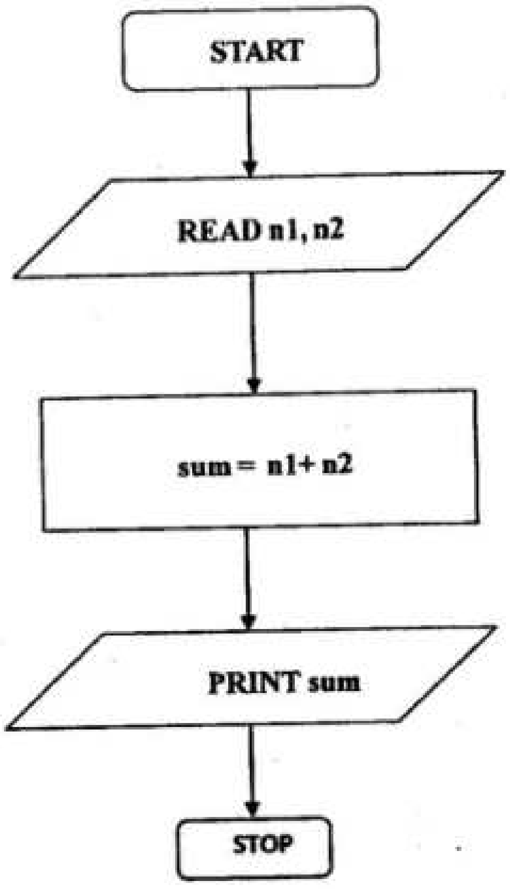
Now we will go through certain examples of flowchart having different control structures.

Example: Flowchart to print "Welcome to IDOL"



This is a simple example to print the given line. Here no variables are used as they are not required.

Example: Flowchart to find the sum of two numbers.

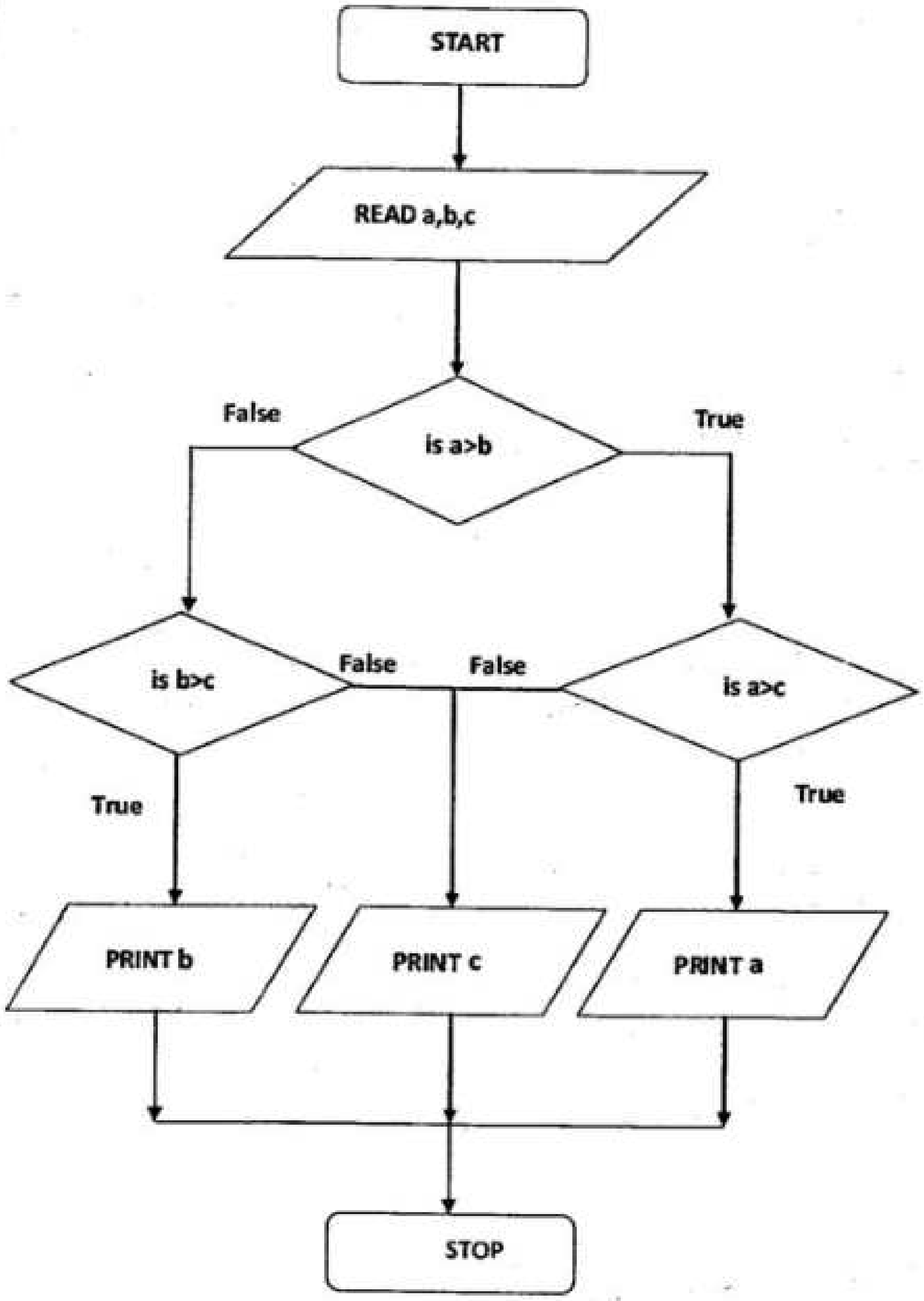


Here n1 and n2 are two variables where the two numbers to be added will be stored. The final result is stored in the variable sum.

Space for learners notes

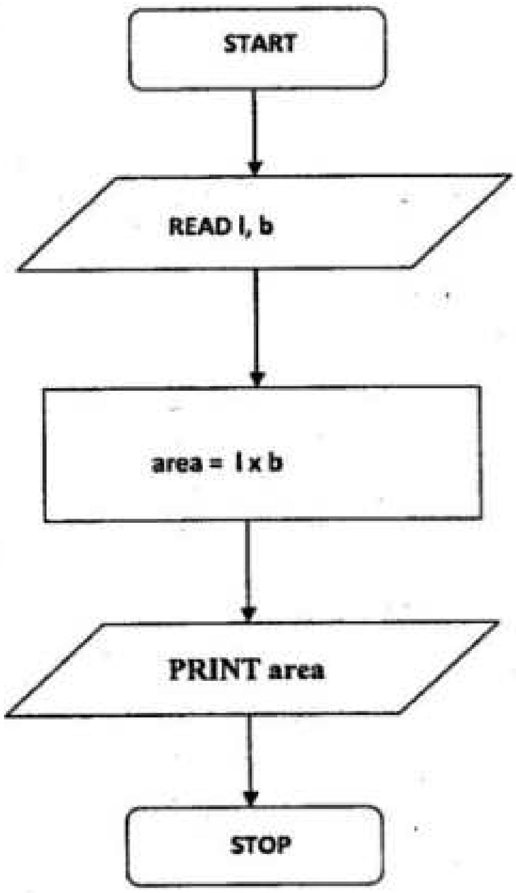
Space for learners notes

Example: Flowchart to find the largest of three numbers



Here a,b and c are the three varibles where the three numbers to be compared are stored. Three decision symbols are used for three comparisons.According to the outcome of the comparison, the control follow one of the all possible paths.

Example: Draw a flowchart to find the area of a rectangle

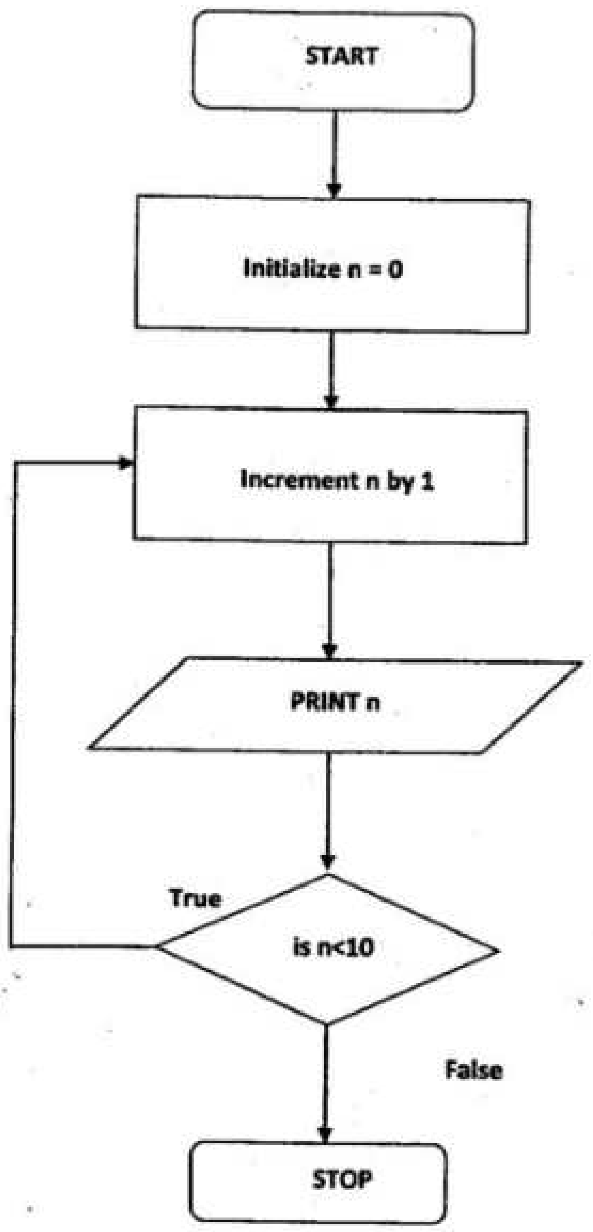


Here l and b are two variables for reading the value of length and breadth respectively. The third variable area is used to store the area of the rectangle which is nothing but the product of length and breadth.

Space for learners notes

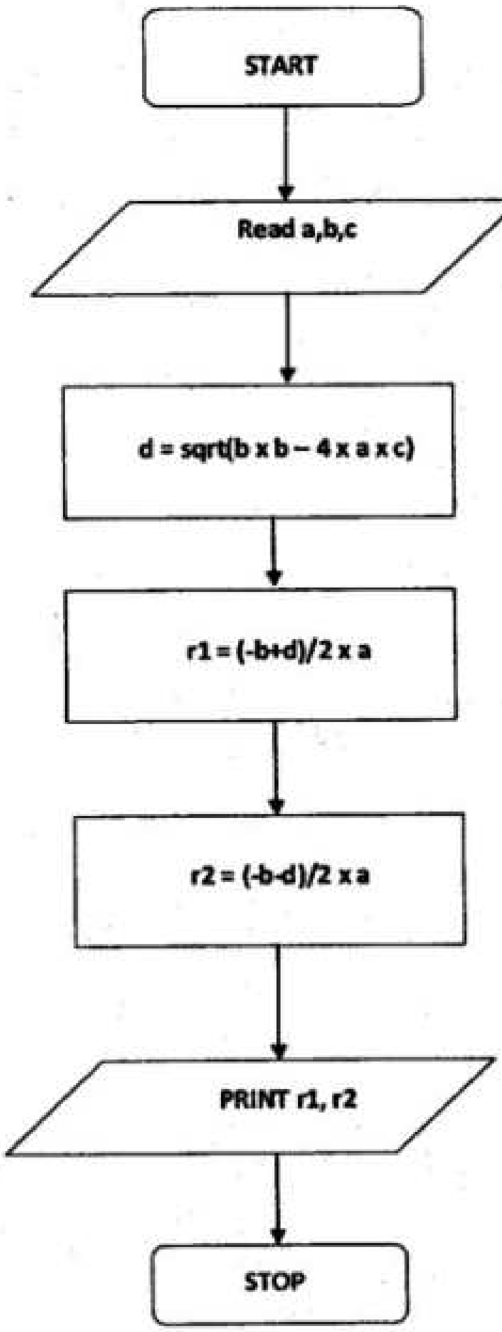
Space for learners notes

Example: Draw a flowchart to print the numbers from 1 to 10



Here n is a variable that is first initialized as 0. The value of n is then incremented by 1 and after this the value of n is printed. According to the question the values 1 to 10 need to be printed; so a loop is being used to check whether the value of n is less than or equal to n. The program prints n till it keeps on satisfying the given condition.

Example: Draw a flowchart to find the square roots of the quadratic equati $ax^2 + bx + c = 0$



Here the variables a, b and c are used for storing the coefficients of the given equation. As we know the roots of a quadratic equation is given by

$$x = \frac{b \pm \sqrt{b^2 - 4ac}}{2a}$$

The $\sqrt{b^2 - 4ac}$ part is assigned in the variable d. Thus the two roots of the equation are

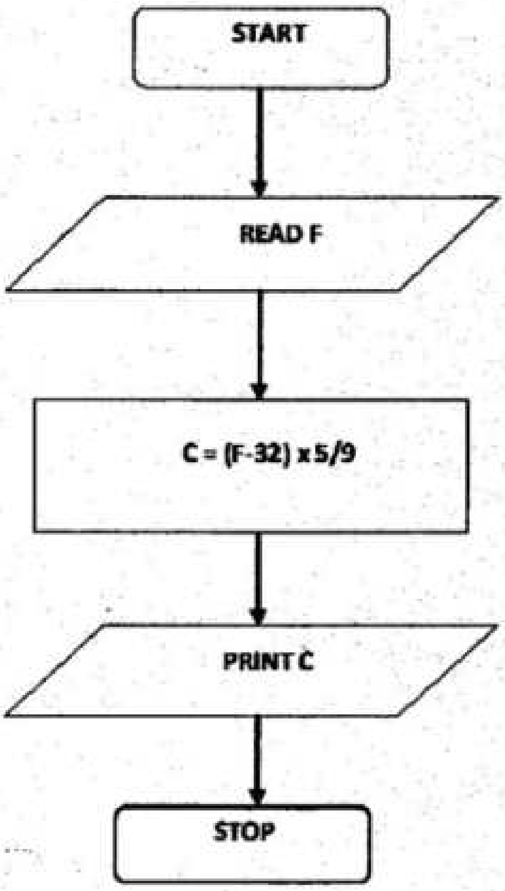
Space for learners notes

Space for learners notes

r1 = -(-b+d)/2 x a

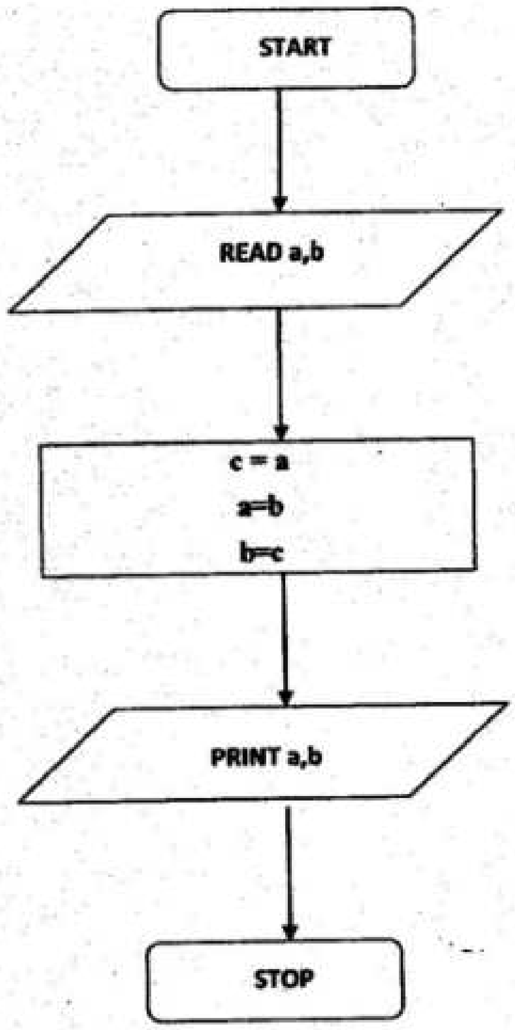
r2 = (-b-d)/2 x a

Example: Draw a flowchart to calculate temperature from Fahrenheit to Celsius



Here the variable F is used to read the temperature given in Fahrenheit. The variable C is used to convert the temperature from Fahrenheit to Celsius using the defined formula.

Example: Draw a flowchart to interchange the value of two numbers.

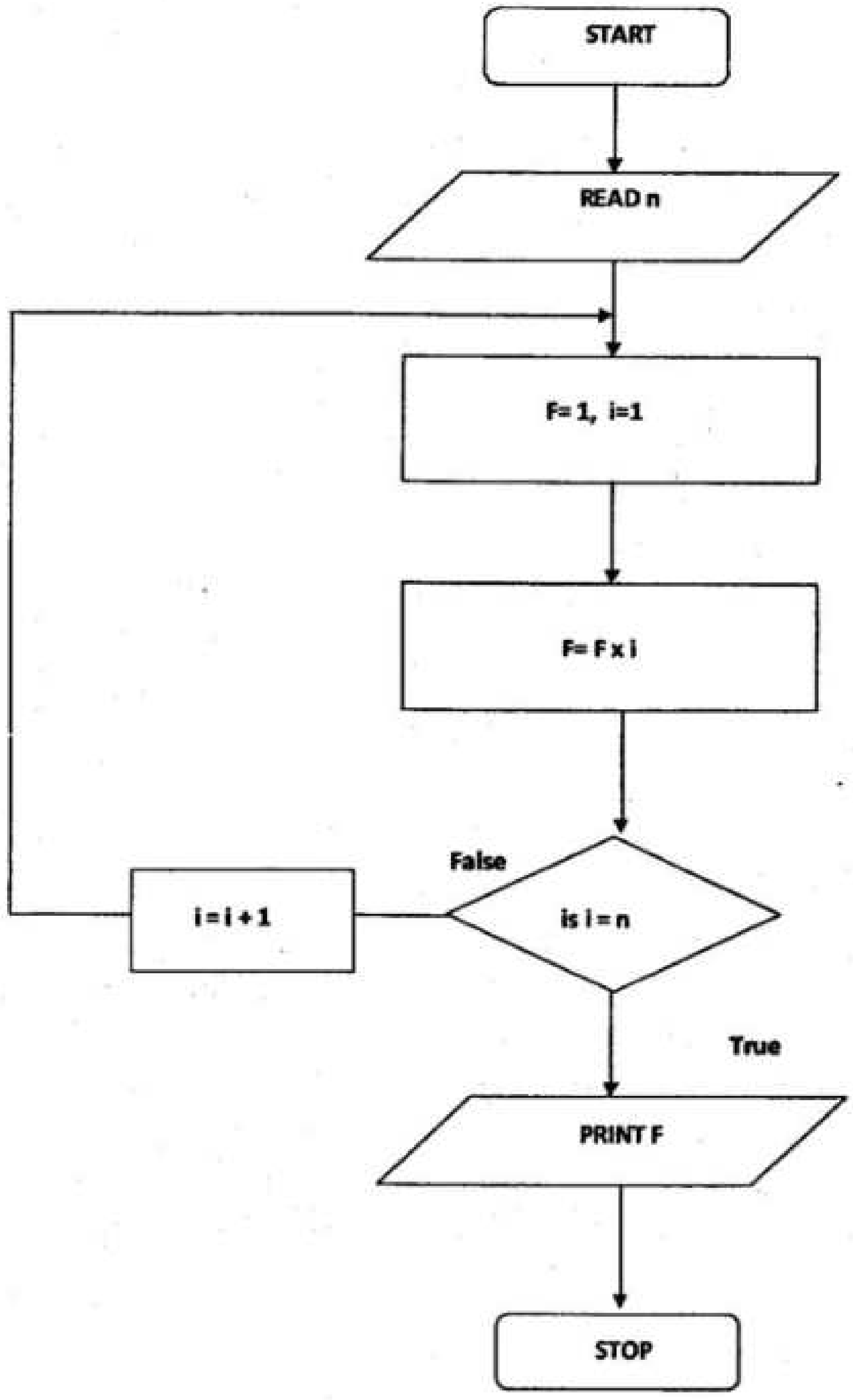


Here a and b are two variables used to store the values that are to be interchanged or swapped. The variable c is used to temporarily store the value of a, so that the value of b can be assigned to a, and the value of c- which is nothing but the value of a – is assigned to b.

Space for learners notes

Space for learners notes

Example: Draw a flowchart to find the factorial of a number.



Here n is used to store the number whose factorial is to be found out. The variable F is initialized as 1 and is used to store the factorial of the number. The variable i is used to increment the count of the number. The decision symbol is used to check whether the variable i after every increment becomes the number itself or not. Till the condition is met the control remains in a loop. When the value of i equals the value of n the control exits the loop and finally prints the result.

Space for learners notes

Self Assessment Question

- Draw a flowchart to find the average of five numbers.
- Draw a flowchart to find whether a number is positive or negative.

1.5 PSEUDOCODE

Pseudocode is an informal way of describing the operating principle of a program without any strict programming syntax or rules. It helps the programmer in developing algorithms. It is usually written before writing the final program and is in a form that can be easily and quickly converted into a program. As no prior technical knowledge of programming is required, anyone can write a pseudocode with very little effort. It uses short terms of the English alphabets and has input step for data, mathematical expression for manipulation and output steps for getting the final result.

Advantage

- Easy to write and understand.
- Can be easily modified.
- Can be easily converted to a programming language.

Disadvantage

- Doesn't have a proper structure, so the programmer may not understand the logic of the program.
- Flowcharts are Much more convenient than pseudocode to develop programs.

Let us go through some examples:

Example: Pseudocode to find the product of two numbers

```
Start
Read n1, n2
Compute product as n1x n2
Write product
End
```

Space for learners notes

Example: Pseudocode to find the area of a circle

```
Start
 Read radius
Compute area as 3.14 x radius x radius
 Write area
 End
```

Example: Pseudocode to find the perimeter of a rectangle

```
Start
Read length, breadth
Compute perimeter as 2x(length + breadth)
Write perimeter
End
```

Example: Pseudocode to print all multiples of 3 between 1 and 100

```
Start
Set n to 1
While(n<=33)
Compute n = n x 3
Write n
end While
```

Example: Pseudocode to find the sum of first 10 natural numbers

```
Start
Set n to 1 and sum= 0
While(n<=10)
Compute sum=sum +n
Compute n=n+1
endWhile
Write sum
```

Example: Pseudocode to display numbers 1 to 100

```
Start
For i = 1 to 100
Write i
end For
```

Space for learners notes

STOP TO CONSIDER: In the last three examples we have used loop statements while and for. We use them when a statement or a group of statements need to be executed repeatedly until a given condition is satisfied

Self Assessment Questions

- Write the pseudocode for finding the average of five numbers
- Write the pseudocode for finding all the multiples of 2 between 1 and 30

1.6 PROGRAM

A program is an ordered set of instructions given to a computer to perform a particular task. The person who writes a program is called a programmer and the program is written in a programming language. In order to get the results as desired by the programmer, it should be properly written with correct logic statements. The characteristics of a good program are:

- User friendly: It should be written in such a way that any reader can easily understand the logic of the program.
- Portable: It should be platform independent so that a program written in one machine can be easily shifted to another machine.
- Efficiency: It should use its resources efficiently. It should be fast to produce the output with the memory assigned to it in limited time.
- Reliable: It should be reliable enough to handle any type of error and should be able to display error message or warning.
- Documentation: It should use proper names for the variables and constants used in the program. It should have comment section wherever necessary so that one can understand the functionality of the program easily. It is one of the most important component of a program.

1.7 PROGRAMMING LANGUAGES

A programming language is a formal language which is comprised of a set of vocabulary and rules to instruct a computer to accomplish a certain task. Just as we human beings need a language to communicate, computers too require a language through which

Space for learners notes

they can communicate with the programmer so that it might solve the task assigned to it. As computer hardware have developed a lot from the 1st generation computers, in due course of time, computer languages, too, have come off a long way right from machine oriented language (that used strings of only 0 and 1) to problem oriented language. Programming language can broadly be classified into three categories:

1. Machine Language (First Generation Language)
2. Assembly Language (Second Generation Language)
3. High Level Language (Third Generation Language)

1. Machine Language: A computer understands only binary language which consists of two numbers viz. 0 and 1. The programming language that uses a sequence of 0 and 1 for writing instructions are known as machine language. Machine language was the first generation language. An instruction consists of two parts: op-code and operand. Op-code is the operation to be performed and operand is the address of the data where the op-code is to be applied. In case of machine language both op-code and operand will consist of a string of zero and one. An example of machine language is represented below

OP Code	Operand
001	010001110

One can easily imagine how tough it would be to write a program in machine language. One has to remember the op-code for each operation and also has to keep a list of addresses of the data to be operated on. This is simply cumbersome, time consuming and error-prone. Moreover the number of operands depends on the computer, so a program written for one computer is not applicable to another computer. Also correcting errors in such type of program is a mammoth task.

2. Assembly Language: Assembly language is the second generation language which is an upgrade of first generation language. Instructions written in assembly language used symbolic codes called mnemonics for the operation code and a string of characters for the operand. An example of assembly language program is represented below:

Operation	Operation address
READ	M
ADD	L

Space for learners notes

Programs written in assembly language is converted to machine codes by a special program called assembler. Assembly language is comparatively easier to remember than machine language but not at all convenient as it is machine dependent. The codes written for one computer cannot be implemented in another.

3. High Level Language: The third generation language is the high level language. Due to lack of portability in earlier languages, high level languages were developed to relieve the programmer of the low level details of the hardware part. High level language uses English phrases which are easier to learn and understand. A program written in high level language is machine independent and hence a program written in one computer can be easily shifted to another without or with little modification. Programs written in high level language is converted to machine language by a program known as compiler or interpreter. The features like portability, user friendliness, relative ease of maintenance have made high level languages very popular and convenient for performing any task. Examples of high level language are- FORTRAN, COBOL, BASIC, PASCAL,C,C++,JAVA etc.

In this course we are going to learn the C language and after completion of this course you will be easily able to write a program in C.

STOP TO CONSIDER:

First generation, second generation and third generation language is also called 1GL,2GL and 3GL respectively. Besides this, there is also fourth generation (4GL) and fifth generation language(5GL)

CHECK YOUR PROGRESS

3. What is a programming language?
4. Give two examples of high level language.
5. What is the difference between compiler and assembler ?

1.8 TRANSLATORS

Computer understands only binary language. So a program written in high level language needs to be converted into machine codes for its execution. A translator is a term that is used in programming to refer to a compiler, interpreter, assembler or anything that converts a high level language to another equivalent high level language or to a low level language. It is a programming processor that helps a programmer to convert the program written in any high level language called the source code to low level language called the object code without losing the semantics of the original code. There are various type of translators depending upon the functions performed by them.

Space for learners notes

1. **Compiler:** A compiler translates a program written in high level language into machine language. The compiler takes the whole program at a time and produces the equivalent machine codes. If errors are present in the program the programmers need to check the source code to correct it. It is recompiled to get the final executable file. Once it gets compiled you can execute the program any number of times without a compiler.
2. **Interpreter:** An interpreter also translates a program written in high level language into machine language but it executes one statement at a time and is therefore, slower than a compiler. As the interpreter checks a program line by line, it stops whenever an error is encountered. Here no executable file is produced. So a program needs to be interpreted every time it needs to be executed.
3. **Assembler:** An assembler is used to translate a program written in assembly language program to machine language. Similar to a compiler, an assembler produces an executable file and hence a program once assembled need not be re-assembled.

1.9 SUMMING UP

In this unit you have learned about problem solving and different approaches used in problem solving. You learned about algorithm which is a step by step procedure to solve a problem, and also about the efficiency of an algorithm which depends upon the two factors: time and space. Similarly, you have gained an understanding of flowchart which is a diagrammatic representation of an algorithm, and how it makes use of some symbols to represent the flow of data in a problem. There are three basic structures of a flowchart: sequence, selection and repetition. After this you have learned about pseudocode. Pseudocode is an informal language which helps the programmer in developing an algorithm. You have also learned the definition of a program and its characteristics. You have also learned about the requirement of a programming language and its different types. Lastly you have come to know that to convert a high level language into low level language we need translators. There are different types of translators such as compiler, interpreter and assembler.

1.10 KEY TERMS

- **Algorithm**: An algorithm is a basic technique to solve a given problem in a finite number of steps.
- **Flowchart**: Flowchart is a diagrammatic representation of an algorithm
- **Pseudocode**: Pseudocode is an informal way of describing the operating principle of a program without any strict programming syntax or rules
- **Program**: A program is a set of instructions given to a computer to perform a particular task.

- **Translator**:A translator is a term that is used in programming to refer a compiler, interpreter, assembler or anything that converts a high level language to another equivalent high level language or to a low level language.

Space for learners notes

1.11 ANSWERS TO CHECK YOUR PROGRESS

1. An algorithm is a basic technique to solve a given problem in a finite number of steps.

2. Algorithm to find whether a number is odd or even

```
Step 1: Start
Step 2: Declare variable n
Step 3: Read a value in variable n
Step 4: If n divisible by 2
        Go to step 5,else step 6
Step 5: Print "Even" and stop
Step 6: Print "Odd" and stop
```

3. A programming language is a formal language which comprises a set of vocabulary and rules to instruct a computer to accomplish a certain task. Just as we human beings need a language to communicate, computers too require a language through which they can communicate with the programmer so that it might solve the task assigned to it.

4. Two examples of high level language: C and C++

5. A compiler translates a program written in high level language to low level language whereas an assembler translates a program written in assembly language to low level language.

1.12 QUESTIONS AND EXERCISES

Multiple Choice Questions

1) The technique to solve a given problem in a finite number of steps is called:

a)Flowchart	b) Pseudocode
c)Algorithm	d) None of the above

2) The efficiency of an algorithm depends upon two factors:

a) Time and space	b) Time and money
c) Space and symbols	d) Space and technique

3) C is a/an

a) High level language b) Low level language

Space for learners notes

c) Assembly language d) Machine language

4) The characteristics of a good program are:

a) User friendly b) Reliable

c) Portable d) All of the above

5) An example of translator is:

a) Compiler b) Debugger

c) Text editor d) Operating system

Answers: 1(c), 2(a), 3.(a),4.(d),5.(a)

State True or False:

1) In selection control structure, a condition is evaluated and depending upon whether the condition is true or false, the control may follow a certain path.
2) The flow of a program in a flowchart is always from bottom to top.
3) The efficiency of an algorithm determines how effectively it uses its resources.
4) An algorithm can have infinite number of steps.
5) Assembler is not a translator.

Answers: 1)True, 2)False,3)True, 4) False 5) False

Fill in the blanks:

1) _____________ is an informal way of describing the operating principle of a program without any strict programming syntax or rules.
2) A program is a set of _________ given to a computer to perform a particular task.
3) _____________ is the second generation language.
4) An _____________ executes one statement at a time and hence slower than a compiler.
5) ____________ is one of the important components of a program.

Answers: 1) Pseudocode,2) Instructions,3) Assembly language,4) Interpreter, 5) Documentation

Match the columns:

1) Java	a) Translator
2) First Generation Language	b) Diagrammatic representation of an algorithm
3) Reliable	c) Debugger
4) Flowchart	d) Assembly Language
5) Interpreter	e) Diagrammatic representation of a pseudocode
	f) Ability to handle any type of error and should be able to display error message or warning
	g) High level language
	h) Machine Language

Answers: 1) (g),2) (h) 3) (f) ,4) (b), 5) (a)

Long Answer Questions

1) What is problem solving?
2) What is an algorithm? State the properties of an algorithm.
3) Write a brief history about flowchart.
3) What is a flowchart? List the rules of drawing a flowchart?
4) Explain the basic control structure of flowchart with the help of diagrams.
5) What is a pseudocode? List two advantages and disadvantages of pseudocode.
6) What is a program? What are the characteristics of a good program?
7) Why do we need a programming language?
8) Explain the different types of programming languages.
9) What is a translator? Explain the different types of translator.
10) Write algorithm and draw flowchart

    To find the sum of digits of a number

    To find the factorial of a number

    To find whether a number is prime or not

11) Write pseudocode

    To find the largest of three numbers

    To find the Simple Interest

Space for learners notes

Space for learners notes

Suggested Reading List

1) The Art of Programming Through Flowcharts & Algorithms By Anil Bikas Chaudhuri Laxmi Publications (December 30, 2005)

2) https://nptel.ac.in/courses/106105171/1

UNIT 2 HISTORY OF C, VARIABLES, CONSTANTS AND OPERATORS IN C

Space for learners notes

CONTENTS

Space for learners notes

2.1 INTRODUCTION

Like natural languages, C language also has sets of alphabets, symbols (also operators), library words etc. Also it has specifications for writing language statements, mathematical expressions, formulas etc. The C language is case-sensitive. This means that

- if a term is defined in C library in upper-case then if we use the term in our program only in uppercase, and
- if we define a term in upper-case in a program then the further use of the term should be in upper-case throughout the same program.

2.2 OBJECTIVES

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

- get acquainted with the history of C language and its features.
- understand the structure of a C program.
- know how to write, compile and execute a C program in Windows and Linux operating system.
- learn the different types of errors generally occur while compiling and running a C program.
- understand the different types of tokens in C in detail, e.g., keywords, identifiers, constants etc.
- know about data types and its categories in detail.
- learn about variable concept and basic input/output functions with syntaxes in C.
- have idea about different operators and their use. You will also be able to understand what does operator precedence and associativity mean.

2.3 HISTORY OF C

C is a general-purpose language which was developed for the **UNIX Operating System**. **C** was developed in the early **1970**s by **Dennis M. Ritchie** who was an employee from **AT&T**(Bell Labs).

Ritchie with Ken Thompson and others was initially involved in a project called **Multics in 1960** at AT&T. Development of an Operating System for large Computers that can be used by hundreds/thousands of users was the main goal of Multics. But the project Multics could not be able to produce useful system. So, in **1969** the project was withdrawn by AT&T.

Space for learners notes

Thompson and Ritchie began to work on the development of a new file system for **DEC PDP-7**. With the knowledge from the Multics project, they were able to make improvements and expansions. At last it took a complete form and Brian W. Kernighan termed the system as **UNIX**. Side-by-side, Ken Thompson was working on the development of a programming language called B (derived from Martin Richards **BCPL**). UNIX was enabled with an interpreter for the language B. After the development of B, it was used for further development of UNIX.

In B, as most were expressed in machine codes (not easy to work-with) and the other drawbacks created various lags. These lags forced Ritchie to develop the programming language C, keeping most of the language B syntax. Thus C became a powerful mix of high-level functionality. Later-on, most of the UNIX components were rewritten in C.

The **American National Standards Institute** (ANSI) established a committee in the year 1983 which provides the definition, the ANSI standard, or "ANSI C" (in late 1988).

2.4 FEATURE OF C

C Language is simple in terms of syntax and variety of decent functionalities. Following are the important features of C language.

- Robust language with a very rich set of operators and library functions. C has provisions for creation of programmers' own library of functions.
- Due to its variety in data-types and also rich set of powerful in-built functions, the programs written in it are fast and efficient.
- It is highly portable as because programs written in it can be run on different machines with a little or no modification.
- It is known as middle-level language as its compiler has the capabilities of a low-level language (i.e. assembly language) along with the features of a high-level language. This is the reason that C language is well suited for writing both System Software and Application Software.
- Extendibility is one of the major features of C language i.e. is has the ability to extend itself.

2.5 STRUCTURE OF A C PROGRAM

The structure of C program with its programming elements is shown below:

Space for learners notes

Pre-processor directive (e.g., Header File inclusion)
Global Variable(s)
User-defined Functions Declarations
main() Function
User-defined Functions Definitions

A simple C program that prints 'Hello World' on the monitor(output screen) is shown below.

```
#include<stdio.h>
void main()
{
        printf("Hello World!");
}
```

Now, let's try to understand the above program as per the structure mentioned above. First line is the **Pre-processor Directives**. We know that natural languages like Assamese, English etc. have their own dictionary/library. Thus C also has its library and it consists of some pre-written files, known as **header files**. In a C program, we can include header files as per the requirements of the program. Here, in this program, the header file, namely '**stdio.h**' is included as it the basic header file that needs to be included for a simple C program.

STOP TO CONSIDER

The extension of a header file in C is '**.h**'. The header file, **stdio.h,** contains the basic input/output functions and other items those are important for a simple C program.

There is no line for **Global Variable(s) Declaration.**

There is no line for **User-defined Functions Declaration**.

The remaining lines are under **main() Function**. The lines inside the brackets **{ and }** are under the **main()**. **Function** will be discussed in the Sections/Units to follow. But, for now take the function as a block of lines (also known as statements in programming). So, the **main()** function (block set by { and } brackets) contains the statement,

```
        printf("Hello World!");
```

which will display the message, **Hello World!**, on to the computer screen/monitor. The main() function is necessary for each and every C program as when we execute a C program, the execution starts from **main()**. The necessity of the use of '**void**' will be understood in the following sections/units.

Space for learners notes

Also, there is no line for **User-defined Functions Definition.**

Each and every line in the above program is called a **Statement**. As in English language a sentence is marked end with full-stop(.), a **C statement** ends with a semicolon(**;**) except the few e.g., the above mentioned Pre-processor Directives (statements).

2.6 WRITING, COMPILING AND EXECUTING A C PROGRAM

From writing a C program to its execution, various softwares are required. These are namely: a **Text Editor** for editing; a **Compiler** for compiling; a **Debugger** for debugging; a **Linker** for linking.

Linker

$$\begin{tikzcd}\boxed{\text{Source Code}} \arrow[r, "\text{Compiler}"'] & \boxed{\text{Object Code}} \arrow[r, "\text{Linker}"'] & \boxed{\text{Machine Code}}\end{tikzcd}$$

Thus, each of the softwares is to be installed in your machine separately for this purpose. There are software packages available bundled with all the above mentioned system softwares with additional functionalities. Thus these softwares enable a user/ programmer to write, compile, debug and execute (run) a C program. This kind of packages is generally termed as **Integrated Development Environment** (IDE). Not only for C language, there are **IDEs** available for other Computer Languages also. For C language an example of **IDE** is **Turbo-C**.

For a C program to execute, the file (saved with the extension **.c** or **.C**) containing statements written in C has to be translated from a high-level language to a low-level language (executable code). This task is accomplished with the help of compiler and linker. A compiler takes in a source code and produces an object code which is further passed to a linker. A linker links the source program with the external entities such as the header files and other user-defined files, if any, to produce the final executable code.

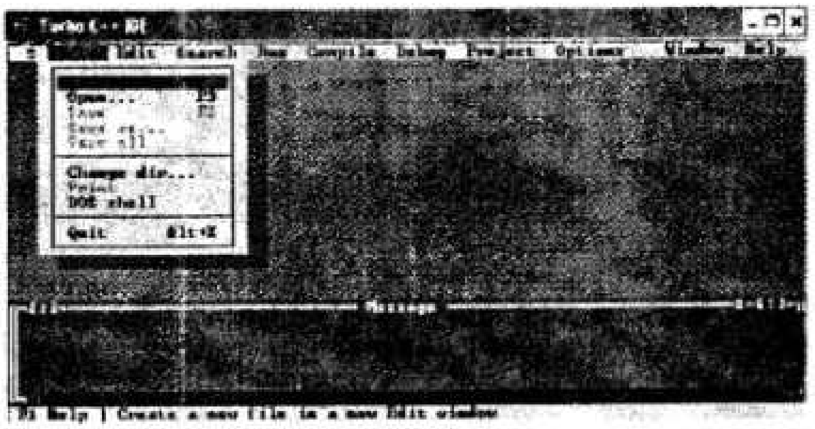
For Windows users, the procedure for writing and executing a C program in **Turbo-C** is shown below.

1. Open the **Turbo-C IDE.**

   Double-click the icon. It can be found in **C:\TC\bin** folder. Once Turbo-C opens up, one should be able to see the environment like below.

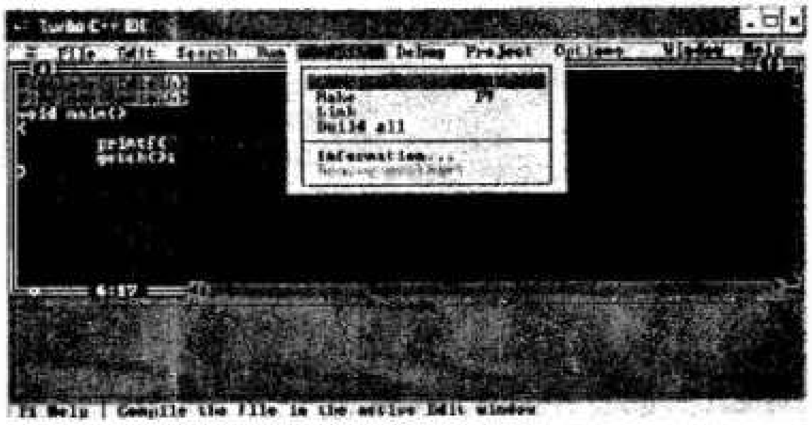
Space for learners notes

2. Select '**New**' from the '**File**' menu.

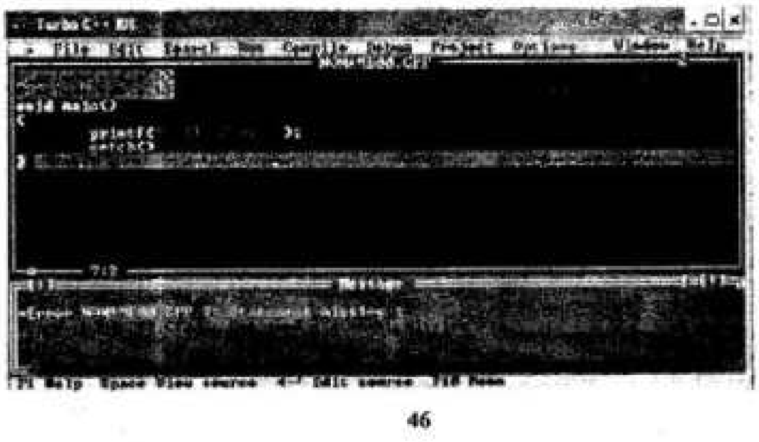

Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
Open... F3
Save F2
Save as...
Save all
Change dir...
Print
DOS shell
Quit Alt+X
Message
F1 Help | Create a new file in a new Edit window

Write the program and save it in a location of your choice. The default location where a program is saved is '**C:\TC\bin**'. To save a program, press **F2** or select '**Save**' from the '**File**' menu. The extension of the file must be **.c** or **.C** for a C program.

3. To compile select '**Compile**' from the '**Compile**' menu. (or press **Alt+F9**)

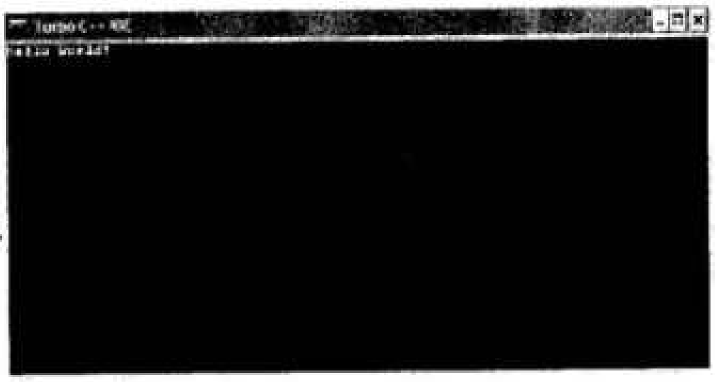

Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
Make F9
Link
Build all
Information...
F1 Help | Compile the file in the active Edit window

If there are some errors in the program, the error messages will be displayed in the 'Message' box. Shown below is an example where a semi-colon is missing at the end of the statement. Such compile-time errors need to be corrected for a successful compilation.


Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
Message
F1 Help Space View source ←┘ Edit source F10 Menu

4. After a program is compiled, it can be executed. To do so, select 'run' from the 'run' menu.

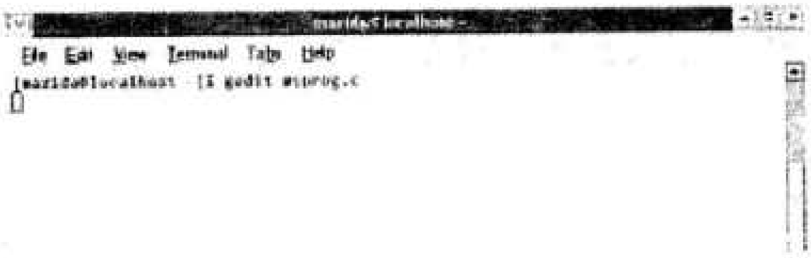
   The output of the program will be displayed on the screen. For our example, 'Hello World' will be displayed.



```
Hello World!
```

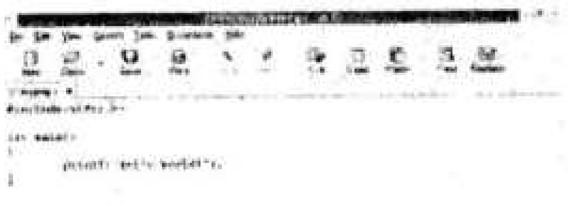
For ***linux/unix*** users, the process for compiling and execution is shown below:

1. Open a text editor (via a terminal). For demonstration, **gedit** text editor is used. The user is however free to choose any text editor.



```
[marida@localhost ~]$ gedit myprog.c
```

2. Write the program and save it.



```
#include<stdio.h>

int main()
{
        printf("Hello World!");
}
```

3. Compile the program by a compiler. Common compilers available in a unix environment are cc, gcc etc. To compile a program, type in the terminal

   <Compiler_name> <Program_name> [**-o** *<Output_file_name>*]

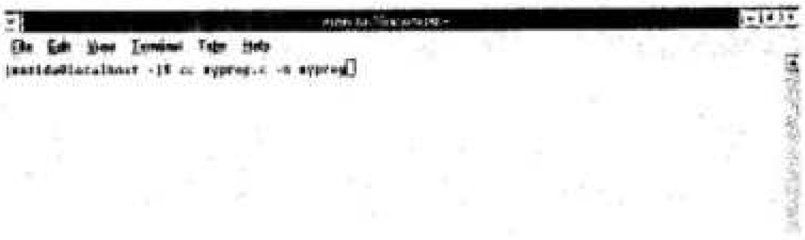
Space for learners notes

Space for learners notes

For example,

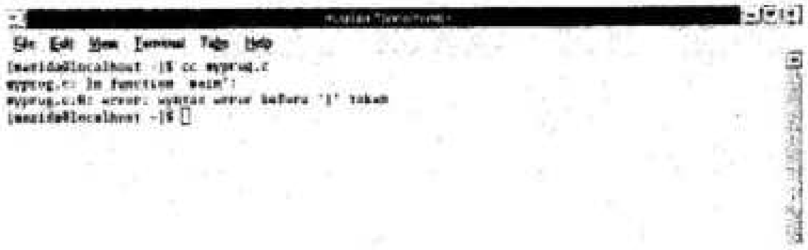
cc myprog.c –o myprog

Thus according to syntax '**cc**' is the compiler name, '**myprog.c**' is the program file name and '**myprog**' is the output file name. '-o' is used to tell the compiler that the output file to be produced should be '**myprog**'.



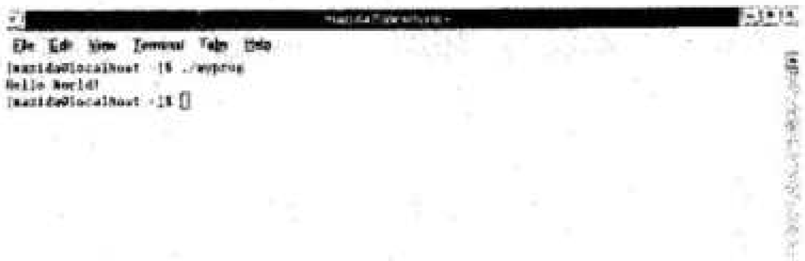
```
[nazida@localhost ~]$ cc myprog.c -o myprog
```

If the program has some compile-time errors, error messages will be displayed bearing the line numbers; as in the example below- line numbered 6 has a syntax error.



```
[nazida@localhost ~]$ cc myprog.c
myprog.c: In function 'main':
myprog.c:6: error: syntax error before '}' token
[nazida@localhost ~]$
```

4. After successful compilation, the program can be executed.



```
[nazida@localhost ~]$ ./myprog
Hello World!
[nazida@localhost ~]$
```

STOP TO CONSIDER

In the syntax for compilation the items appearing inside [ ] are optional. So, if no name for the output file is provided, the default output file produced after compilation is '**a.out**'.

2.7 ERRORS IN C

There are various types of errors those may occur during compilation and execution of a C program. These are:

- **Syntax Error**

  Such error occurs when a statement does not comply by the rules of the language. Those errors are detected by the compiler and need to be corrected before the code is executed. So, this kind of errors is also known as a compile-time error. Some common compile-time errors are statement missing semi-colon(s), mis-spelt keywords, undefined identifier etc. The example below has a statement missing semi-colon.

  Example-1:

```
#include<stdio.h>
void main()
{
    int a,b          //semi-colon(;) missing
    a=2;
    b=3              //semi-colon(;) missing
    printf("Sum=%d", a+b);
}
```

  Compilation Output: Declaration syntax error, statement missing semi-colon

- **Semantic Error**

  These errors violate the meaning or the logic of the language and hence fail to produce the desired result. These errors are also detected by the compiler most of the time. Consider the following statements of C program.

```
int a=2, b=4, c;
a + b = c;
```

  Compilation Output: Lvalue required

- **Run-time Error**

  Even after successful compilation, there are times when a program fails to execute properly. Some situations which may generate run-time errors are:

  - when a number is divided by zero,
  - while trying to open a file that does not exist,
  - accessing a list of elements beyond its boundary etc.

  Consider the following statements,

```
b = x-y;
c = a / b;
```

  Consider symbols – and / mean subtraction and division respectively. Now, the above statements are correct. But if it happens that for the first statement the value of **b** becomes **0**. Then in the second statement **a** will be divided by **0**, **a/0** which is an invalid operation and will lead to termination of the program execution.

Space for learners notes

Space for learners notes

2.8 C CHARACTER SET

C Character Set	
Letters:	**Digits:**
Lower Case: a.....z	0,1,....,9
Upper Case: A....Z	
Symbols:	
, Comma	} Right brace
. Period	[Left bracket
; Semicolon	] Right bracket
: Colon	< Opening angle bracket/ or less than sign
? Question mark	> Closing angle bracket/ or greater than sign
' Apostrophe	/ Slash
"Quotation mark	\ Backslash
! Exclamation	| Vertical bar
# Hash	= Equal sign
$ Dollar sign	- Minus sign
^ Caret	_ Underscore
& Ampersand	+ Plus sign
* Asterisk	~ Tilde
(Left parenthesis	
) Right parenthesis	
{ Left brace	
White Spaces:	
Blanks	New Line
Horizontal Tab	Carriage Return
Vertical Tab	Form Feed

Table-2.1: C Character Set

A character denotes an alphabet, digit or special symbol used to represent information. Like natural languages, computer language will also have well defined character set, which is useful to build the programs.

The C Character Set includes alphabets: a to z (lower case), A to Z (upper case), digits: 0 to 9, special symbols: !, @, #, $, % and many more. It also includes white spaces such as blank, tab, new line, form feed etc. ***Table-2.1*** list the C Character Set.

The C language follows the American Standard Code for Information Interchange (ASCII) for representing characters where each character has a unique 7-bit binary value representation. The characters are coded from 0000000 to 1111111, forming a total of 128 characters.

Space for learners notes

There are few ASCII characters which are unprintable, i.e., they will not be displayed on the screen. These are used only to perform some specific functions aside from displaying text. Examples are backspace, newline, alarm.

2.9 C TOKENS

In a C program, tokens are the building blocks. A token is the smallest individual element or unit in a program. Tokens are composed of the characters, symbols etc. Programs are coded using these tokens according to the rules of the language. In C language, tokens can be classified under five categories and they are:

- Keywords
- Identifiers
- Constants
- Operators
- Special Symbols/Characters

Let us consider the following C program.

Program-1:

```
void main()
{
        int x;
        printf("Enter a number:");
        scanf("%d", &x);
        x = x + 1.2 * x;
        printf("Incremented value:", x);
}
```

Tokens used in the above **Program-1** under different categories are presented in the **Table: 2.2**.

Type of Tokens	**Tokens Used**
Keywords	int, void
Identifier	x, main, printf
Constant	Enter a number:, 1, 2, Incremented value:
Operators	Addressof (&), Addition (+), Multiplication (*)
Special Symbols	(,), {, }, ;, ", %, &, :

Table- 2.2 Typed of tokens used in Program-1

Space for learners notes

2.9.1 Keywords

Keywords are the reserved words that have well-defined purposes. A keyword should be used only for its particular use and not for any other purposes like naming a variable or function. Keywords, when used in programs, should not be modified or altered from their defined format.

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

Table 2.3: Complete set of keywords used in C language

They are specific to programming languages, that is, every programming language has their own set of keywords. **Table-2.3** gives the complete set of keywords for the C language.

2.9.2 Identifiers:

Identifiers are the name given to the entities such as variables, constants, functions, files, structures etc. Just as persons, cities or streets have names, the C entities such as variables, functions, files etc. are given uniqtie names (identifiers) for their identification in a C program.

Rules for Naming Identifiers:

Identifiers are basically composed of alphanumeric characters i.e. alphabets or digits. The basic rules for naming an identifier are:

- ➢ The first character should be an alphabet or an underscore.
- ➢ No special symbols except the underscore, is allowed in an Identifier.
- ➢ The Identifiers could be of any length but only the first 31 characters are significant.
- ➢ Keywords cannot be used as identifiers.

Following are some examples of identifiers-

Identifier	**Valid?**	**Remark**
Sum	valid	
char	invalid	keywords are not allowed
price#	invalid	special symbols not allowed
var 1	invalid	blank space not allowed
avg_num	valid	

Although any combination of letters, numbers and underscore is an identifier, it is advisable to create an identifier that reflects the meaning and purpose of the entity.

Space for learners notes

2.9.2 Constants, Operators and Special Characters:

Constants can be defined as fixed values that do not alter during the execution of a program. Following are the different types of **Constants**:

- Integer Constants
- Real Constants
- Character Constants
- String Constants
- Special Character Constants
- Symbolic Constants

Operators are one of the important building blocks in C language. Operators are used to perform specific mathematical and logical computations/comparisons on operands. Few examples of operators are: +, -, /, * etc. In ***Unit-Section 2.14***, Operators will be discussed in detail.

Special Symbols are the symbols other than the operators. These are used in programs for various purposes as and when necessary. Refer to ***Table-2.2*** for special symbols used in **Program-1**.

Now, let's discuss different types of **Constants** one by one.

- <u>**Integer Constants:**</u>

Integer Constant is a whole number (without ***decimal point***). It can be defined as a ***sequence of digits*** (from **0 to 9**). An **integer constant** can be ***preceded*** by – (for **–ve** value) or optional + (for **+ve** value). Following are some examples of **Integer constants,**

Integer Constant	**Valid?**	**Remark**
1	valid	
300	valid	
-20	valid	
+15	valid	
20.3	invalid	Not a whole number (without decimal point)
1,200	invalid	Comma not permitted between digits

- <u>**Real Constants:**</u>

Real Constant is a number containing ***fractional part*** (with ***decimal point***). It is also called **Floating Point Constant**. Like integer constant it also can be ***preceded*** by – or optional +. For example following are some valid Real Constants.

20.5 -12.40 +2.67 -.50

Space for learners notes

A **Real Constant** can also be expressed in exponential form. The form is:

mantissa *e* exponent

Mantissa can either be an integer or a real number. The **Exponent** is an integer with – sign or + sign(optional). '**e**' separates the **mantissa** and **exponent**. Instead of '**e**' we can write '**E**' also. Following examples illustrates this representation.

In Real Form	**In Exponential Form**
1286.45	12.8645e2 or 12.8645E2
0.034	3.4e-2 or 3.4E-2
1,200	Comma not permitted between digits

- **Character Constants:**

A **Character Constant** is a ***single alphabet/digit/symbol/blank space*** enclosed in ***single quotes*** (''). For example, **'a'**, **'A'**, **'5'**, **'9'** are valid **Character Constants.** But do not confuse **'5'** and **5** as first one is a character constant and other one is an integer constant.

- **String Constants:**

A **String Constant** consists of a sequence of characters (*alphabets/digits/symbols/ blank spaces*) enclosed in ***double quotes*** (" "). Following are few examples of valid string constants.

"A" "IDOL" "Hello! Welcome to IDOL" "1+2+3+4"

- **Special Character Constants:**

There are some **Special Character Constants** which are basically used in functions that display data. These character constants combine '\' with an alphabet/symbol. These character constants are also termed as **Backslash Character Constants.** Following **Table-2.4** lists the different Special Character Constants available in C with their meaning.

Special Character Constants	**Meaning**
'\n'	adding new line
'\a'	adding an alert(bell)
'\b'	applying backspace
'\f'	adding form feed
'\r'	adding carriage return
'\t'	applying horizontal tab
'\v'	applying vertical tab
'\?'	adding question mark in the output
'\\'	adding back slash in the output
'\0'	**null** value
'\''	adding single quote in the output
'\"'	adding double quote in the output

Table-2.4: Special characters available in C

Program-2(in ***Turbo-C***): Using few of the above special character constants

```
#include<stdio.h>
#include<conio.h>
void main()
{
        clrscr();
        printf(“Courses offered by GUIDOL\a”);
        printf(“\nMA in Assamese”);
        printf(“\nMA in English”);
        printf(“\tMA in Economics”);
        printf(“\nMA in History”);
        printf(“\\ MA in Political Science”);
        printf(“\tand many other courses\?\?\?”);
        getch();
}
```

Now, before discussing the ***Program-2***, we need to know some basic statements in C. Turbo-C, as discussed above, is a compiler for Windows which comes in the form of a software package (IDE) enabling a user/programmer to write, compile and execute a C program. In case of Turbo-C (in ***Windows***) the statements,

```
        clrscr();
        getch();
```

are necessary. But in case of the ***Unix/Linux***, the above statements are not necessary at places where they were put in the above program.

The function **clrscr()** is used to clear the screen. It is contained in the header file '**conio.h**'. It is basically used in programs using Turbo-C IDE (in Windows) as because while running the programs the output screen may contain the output from earlier executed programs. So, here in the ***Program-2*** this function is used before the first output statement (i.e. **printf**). But in Unix/Linux, the C library does not contain the header file '**conio.h**' and so **clrscr()** function cannot be used.

Like **clrscr()** function, the **getch()** function also contained in '**conio.h**'. So, in Unix/Linux this function does not work. This function takes a character from the keyboard. In the ***Program-2*** this function is the mentioned as the last statement but logically it is not required. Calling the **getch()** function as the last statement keeps the program waiting for a character input(i.e. a key to be pressed) to complete execution. This, in turn, lets the users view the previous outputs displayed on the screen.

The other statements except the last one will display the messages put within " " (double quotes) on to the screen. The **printf()** is a function which is used to display data on to the screen.

Now let's discuss the output of the above program.

Space for learners notes

Space for learners notes

Output:

```
Courses offered by GUIDOL
MA in Assamese
MA in English
        MA in Economics
MA in History\ MA in Political Science
        and many other courses???
```

Explanation:

- ✓ First **printf()** displays the message (within " ").
- ✓ Second and third **printf()** displays the messages (within " ") in new lines because of the special character constant **'\n'**.
- ✓ Fourth **printf()** displays the message in new line but after a tab space because of the special characters **'\n'** and **'\t'** respectively.
- ✓ Fifth **printf()** displays the message in new line because of the special character **'\n'**.
- ✓ Sixth **printf()** displays the message in the same line just after the last message in the same line printed with the symbol '\' and a single space before the message because of the special character '\\' followed by a space. This message is not displayed in a new line for not using the **'\n'** special character.
- ✓ The last **printf()** displays the message in new line but after a tab space and three **'?'** marks are at the end of the message because of **'\t'** and three **'\?'** special character constants.

• **Symbolic Constants:**

A **Symbolic Constant** can be defined as the combination of a ***name*** (except keywords) and a **constant value.**

The syntax of defining **Symbolic Constant** is

#define symbolic-constant-name constant-value

and it should be defined before "**main()**". In the process of compiling a program, first there is a pre-processing step in which, apart from other tasks, symbolic constants are processed, i.e. wherever in the program the **symbolic-constant-name** appears it is replaced by the **constant value.** Consider the following C program.

Program-3(in Turbo C): Demonstrates the use of Symbolic Constant.

```
#include<stdio.h>
#include<conio.h>
#define PI 3.14
void main()
{
        int rad;
        float area;
        clrscr();
        printf("Enter the value of Radius:");
        scanf("%d", &rad);
        area = PI *rad*rad;
        printf("The area of the circle = %f", area);
        getch();
}
```

Space for learners notes

Here, in the above program (Program 3) **PI** is the **symbolic constant** with the value **3.14**. So, the statement,

```
        area = PI*rad*rad;
```

during pre-processing, becomes,

```
        area = 3.14*rad*rad;
```

i.e. **PI** is replaced by **3.14**(as defined).

CHECK YOUR PROGRESS:

1. Who developed the C language?
2. Why does the C language termed as middle-level language?
3. What is header file?
4. What do you understand by errors in C?
5. What is keyword? Write down five keywords available in C.
6. Write down the rules for naming identifiers in C.
7. IDE stands for ______________________________.
8. ASCII stands for ____________________________.
9. Division by zero (0) is a ______________ error.

2.10 DATA TYPES

In a program, we have to work with data along with other functionalities. In a C Program, we can store/assign data and use the stored data along with other functions. So, in C language there are various "**Data Types**" to cover all the possible data that can be used in C programs. But C also allows creation of new **Data Types** and also customization/enhancement of offered **Data Types** as per need of a program. Data Types in C can be broadly classified in three classes (Fig-2.1), namely:

- Primary/Built-in Data Type,
- Derived Data Type and .
- Built-in

Space for learners notes

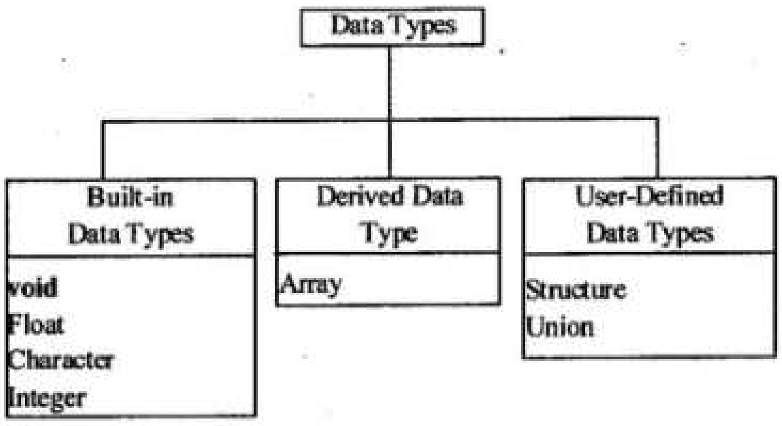


Fig-2.1: Data Types in C

2.10.1 Primary/Built-in Data Type:

The **Built-in Data Types** available in C are listed in ***Fig-2.2***. Basically, C supports five **fundamental types** (in ***Fig-2.2*** marked bold and in small cases) and they are **char**, **int**, **float**, **double** and **void**. The others are the extensions (in ***Fig-2.2*** marked normal) of the **fundamental types**.

Each of the **fundamental type** has its ***size*** (in **bytes**) and thus depending on the ***size*** it has ***value range***. The ***size*** in **bytes** (**1 byte = 8 bits**) and range of each of the **fundamental types** are listed in the following ***Table-2.5***.

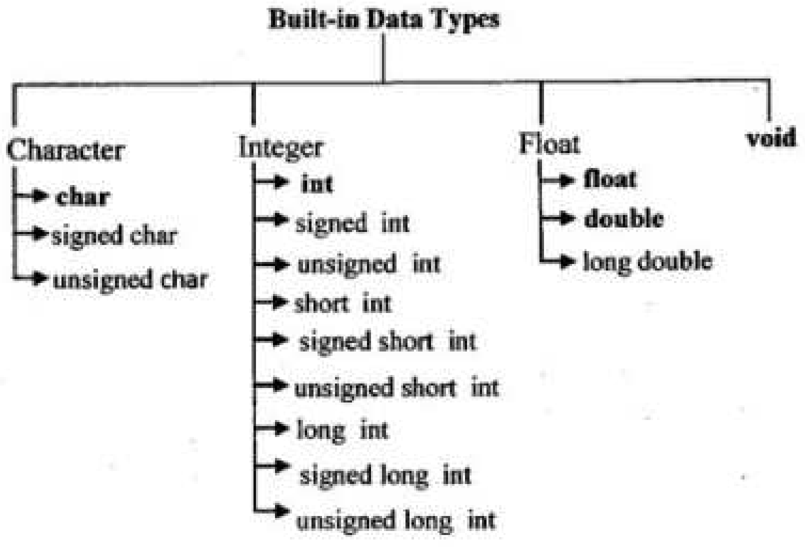


Fig-2.2: Built-in Data Types in C

Data Type	Size (in bytes)	Value Range
char	1	-128 to 127
int	2	-32,768 to 32,767
float	4	3.4e-38 to 3.4e+38
double	8	1.7e-308 to 1.7e+308

Table-2.5: Data types, size and value range

Space for learners notes

For a **character** data, the **fundamental type** is **char** and it is of *size* **1 byte** (of internal storage i.e. primary memory). A character means the value like '**A**' but from the ***Table-2.5*** the ***value range*** of **char** is **-128 to 127**, which is a number range. Now, you may wonder how a character data is represented internally, i.e. represented either in the form like '**A**' or as a number. It is to be noted that every ***alphabet/ number/symbol*** present in a ***keyboard*** is associated with an **ASCII** value (a whole number). For example, **ASCII** value for '**A**' is **65**, '**a**' is **97**, '**=**' is **104**, '**+**' is **43** etc.

Integers are whole numbers, i.e. numbers without decimal point. The fundamental type for an **Integer** data is **int**. The size of **int** type depends on the **word size** for a particular machine. A **word** is defined as the maximum number of bits that a CPU can process at a time. The progress in the hardware technology of CPU enables computers to handle larger amount of bits. A **word size** can be as high as **64 bits (8 bytes)**. **For our discussion, let's consider that the word size is of 16 bits (2 bytes).**

The terms, **short** and **long** (from ***Fig-2.2***), are called **Modifiers**, i.e. they are used to ***modify/extend*** the size of the **fundamental types** and thus the ***value range*** also increases.

- In case of **short**, the *size* is same as the *size* of the associated **fundamental type**. For example, *sizes* of **int** data-type and **short int** data-type are the same, i.e. **2 bytes**.
- But the **long** modifier generally ***doubles*** (exception in few cases) the *size* of the associated **fundamental type**. For example, the *size* of the **long int** data-type is **4 bytes**, i.e. the ***double*** the *size* of **int/short int** type (**2 bytes**).

The terms, **signed** and **unsigned** (from ***Fig-2.2***), are called **Qualifiers**, which have no effect of the *size* of the type but have effect on the ***value range***.

- In case of **signed** data, the ***left-most bit*** is reserved for the sign (+ve or –ve), and so this will allow data to be either negative or positive. Thus the all the bits present except the ***left-most*** one are used for the data.
- In case of **unsigned** data like **signed**, no bit is reserved for the **sign** and thus all the bits present are used for the data.
- If along with a fundamental type you don't mention **signed/unsigned** then ***by default*** that fundamental type will be treated as **signed** one.

And therefore, in **signed** data, the **values ranges** from a **–ve** to a **+ve** value. But in **unsigned** data, the ***values ranges*** from **0** to a **+ve** value. For example, in case of

Space for learners notes

signed char data-type the ***size*** is **1 byte** and ***value range*** is **-128 to 127**. But in case of **unsigned char** data-type the ***size*** is also **1 byte** but the ***value range*** is **0 to 255** as for the ***maximum value*** (255) all the 8 bits will be **1**s [**11111111**].

STOP TO CONSIDER

The maximum value in 8 bits is **255**

$(1*2^7)+(1*2^6)+(1*2^5)+(1*2^4)+(1*2^3)+(1*2^2)+(1*2^1)+(1*2^0)$

$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$

The following ***Table-2.6*** illustrates these more clearly.

Type	**Size** (in bytes)	**Value Range**
char/signed char	1	-128 to 127
unsigned char	1	0 to 255
int/signed int/short int/signed short int	2	-32,768 to 32,767
unsigned int/unsigned short int	2	0 to 65535
long int/signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
Float	4	3.4e-38 to 3.4e+38
Double	8	1.7e-308 to 1.7e+308
long double	10	3.4e-4932 to 1.1e+4932

Table-2.6: Signed/unsigned data types with size and value range

2.10.2 Derived Data Type:

Though C supports various of data-types, in practice most of the times a program requires large amount of data of a particular type to work with. So, to facilitate this, C needs a powerful data-type for manipulation of large data items of same type. C comes with a category known as **Derived Data Type**. **Array** is the **Derived Data Type** supported in C. Basically an **array** is a list of continuous memory locations (in primary memory) of same type. **Array** not only supports ***fundamental types*** or ***its variations*** but also **User Defined Types** (will be discussed more elaborately in **Unit-4**).

2.10.3 User Defined Data Type:

The term, "**User Defined Data Type**", is self explanatory. This kind of data type is created by the **User** (the programmer) according to his/her need. These are of three user defined data types and they are: **Structure**, **Union** and **Enum**.

Structure and **Union** will be discussed in ***Unit-6***.

Lets' discuss about **Enum**. We know that the set from where an integer, real or character value can be considered. For example, an integer(**int**) may be any value ranging from **-32,768 to 32,767** (from ***Table-2.6***). But there may be situations where we need to restrict the range/pool of values according to the need for a specific purpose. So, **Enum**(enumeration) is a **User Defined Data Type** that can take one value from the values those are predefined. **enum** keyword is used to define the enumerated data type. The syntax for defining this type is:

enum enum-name {value-1, value-2, ..., value-n};

where, **enum-name** is the name of type and **value-1**, **value-2**,.... is the list of values.

Space for learners notes

2.10.4 Typedef:

The **typedef** keyword is used to temporarily (in most of the cases) assign an ***alias*** i.e. ***alternative name*** to a **fundamental/derived/user defined** data type.

The syntax for using typedef is:

typedef existing-type-name alternative-name;

For example, in a program we have to work with a data of data-type **unsigned long int**. This data type name is a long one. Now, we can assign, say **ulint**, as a new name which is much shorter than name of above the type using the **typedef**. So, we can do this by using the following statement:

```
typedef unsigned long int ulint;
```

Now, in the program, when we have to declare a variable of type **unsigned long int** we can use the new name **ulint** instead. Suppose we want to declare a variable called **SUM** of the above type then we can type,

```
ulint SUM;
```

STOP TO CONSIDER

typedef is used with user defined data types, when names of the data types become slightly complicated and too long to use in programs.

2.11 VARIABLES & STORAGE CLASSES:

Space for learners notes

2.11.1 Variables:

Constants are the data those remain unchanged during the execution of a C program. But in a C program we need data those should be able to take different values at different times. This enforces the requirement for a provision of storages (in **main memory**) for storing values(different values at different times) such that at different times a storage can take different values during the program execution. So, programming languages provide the concept of **Variable**. A **variable** is a **named memory location** (in main memory) where one can store different values (of a particular ***type***) at different times. In **Program-1**, the **identifier 'x' is a variable** which can store an ***integer value***. Now, you may think of how we can say that variable **'x'** is an **integer variable**!!!

```
int x;
```

The above statement in the ***Program-1*** ensures that **'x'** is an integer variable as **int** is mentioned before **'x'**.

Like registering (declaring) a name to a newly created company before it starts operating, a **variable** should also be **declared** (i.e. **named**) before its use in a C program.

The syntax for declaring a variable is:

data-type   variable-name;

Here, data-type refers to the type of variable **variable-name**, i.e. what type of data the variable can store at a time. This kind of statement is known as **Variable Declaration Statement**.

The ***rules for naming*** a **variable** is the same the rules for naming an **identifier** as already discussed in ***Unit Section 2.9.2***.

There are different **ways for declaration of variable(s)** in a C program and these are:

- ✓ **A variable** should be declared as in the syntax mentioned above before its use.
- ✓ **Variable(s)** declaration statements should be the first statements in a **function** (e.g., **void main()** in ***Program-1***).
- ✓ For declaring more than one variable of same type in ***one statement***, the syntax is:

  int a, b, c, sum;

  where a, b, c and sum are integer variables. We may also declare the above four variables ***individually*** like,

  int a;

  int b;

  int c;

  int sum;

- ✓ For variables of different types, different statements are required for each of the types. Suppose you want to declare a, b as integer variables and x, y as floating point variables. Then we have to declare them as:

```
int a, b;
float x, y;
```

Space for learners notes

2.11.2 Storage Classes:

As mentioned earlier a variable is a storage area of primary memory where we can store/assign data in a C program. Apart from primary memory, the CPU registers are also a kind of memory locations for the variables declared in a C program. Here comes the concept of **Storage Class**.

Storage Class is related with declaration of variables. It specifies the part of storage space (memory/registers) to allocate memory for variables declared in C program. It also specifies the scope of a variable i.e. the lifetime of a variable during execution. **Lifetime** means that whether the declared variable will exists during the execution of the program or will exists only within the block (generally related with function) in which the variable is declared. These two kinds of lifetime are termed as **Global** and **Local**. The storage class also determines the variable visibility level i.e. a variable may has global lifetime but only visible from within the block in which it is defined.

Four storage classes are provided in C and they are automatic, register, static, and external. The storage class specifiers are listed in the ***Table-2.7*** with their meaning.

Storage Class	**Meaning**
auto	Local variable with Local lifetime, i.e. only to Function(block) in which it is declared. It is the ***default specifier***.
static	Local variable with Global lifetime.
extern	Global variable with Global lifetime, i.e. accessible from everywhere in a C program.
register	Local variable whose storage space is the CPU register.

Table-2.7: Storage classes and meaning

Thus, the syntax for a variable declaration that uses a storage class is:

Storage_Class_Specifier Data_Type Variable_Name;

Following are the few variable declaration statements that use the storage class specifiers.

```
auto int a;       ←Automatic Storage Class
static int b;      ←Static Storage Class
extern char c;    ←External Storage Class
register int d;   ←Register Storage Class
```

Space for learners notes

2.12 OUTPUT AND INPUT IN C

In a C program we have to display, i.e. output, data/messages on to the screen and as well as to take data as input during the execution. There are functions which are used for output and input in C.

Output in C:

As discussed earlier, **printf()** function is used to display message, e.g., the statement

```
printf("Welcome to IDOL");
```

will display the message, Welcome to IDOL, on to the screen. The syntax of the printf() function is(in a simplified form):

```
printf(Formatted-String, [Variable-1, Variable-2, ...]);
```

Here, the **Formatted-String** may only be a string/message to be displayed or be a string embedded with **Format-Specifiers** like **%c, %d, %f** etc. The **Variable-1, Variable-2, ...** are the variables those values to be displayed and these are optional, i.e. to mentioned as per requirement. The **Format-Specifiers** specifies, in general, the type of the variable whose value is to be displayed. Consider the following C statements.

```
int a;
a=100;
printf("The Value = %d", a);
```

The output of the above **printf()** function is,

```
The Value = 100
```

So, in the place of **%d** 100 (i.e. value of **a**) is displayed.

Following are the **Format-Specifiers** present in C.

%c	← for **char** type
%d or %i	← for **signed int** type
%u	← for **unsigned int** type
%l or %li or %ld	← for **signed long int** type
%lu	← for **unsigned long int** type
%f	← for **float**
%lf	← for **double** type
%Lf	← for **long double** type
%s	← for **string** type (will be discussed in ***Unit-4***)

Input in C:

In ***Program-2***, the statement

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

is an input statement. Here **scanf()** is the function to take input during the execution of the program. The syntax of **scanf()** function is,

scanf(Formatted-String, &Variable-1, &Variable-2, ...);

Here, the **Formatted-String** only the ***string containing*** the **format-specifiers** for the types of the values to be input. The **Variable-1, Variable-2, ...** are the variables to which the inputs are to be stored. The symbol '**&**' (before each of the variable names) used here, is termed as **Address-of operator** which gives the location address of the variables those follow it. Consider the following C program.

Program-3: Program which takes a number as input and displays it.

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

void main()
{
        int a;
        clrscr();
        printf("Enter an Integer Value: ");
        scanf("%d", &a);
        printf("The Value = %d", a);
        getch();
}
```

Output:

```
Enter an Integer Value: 50
The Value = 50
```

Explanation:

- ✓ When the first **printf()** executes, it will display the message "Enter an Integer Value:"
- ✓ The execution of **scanf()** function will display the **cursor blinking** at the end of the above message. The blinking **cursor** means the requirement of a value to be typed-in, i.e. an input, which is an integer as the format-specifier is '**%d**'. Suppose, you type the value **50** and now then press the **Enter** ***key*** to ***complete the execution*** of **scanf()**. Thus input value **50** is stored into the variable **a** (***by accessing the address of variable a using '&' operator***).
- ✓ The last **printf()** will now display the message with the value of **a**, i.e. **50**, as in the mentioned in the output.

Program-4: Program which takes two numbers as input and display them.

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

Space for learners notes

```
void main()
{
        int a, b;
        clrscr();
        printf("Enter two numbers: ");
        scanf("%d%d", &a, &b);
        printf("The Values are %d and %d", a, b);
        getch();
}
```

Output:

```
Enter two numbers: 5 100
The Value are 5 and 100
```

Or, the program can also be written as:

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

void main()
{
        int a, b;
        clrscr();
        printf("Enter a number: ");
        scanf("%d", &a);
        printf("Enter another number: ");
        scanf("%d", &b);
        printf("The Values are %d and %d", a, b);
        getch();
}
```

Output:

```
Enter a number: 5
Enter another number: 100
The Values are 5 and 100
```

2.13 OPERATORS

As we understand form the **Unit-Section-2.9.2** that **Operators** are the symbols those are used to perform certain mathematical and logical computations/manipulations. In brief, these are used to manipulate data and data inside variables in a C program. Functionally, operators in C are classified into the following categories:

1. Assignment Operator,
2. Arithmetic Operators,
3. Relational Operators,
4. Logical Operators,
5. Increment and Decrement Operators,
6. Conditional Operators,
7. Bitwise Operators and
8. Special Operators.

Before discussing about the above categories of **operators**, let's first discuss about **Expressions** and the **Assignment Operator** '='.

Space for learners notes

2.13 Assignment Operator:

From earlier discussions, you hopefully understood how to write a C statement. Basically, every statement in C is to be terminated by '**;**' with few exceptions. Those exceptions are, for example

```
#include<stdio.h>
#define max 100
```

Few examples of valid C statements are,

```
int x, y;
printf("Welcome to IDOL");
clrscr();
```

Now consider the following statement,

```
X = 5;
```

The symbol '=' used in the above statement, is known as **Assignment Operator**. This operator is used to store value to a variable, i.e. in programming terms '=' operator is used to **assign value** to **a variable**. Not only for assigning a value, '=' operator is also used for

- assigning value of a variable to another variable,
- assigning the result of an expression to a variable (will be discussed in 2.5.2) and
- assigning the return value form a function call to a variable (will be discussed in Unit-4).

Few examples of using assignment operator are mentioned the ***Table-2.8*** below:

Examples	**Meaning**
A=100	100 is assigned to the variable A (i.e., A now contains 100)
B=A	value of A is assigned to the variable B (i.e., B now also contains 100)

Space for learners notes

C=A+B	values of A and B are added and the total is assigned to the variable C
VAL=sum(10, 20)	the result of function 'sum()' is assigned to variable VAL (will be discussed in Unit: 4: Functions)

Table-2.8: Use of assignment operator

2.13.2 Arithmetic Operators (and Expression):

In the following ***Table-2.9***, the arithmetic operators are listed.

Operators	Meaning
+	Addition
-	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulo (Remainder after division)

Table-2.9: Arithmetic Operators

All of the above operators are used with more than one **Operands**(i.e., data/values); besides the operators '+' and '-' are also used with one operand. An **Operand** can be a variable or a constant. The uses of arithmetic operators are illustrated in the ***Table-2.10***: Suppose, A and B two integer variables and they contain the values 50 and 10 respectively.

Operator	Example	Meaning	Result
+	A + B	Add A with B	60
-	A - B	Subtract B from A	40
*	A * B	Multiply A with B	500
/	A / B	Divide A by B	5
%	A % B	Remainder from A divide by B	0
- (unary)	- A	Multiply A with -1, i.e., will changes A's sign.	-50

Table: 2.10: Use of arithmetic operand

A+B, A-B are known as **Expressions**. An **Expression** can be defined as the sequence of operands and operators that reduces to a single value. Suppose we have to use a mathematical formula "**a+b+2ab**" in a C program. Now, consider the value of **a** is **2** and **b** is **3**. So, in a program, for the above tasks we can write statements:

```
int a, b, res;
a=2;
b=3;
res = a+b+2*a*b;
```

Except the first one, all the other three statements are Assignment Statements. But in the last statement, the left-hand side of = is a variable and right-hand side is the considered mathematical function. Now, here the mathematical formula '**a+b+2*a*b**' is known as expression.

Space for learners notes

2.13.3 Relational Operators:

In a C program, we may have to compare two data and take certain decisions and this can be fulfilled by using Relational Operators. In short Relational Operators are used for comparison of two values/values in variables. ***Table-2.11*** presents the relational operators in C and their meanings.

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

Table-2.11: Relational Operators

Like-wise Arithmetic Expression, a Relational Expression is the expression where a relational operator is used. The form of a relational expression is given below:

op-1* relational-operator *op-2

op-1 and **op-2** are may be arithmetic expressions, variables, constants etc. Relational operators are used in ***decision making/control statements*** such as ***if-else-elseif, do-while, while, for*** etc. You will get acquainted with these kinds of statements in the following ***UNIT-3.***

Operator	Example	Meaning	Result
<	A < B	is A less than B	FALSE
>	A > B	is A greater than B	TRUE
<=	A <= B	is A less than or equal to B	FALSE
>=	A >= B	is A greater than or equal to B	TRUE
= =	A = = B	is A is equal to B	FALSE
!=	A != B	is A is not equal to B	TRUE

Table-2.12: Use of relational operators

Space for learners notes

The uses of relational operators are illustrated in the ***Table-2.12***. Suppose A and B are two integer variables having values 50 and 10 respectively. As you can see that the result of each of the expressions is either TRUE or FALSE, which means that if an expression satisfies what is it written for then it means TRUE. So, in general relational operators are used for forming conditions in a C program using conditional/control statements for making certain decisions making tasks.

2.13.4 Logical Operators:

Logical Operators are operators which are used to combine more than one relational expression to form a larger relational expression in a conditional/control statement.

Consider a situation where, A and B are two variables containing marks of two subjects scored by a student in an examination. Now, we have to check that if the mark of both these subjects (A, B) are greater than 30, then the result will be "Pass".

Now, you may be thinking of relational expression like **A, B > 30**. But, this is incorrect as because in a relational expression only two operands can be used with a relational operator. Thus such situation demands combining more than one relational expression to form a single expression. Here, Logical Operators will play their role for what they are defined in C.

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

Table-2.13: Logical Operators

The ***Table-2.13*** presents the Logical Operators defined in C. The uses of logical operators are illustrated in the ***Table-2.14*** where suppose A contains 28 and B contains 50.

Operator	Example	Meaning	Result
&&	A>30 && B>30	are A B greater than 30	FALSE
||	A>30 || B>30	is A greater than 30 **OR** B greater than 30/both A and B greater than 30	TRUE

Table-2.14: Uses of logical operators

The use of '!' logical operator will be discussed in the following units while working with different programs.

2.13.5 Increment and Decrement Operators:

Space for learners notes

Like Arithmetic Operators, C offers two other operators who does arithmetic operations but in a different style. These two operators are '++' and '—' and they are known as **Increment** and **Decrement Operators** respectively. These operators are illustrated in the following ***Table-2.15.***

Operators	Meaning
++	adds 1 to the value of the associated operand and update it
—	subtracts 1 from the value of the associated operand and update it

Table-2.15: Increment and decrement operators

The uses of these operators are illustrated in the ***Table-2.16*** where suppose both x and y contains 10.

Operator	Example	Meaning	Result
++	x++	adds 1 to the value of x and update x	value of x becomes **11**
	++x		
—	y—	subtracts 1 from the value of y and update y	value of y becomes **9**
	—y		

Table-2.16: Meaning of x++ and ++x/– –x

Basically, the expression 'x++' is same as the expression 'x=x+1'. From the ***Table-2.16*** it is clear that if x contains the value 10, then the expression 'x++' will increment the value of x by 1 i.e., value of x will now be 11. Same as in case of the expression 'y—' but here the value of y will be decremented by 1.

From the above ***Table-2.16***, if the expressions '**x++**' and '**++x**' does the same task then what is the difference between. The differences in these two expressions lies the placing of the operand (A) around '**++**' and also the time of increment.

2.13.5.1 Postfix Increment & Decrement Operations:

Consider the following statement which contains a **Postfix Increment Expression.**

C = X++;

where, C and X are two variables. Suppose X contains the value 10. The above expression combines of two tasks: one is the increment of X by 1 and another is the assignment of the value of the expression '**X++**' to C.

The above example contains the **Postfix Increment** expression '**X++**'. Now, a **Postfix Increment** can be understood by,

- ***in terms of position of the operator ++:*** the operand(X) will be placed before the '++' operator,

Space for learners notes

> ***in terms of execution of the expression:*** first the statement (containing the increment expression) will be executed and then the increment expression(in the same statement) will be evaluated, i.e. the value of the operand(variable X) will be incremented by 1.

Thus, after execution of the above statement '**C=X++**' (where variable C contains 10) the values of:

- ✓ C will be 10 as because the '**X++**' is not evaluated during the execution)
- ✓ X will be 11 as because the '**X++**' is now be evaluated(value of X is incremented by 1).

For **Postfix Decrement**, consider the following statement (where value contained in X is 10):

C = X—;

Thus, after execution of the above statement the values of:

- ✓ C will be 10 as because the '**X—**' is not evaluated during the execution)
- ✓ X will be 9 as because the '**X—**' is now be evaluated and value of X is decremented by 1.

Example-6: Program to demonstrate postfix increment.

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

void main()
{
        int a, result;
        clrscr();
        a=5;
        result=a++;
        printf("Result: %d", result);
        printf("\nValue of a: %d", a);
        getch();
}
```

Output:

```
Result: 5
Value of a: 6
```

Explanation:

- ✓ In the statement, **result=a++**, first value of **a** (i..e. **5**) is assigned to variable **result** and then **a** is incremented by **1** (i.e. postfix increment) and the value of **a** becomes 6.

- ✓ The first **printf()** statement will display the value of the variable **result**, which is 5 along with the set message in the function.
- ✓ The second **printf()** statement will display the value of the variable **a**, which is **6** along with the set message in the function.

Space for learners notes

2.13.5.2 Prefix Increment and Decrement Operations:

Consider the following statement which contains a **Prefix Increment Expression.**

C = ++X;

where C and X are two variables. Suppose X contains the value 10. The above expression combines of two tasks: one is the increment of X by 1 and another is the assignment of the value of the expression **'++X'** to C.

The above example contains the **Prefix Increment** expression **'X++'**. Now, a **Prefix Increment** can be understood by,

- ➢ ***in terms of position of the operator ++:*** the operand(X) will be placed after the '++' operator,
- ➢ ***in terms of execution of the expression:*** first the prefix increment expression will be evaluated and then the statement will be executed, i.e. the incremented value of X (by 1) will be assigned to C.

Thus, after execution of the above statement **'C=++X'**, the values of:

- ✓ X will be 11 as because the **'++X'** is evaluated (value of X is incremented by 1).
- ✓ C will be 11 as because the **'++X'** is evaluated before the execution of the statement completes.

For **Prefix Decrement**, consider the following statement (where value contained in X is 10).

C = —X;

Thus, after execution of the above statement **'C=—X'**, the values of:

- ✓ X will be 9 as because the **'—X'** is evaluated (value of X is decremented by 1).
- ✓ C will be 9 as because the **'—X'** is evaluated before the execution of the statement completes.

Example-7: Program to demonstrate Prefix Decrement operations,

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a, result;
        clrscr();
```

Space for learners notes

```
        a=5;
        result=—a;
        printf("Result: %d", result);
        printf("\nValue of a: %d", a);
        getch();
}
```

Output:

```
Result: 4
Value of a: 4
```

Explanation:

✓ In the statement, **result =—a**, first value of **a** is decremented by **1**(i.e. **4**) and then the value of **a** is assigned to the variable **result** (i.e. **4**).

✓ The first **printf()** statement will display the value of the variable **result**, which is **4** along with the set message in the function.

✓ The second **printf()** statement will display the value of the variable **a**, which is **4** along with the set message in the function.

2.13.6 Conditional Operator:

?: operator is known as **Conditional Operator** used in C. This operator is also called **Ternary Operator**. As we know that operators are generally unary(one operand) and binary(two operands). As ternary means three and so, the '**?:**' operator is termed as ternary Operator just because of association of three operands with this operator while writing an expression. The operands may also be expressions also.

The form of an expression which uses **?:** operator is mentioned below:

exp-1 ? exp2 : exp3

where, **exp-1**, **exp-2** and **exp-3** may be single variables, may be expressions or may be combinations of both. Let's understand the above form with the help of an example. Consider **a**, **b** and **res** are three integer variables. Now, suppose we want to find the maximum value between **a** and **b**. The following statement will do this task, which uses the **?:** operator.

```
res = (a>b) ? a : b;
```

Here, **(a>b)** is the **exp-1**, **a** is the **exp-2** and **b** is the **exp-3**. Now, you may be getting puzzled how this whole expression will be performed! Here is the answer given below:

✓ First, **(a>b)**, the **exp-1** is performed.

✓ Now, if **exp-1** satisfies, i.e. **a** is greater than **b**, then the value of **exp-2** is the result,

✓ But if **exp-1** does not satisfy, i.e. **a** is not greater than **b**, then the value of **exp-3** is the result.

✓ And the result of the expression will be assigned/stored to variable **res**.

And thus we will get the value of the maximum between **a** and **b** in the variable **res**. Now, suppose variable **a** contains the value **10** and variable **b** contains the value **50**. Then, after the execution of the statement value of **b** will be stored in variable **res** as value of **b** is the maximum which is **50**.

Space for learners notes

Example-8: Program to demonstrate the use of conditional operator for finding maximum of two input numbers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a, b, max;
        clrscr();
        printf("Enter the value of a=");
        scanf("%d", &a);
        printf("Enter the value of b=");
        scanf("%d", &b);
        max=(a>b) ? a : b;
        printf("Maximum Value: %d", max);
        getch();
}
```

Output:

```
Enter the value of a=100
Enter the value of b=50
Maximum Value: 100
```

Explanation:

- ✓ Suppose, the two **scanf()** functions take input 100 for **a** and 50 for **b**.
- ✓ In the statement, **max=(a>b) ? a : b**, the condition (a>b) is evaluated and
  - o if condition satisfies, means **a** is the maximum, the value of **a** is assigned to **max**.
  - o if condition not satisfied, means **b** is the maximum, the value of **b** is assigned to **max**.
- ✓ The last **printf()** function will display the value of the variable **max**, which is **100** along with the set message.

2.13.7 Bitwise Operators:

Following ***Table-2.17*** lists the **Bitwise Operators** in C language. These operators are used for bitwise manipulation of data.

Space for learners notes

Operators	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	shift bits left
>>	shift bits right

Table-2.17: Bitwise operators

These operators are my not be applied to float or double type of data.

2.13.8 Special Operators

Apart from the operators discussed above, there are other operators in C and they are

sizeof Operator

Type-Cast Operator: **(type)**

Pointer Operators: **&** and *

Member Selection Operators: **.** and →

The **sizeof** operator is used to get the number of bytes occupied by a operand/type. For example, **A** and **S** are integer variables. Consider the C statement mentioned below:

```
S = sizeof(A);
```

Here in the above example the **sizeof** operator will return the value **2** as **A** is of **int** (integer) type and the size of **int** data-type is **2 bytes**. So, value in **S** is **2** after execution of the above statement.

Now, what is **Type-Cast Operator, (type)**? Basically this operator is used to convert the type of a data to another compatible type temporarily.

To understand this operator let's first consider the following statements.

```
int a=7;
float res;
res = a/2;
```

So, the variable **a** is assigned with **7**. After the execution of these statements, you may think that the value in **res** will be **3.5**. But in practice it will be **3**. Since; we are dividing **a** by **2** where **a** is an integer and therefore instead of getting **3.5** we will get **3** though the variable **res** is of type float.

But we are expecting that the last statement would assign **3.5** to the variable **res**. So, how to get **3.5** as a result of the above expression?? The use of **Type-Cast Operator** will give us our expected result, i.e. **3.5**. So, we have to replace the expression in the last statement with the expression '**(float) a / 2**' and so the last statement to be written as,

res = (float)a/2;

This means that the value of **a** (i.e. 7) is temporarily converted to float type (i.e. **7.0**) without effecting **a** and then the converted value (i.e. **7.0**) is divided by **2**. Thus the result **3.5**, will be assigned to **res**.

The use of **Pointer Operator *** and **Member Selection Operators . ** and **→** will be discussed in the **Units 6** and 7.

The **Pointer Operator &** which is known as **Address-of Operator** is used to get the ***address of a location*** (discussed in section 2.5).

Example-9: Program to demonstrate the use of the **sizeof()** and **(type)** type-cast-operator. Here in the program the three integer variables sub1, sub2 and sub3 are for storing the marks of three subjects.

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

void main()
{
        int sub1, sub2, sub3;
        float average;
        int size;
        clrscr();
        printf("Enter marks for subject-1, subject-2 and subject-3:");
        scanf("%d%d%d", &sub1, &sub2, &sub3);
        average=(float)(sub1+sub2+sub3)/3;
        printf("The average mark is: %f", average);
        size=sizeof(int);
        printf("\nThe size of int data-type is: %d bytes", size);
        getch();
}
```

Output:

```
Enter marks for subject-1, subject-2 and subject-3: 50 65 45
The average mark is: 53.3333
The size of int data-type is: 2 bytes
```

Explanation:

- ✓ The **scanf()** statement take three inputs, suppose **50**, **65** and **45** for **sub1**, **sub2** and **sub3** respectively.
- ✓ In the statement next to **scanf()** the average of the three marks, taken as inputs(**50**, **65** and **45**), is calculated. The average may definitely in the form of a ***floating point number***. But, all the three variables containing marks are of

Space for learners notes

Space for learners notes

data type **int**. The expression "**(sub1+sub2+sub3)/3**" will give an integer but not a ***floating point number***. So, to get a ***floating point number*** the result of the above expression is temporarily converted into float using the **(float)** operator. Now, the result of the expression "**(float)(sub1+sub2+sub3)**" becomes a **float** value and this is divided by **3** resulting a **float** value and it is assigned to the float variable **average**. Thus the variable average contains the value **53.3333**.

✓ In the statement,

```
size=sizeof(int);
```

the **sizeof()** operator will give the size of **int** data type in bytes and this value(i.e. **2**) is assigned to the variable **size**.

2.14 OPERATOR PRECEDENCE AND ASSOCIATIVITY

When an expression contains more than one operator then the concept of **Operator Precedence** applies. **Operator Precedence** can be defined as the rule for determination of which operator to be performed first, which to be 2nd and so on in case of an expression with more than one operator.

Consider the following statement, where a=2, b=5 and c=3.

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

Now, you can think of how the expression part (right-hand side of =) will be evaluated. Here the evaluation may take place in two possible ways:

WAY-1:

- At first the values of **a** and **b** will be added and then
- the total value will be multiplied by the value in **c**.

i.e. **x = (2 + 5) * 3**
= 7 * 3
= 21

WAY-2:

- At first value of **b** is multiplied with the value of **c** and then
- the total value is added with the value in **a**.

i.e. **x = 2 + (5 * 3)**
= 2 + 15
= 17

We know that **WAY-2** is the actual way of evaluating this expression as according to mathematics, first multiplication(*) operation will take place and then the addition(+). Thus the value in **x** will be **17**. This is an example of application of **Operator Precedence**.

Space for learners notes

In case of Arithmetic Operators, there are two distinct levels of priority in C and they are:

High Priority Operators: *** / %** (same precedence)

Low Priority Operators: **+ -** (same precedence)

STOP TO CONSIDER

While writing a mathematical expression that contains more than one operator with different precedence (or with same precedence), use brackets **( )** to specify the evaluation more precisely. Consider the expression **a+b*c** and suppose you want the evaluation as **a+b** and then multiply with **c**, so to be precise write the expression as **(a+b)*c**.

Now, let's try to understand what does **Associativity** mean? **Associativity** also can be defined as the rule which needs to be applied for evaluation when an expression contains more than one operator with the same precedence. Associativity can be either **Left-to-Right** or **Right-to-Left**.

The Associativity of Arithmetic Operators with same precedence is **Left-to-Right**. We know that the operators, + and –, have the same precedence. Now, let's see how the arithmetic expression in the following C statement will be evaluated.

X = 10 + 2 - 3

Here in the above example, the evaluation of the expression '**10 + 2 - 3**' will start from **Left-to-Right**(because of **associativity**). So, the evaluation will be in the form mentioned below:

(10 + 2) – 3

i.e.

- first evaluation of '**10+2**' will take place, then
- the value **3** will be subtracted from the result value of '**10+2**'.

So, after execution of the above statement, the variable X will contain the result of the evaluation which is **9**.

The following **Table-2.14** lists the **Precedence** and **Associativity** of the Operators present in C.

Operator	**Description**	**Associativity**
( )	Parentheses	
[ ]	Brackets (related to Array)	
.	Member Selection (using Object Name)	left-to-right
->	Member Selection (using Pointer)	
++ —	Postfix Increment/Decrement	
++ —	Prefix Increment/Decrement	
+ -	Unary Plus/Minus	
! ~	Logical negation/bitwise complement	

Space for learners notes

(*type*) * & sizeof	Cast (convert value to temporary value of *type*) Dereference (related to Pointer) Address of Oper and For Size in Bytes	right-to-left
* / %	Multiplication/Division/Modulus	left-to-right
+ -	Binary Addition/Subtraction	left-to-right
<< >>	Bitwise Left-Shift, Bitwise Right-Shift	left-to-right
< <= > >=	Relational is Less Than/Less Than or Equal To Relational is Greater Than/Greater Than or Equal To	left-to-right
== !=	Relational is Equal To/is Not Equal To	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise Exclusive OR	left-to-right
|	Bitwise OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary (Conditional)	right-to-left
=	Assignment	right-to-left
,	Comma (for separation of expressions)	left-to-right

Table-2.14: Operator Precedence and Associativity

CHECK YOUR PROGRESS:

1. What do you understand by Data Type? Mention its different categories.
2. Mention the fundamental data types in C with their respective size.
3. What is typedef?
4. What is variable? Write down the syntax for declaring a variable.
5. Mention the use of **scanf()** function in C.
6. Define the terms: Operator Precedence and Associativity.
7. The size of the char data type is ____________.
8. Structure is an example of ________________ data type.
9. The format specifier for integer (**int**) type is ______.
10. % arithmetic operator is used to ______________________.

State True or False:

11. sizeof() operator gives the size of a given type.
12. – (minus) operator can be used as binary and unary operators.
13. If **a=10** and **b=20**, then **a>b** will be evaluated as **TRUE**.

14. If **x=110**, then the output of the following statement is **111**.

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

15. In postfix increment operation expression, the ++ operator is placed before the operand.

Space for learners notes

2.15 SUMMING UP

In this Unit, the history of C language is briefly described. The features of C are listed more elaborately including the most specific feature as being a middle-level language which combines the capabilities of a low-level language (i.e. assembly language) along with the features of a high-level language.

Different steps starting from writing a C program to its execution is described in this Unit. These steps are described here in a very well-organized manner using different screen-shots. The steps shown are not only for Windows operating system but for UNIX operating system also. The structure of C program is also described in detail.

This Unit describes the errors generally occur during compilation and execution of a C program with the help of examples. The errors described here are namely Syntax Errors, Semantic Errors and Run-time Errors.

Here, in this Unit the list of C character set is also given with different characters, numbers and symbols with their names.

A C program consists of tokens which can be categorically classified into namely: Keywords, Identifiers, Constants, Operators and Special Characters. All these different types of C tokens are very elaborately described in this Unit.

The concept of data type is described here in this Unit. There are three categories of data type and they are: Primary/Built-in (e.g., char, int), Derived (e.g., Arrary) and User Defined (e.g., Structure). The size of each of the type is clearly discussed with different examples.

A variable as discussed in this Unit, is a storage space into which one can store data. It needs to be declared before its use. In declaration statement, the data type of the variable should be mentioned. As C language is case sensitive, the case in which a variable is declared should remain the same during its use in a program.

Input and Output are basic requirements of a C program. The 'printf()' function is used to display data/message on to the screen. The 'scanf()' function is used to take input from keyboard. The syntaxes of these two functions are described in this Unit.

There are different types of operators defined in C. Few operators are necessary for arithmetic expressions while few are useful for conditional expressions and so on. The operators have precedence and associativity. This Unit gave a detailed description regarding the operators' precedence and associativity.

Space for learners notes

2.16 ANSWERS TO CHECK YOUR PROGRESS-1

1. C language was developed in the early 1970s by Dennis M. Ritchie who was an employee from AT&T(Bell Labs).
2. C Language is also known as middle-level language as its compiler has the capabilities of a low-level language (i.e. assembly language) along with the features of a high-level language. This is the reason that C language is well suited for writing both System Software and Application Software.
3. Like Natural Languages, C language has its library(dictionary) and it consists of some pre-written files. The pre-written files are known as header files. For example stdio.h, conio.h are header files.
4. Errors are the consequences of some mistakes while writing a C program. The errors are displayed while compiling or running a C program.
5. Keywords are the reserved words that have well-defined purposes. A keyword should be used only for its particular use and not for any other purposes like naming a variable or function.

   The four examples of keyword are: auto, int, struct, while.
6. Identifiers are basically composed of alphanumeric characters i.e. alphabets or digits. The basic rules for naming an identifier are:
   - The first character should be an alphabet or an underscore.
   - No special symbols except the underscore, is allowed in an Identifier.
   - Keywords cannot be used as Indentifiers.

   The identifiers could be of any length but only the first 31 characters are significant.
7. Integrated Development Environment (IDE)
8. American Standard Code for Information Interchange (ASCII)
9. Runtime error

2.17 ANSWERS TO CHECK YOUR PROGRESS-2

1. Data type can be defined as the type of data that can be used in a program. C also allows creation of new Data Types and also customization/enhancement of offered Data Types as per need of a program. Data Types in C can be broadly classified in three classes and they are:
   - Primary/Built-in Data Type,
   - Derived Data Type and
   - User-Defined Data Type.
2. Basically, C supports five fundamental types and they are char, int, float, double and void. Their sizes are given in the following table:

Data Type	Size (in bytes)
char	1
int	2
float	4
double	8

3. The typedef is a keyword and it is used to assign an *alias* i.e. *alternative name* to an existing fundamental/derived/user defined data type. The syntax for using typedef is:

    typedef existing-type-name alternative-name;

4. A variable is a named memory location (in main memory) where one can store different values (of a particular *type*) at different times. The syntax for declaring a variable is:

    data-type variable-name;

5. scanf() is the function to take input during the execution of the program. The syntax of scanf() is,

    scanf(Formatted-String, &Variable-1, &Variable-2, ...);

6. Operator Precedence can be defined as the rule for determination of which operator to be performed first, which to be 2nd and so on in case of an expression with more than one operator.

    Associativity also can be defined as the rule which needs to be applied for evaluation when an expression contains more than one operator with the same precedence. Associativity can be either Left-to-Right or Right-to-Left.

7. 1 byte.
8. User Defined Data Type
9. %d
10. calculate the remainder in a division operation
11. True.
12. True.
13. False.
14. False.
15. False.

2.18 POSSIBLE QUESTIONS

Short answer type questions:

1. Mention the structure of a C program.
2. What is **main()**?
3. What does runtime error mean? Explain briefly.

Space for learners notes

Space for learners notes

4. What is data type modifier? Explain.
5. What is variable? Write down the rules for naming a variable.

Long answer type questions:

1. With the help of an example explain the structure of a C program.
2. What do you understand by signed and unsigned? Explain with the help of examples.
3. Write a C program which takes two integers as input and display their summation.
4. With the help of an example discuss the use of **typedef**.
5. Discuss the **scanf()** function in C.
6. Write a C program to demonstrate the difference between Postfix Increment and Prefix Increment.
7. What are the consequences of evaluation of an expression with multiple operators with different precedence? Explain with the help of examples.

2.19 FURTHER READINGS

1. Kernighan, B. W., & Ritchie, D. M. (2006). *The C programming language*, PHI.
2. Balagurusamy, E. (2012). *programming in ANSI C*. Tata McGraw-Hill Education.
3. Kanetkar, Y. P. (2016). *Let us C*. BPB publications.
4. Schildt, H., & Turbo, C. (1992). C: The Complete Reference. *McGraw-Hill, Inc., New York, NY, USA*, *4*, 39.

UNIT 3 CONTROL STATEMENTS, DECISION CONTROL STATEMENTS

Space for learners notes

CONTENTS

3.1 INTRODUCTION

In C, all statements written in a program are executed from top to bottom one by one. In some cases, there may arise some situations where depending upon a logical condition, some actions have to be carried out. Control statements are used to execute/transfer the control from one part of the program to another depending on a condition. These statements are also called conditional statements.

Control statements are of the following two types:

- Conditional control
- Loop control

C has three major conditional control statements:

(a) **if** statement (b) **if-else** statement and (c) **switch** statement

On the other hand, in a program, there may arise some situations where a repetitive work has to be carried out until a specific condition is fulfilled. In that case, ***loop*** control statements are used in a program. C has three loop control statements: (a) for (b) while and (c) do while

This unit introduces you the different conditional and loop control statements with some suitable examples.

3.2 OBJECTIVES

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

- know the functions and use of different decision control statements,
- work with different loop control statements in a program and
- use goto, break and continue statement in a program.

3.3 CONDITIONALSTATEMENT

Conditional statements are used to execute statement or group of statements based on some condition.

C supports following conditional statements.

1. if statement,
2. if else statement,
3. if else if ladder and
4. nested if.

3.3.1 The *if* statement

The *if* statement is a control statement that tests a particular condition. Whenever, the evaluated condition comes out to be true, then that action or the set of actions are carried out. Otherwise, the given sets of action(s) are ignored.

The *syntax* of ***if*** statement is:

```
if(condition) {
 //statement(s) will execute if the condition is true
}
```

Example 3.1 Write a program to display the message "you have entered a +ve number" if the user entered a +ve number".

Space for learners notes

Solution:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int number;
clrscr();
printf("Enter an integer:\t");
scanf("%d", &number);
if (number > 0)
  {
          printf("You have entered a +ve number.");
  }
printf("\n The number you have entered is %d", number);
getch();
}
```

When the above code is compiled and executed, it produces the following result:

Output 1:

```
Enter a number: 10
You have entered a +ve number.
The number you have entered is 10
```

Output 2:

```
Enter a number:  -5
The number you have entered is -5
```

Example 3.2: Write a program to find the biggest of two numbers

Solution:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a, b, big;
clrscr();
printf("Enter two numbers:\t");
```

Space for learners notes

```
scanf("%d %d", &a, &b);
big = a;
if(b>big)
big = b;
printf("\n The biggest number is: %d",big);
getch();
}
```

Output:

```
Enter two numbers: 10 20
The biggest numbers is: 20
```

Example 3.3: Write a program to find the biggest of three numbers.

Solution:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a,b,c, big;
clrscr();
printf("Enter three numbers:\t");
scanf("%d %d %d", &a, &b, &c);
big = a;
if(b > big)
          big = b;
if(c > big)
          big = c;
printf("\n The biggest number is: %d", big);
getch();
}
```

Output:

```
Enter two numbers: 10 25 9
The biggest numbers is: 25
```

STOP TO CONSIDER

If more than one statements has to be executed in an ***If*** statement, you should write those statements within { and }

3.3.2 The *if else* statement

Space for learners notes

If else statement is used to execute a statement block or a single statement depending on the value of a condition.

The syntax of ***if else*** statement is:

```
if(condition) {
 /* statement(s) will execute if the condition is true */
}
else {
/* statement(s) will execute if the condition is false */
}
```

If the condition evaluates to **true**, then the statement(s) inside the if block will be executed, otherwise, the statement(s) inside the else block will be executed.

Example 3.4 Write a program to find the biggest of two numbers.

Solution:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a,b,big;
clrscr();
printf("Enter two numbers:\t");
scanf("%d %d", &a, &b);
if (a>b)
        printf("\n The biggest number is: %d", a);
else
        printf("\n The biggest number is: %d", b);
getch();
}
```

Output:

```
Enter two numbers: 10 20
The biggest numbers is 20
```

Explanation:

Here in this case, 10 will be assigned to the variable a and 20 will be assigned to the variable b.

Then the statement if(a>b) will be tested. Since it is false (as 10 <20), so the statement under the else part will execute.

Space for learners notes

Example 3.5: Write a program to check whether a number entered by the user is even or odd.

Solution:

```
#include <stdio.h>
#include <conio.h>
void main(){
int n;
clrscr();
printf("Enter a number you want to check: \t");
scanf("%d",&n);
if((n%2)==0) /* Checking whether remainder is 0 or not. */
printf("The number %d is even.",n);
else
printf("The number %d is odd.",n);
getch();
}
```

Output 1:

```
Enter an integer you want to check: 11
11 is odd.
```

Output 2:

```
Enter an integer you want to check: 14
14 is even.
```

Example 3.6: Write a program to check whether a character entered by the user is vowel or consonant.

Solution:

```
#include <stdio.h>
#include <conio.h>
void main(){
char c;
clrscr();
printf("Enter a character: \t");
scanf("%c",&c);
if(c=='a'||c=='A'||c=='e'||c=='E'||c=='i'||c=='I'||c=='o'||c=='O'||c=='u'||c=='U')
printf("%c is a vowel.",c);
else
printf("%c is a consonant.",c);
getch();
}
```

Output 1:

```
Enter an alphabet: e
e is a vowel.
```

Output 2:

```
Enter an alphabet: x
x is a consonant.
```

Explanation: In this program, user is asked to enter a character which is stored in variable *c*. Then, this character is checked, whether it is any one of these ten characters a, A, e, E, i, I, o, O, u and U using logical OR operator ||. If that character is any one of these ten characters, that alphabet is a vowel; if not then that alphabet is a consonant.

Space for learners notes

Example 3.7: Write a program to check whether a character is alphabet or not.

Solution:

```
#include <stdio.h>
#include <conio.h>
void main()
{
char c;
clrscr();
printf("Enter a character: \n");
scanf("%c",&c);
if( (c>='a' && c<='z') || (c>='A' && c<='Z'))
printf("%c is an alphabet.",c);
else
printf("%c is not an alphabet.",c);
getch();
}
```

Output 1:

```
Enter a character: g
g is an alphabet
```

Output 2:

```
Enter a character: #
# is not an alphabet
```

Space for learners notes

Explanation: When a character is stored in a variable, ASCII value of that character is stored instead of that character itself. For example: If 'g' is stored in a variable, ASCII value of 'g' which is 103 is stored. If you see the ASCII table, the lowercase alphabets are from 97 to 122 and uppercase letters are from 65 to 90. If the ASCII value of number stored is between any of these two intervals then that character will be an alphabet. In this program, instead of number 97, 122, 65 and 90, we have used 'a', 'z', 'A' and 'Z' respectively which are part of the same thing.

3.3.3 Multiple *if else* statement

An **if** statement can be followed by number of **else if else** statements, which is very useful to test various conditions.

The *syntax* of ***if else if else*** statement is

```
if(condition 1) {
  /* Statement(s) to be executed only when condition 1 is true */
}
else if(condition 2) {
/* Statement(s) to be executed only when condition 2 is true */
}
else if(condition 3) {
/* Statement(s) to be executed only when condition 3 is true */
}
else {
  /* Statement(s) to be executed only when none of the above conditions are true */
}
```

Example 3.8: Write a program to check whether a character entered by the user is uppercase or lowercase.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main (){
char a;
clrscr();
printf("Enter an alphabetic character: \t");
scanf("%c", &a);
if (a > 64 && a <=91)
        printf("The character is an upper-case letter.");
else if (a > 96 && a <=123)
        printf("The character is alower-case letter.");
else
        printf("This is not an alphabetic character.");
getch();
}
```

Space for learners notes

Output 1:

```
Enter an alphabetic character: a
The character is a lower-case letter
```

Output 2:

```
Enter an alphabetic character: B
The character is a upper-case letter
```

Output 3:

```
Enter an alphabetic character: 10
This is not an alphabetic character
```

Example 3.9: The marks obtained by a student in 5 different subjects are input through the keyboard. The student gets a division as per the following rules:

(i) Percentage above or equal to 85 - Distinction

(ii) Percentage above or equal to 75 - Star

(iii) Percentage above or equal to 60 - First division

(iv) Percentage between 50 and 59 - Second division

(v) Percentage between 30 and 49 - Third division

(vi) Percentage less than 30 - Fail

Write a program to calculate the division obtained by a student.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int m1, m2, m3, m4, m5, per ;
clrscr();
printf ( "Enter marks obtained by a student in five subjects: \t" ) ;
scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
per = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;
if ( per >= 85 )
```

Space for learners notes

```
        printf("The result is: Distinction");
else if ( ( per >= 75 ) && ( per < 85 ) )
        printf( " The result is: Star" ) ;
else if ( ( per >= 60 ) && ( per < 75 ) )
        printf( " The result is: First division" ) ;
else if ( ( per >= 50 ) && ( per < 60 ) )
        printf( " The result is: Second division" ) ;
else if ( ( per >= 30 ) && ( per < 50 ) )
        printf( " The result is: Third division" ) ;
else
        printf( " The result is: Fail" ) ;
getch();
}
```

Example 3.10: Write a program to display the name of the day in a week depending upon the number entered by the user.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int day ;
clrscr();
printf ( "Enter a number between 1 and 7: \t " ) ;
scanf ( "%d ", &day ) ;
if ( day ==1 )
        printf ( "The day is Monday." ) ;
else if ( day ==2 )
        printf ( "The day is Tuesday." ) ;
else if ( day ==3 )
        printf ( "The day is Wednesday." ) ;
else if ( day ==4 )
        printf ( "The day is Thursday." ) ;
else if ( day ==5 )
        printf ( "The day is Friday." ) ;
else if ( day ==6 )
        printf ( "The day is Saturday." ) ;
else
```

```
        printf( "The day is Sunday." );
getch();
}
```

3.3.4 Nested *if else* statement

An if statement may have another ***if*** statement in the true condition block and false condition block. This compound statement is called ***nested if*** statement. There may be any number of ***if*** statements in the nested form.

The *syntax* of ***nested if*** statement is:

```
if( condition 1)
        if( condition 2)
        {
                <true block 1>
        }
        else
        {
                <false block 1>
        }
else
        if( condition 3)
        {
                <true block 2>
        }
        else
        {
                <false block 2>
        }
```

Example 3.11: Write a program to find the biggest of any three numbers entered by the user using nested if else statement.

Solution:

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

void main()
{
int a, b, c, big ;
```

Space for learners notes

Space for learners notes

```
clrscr();
printf("Enter the first number: \t");
scanf("%d",&a);
printf("\nEnter the second number: \t");
scanf("%d", &b);
printf("\nEnter the third number: \t");
scanf("%d", &c);
if (a > b)
if (a > c)
                big = a;
        else
                big = c;
else
if (b > c)
                big = b;
        else
                big = c;
printf("The biggest number is %d", big);
getch();
}
```

Example 3.12: Write a program to check whether a year is leap year or not.

Solution:

```
#include <stdio.h>
#include <conio.h>
void main(){
int year;
clrscr();
printf("Enter a year: \t");
scanf("%d",&year);
if(year%4 == 0)
    {
        if( year%100 == 0) // Checking for a century year
      {
        if ( year%400 == 0)
                printf("%d is a leap year.", year);
        else
                printf("%d is not a leap year.", year);
        }
else
printf("%d is a leap year.", year );
}
else
printf("%d is not a leap year.", year);
getch();
}
```

Output 1:

```
Enter year: 1900
1900 is not a leap year.
```

Output 2:

```
Enter year: 2012
2012 is a leap year.
```

CHECK YOUR PROGRESS 1

1. Differentiate between ***if*** and ***if else*** statement.

----

2. What is nested if else statement ?

----

3. What will be the output of the following C code?

```
void main()
{
        int a=20,b,c;
        if (a>35)
        {
          b = 10;
          c = 6;
        }
        else
```

Space for learners notes

Space for learners notes

```
            {
              b = 20;
              c = 5;
            }
      printf("The value of b=%d and c=%d", b,c);
      }
```

3.3.5 The *switch* statement

Instead of using the *if else if* statement, the *switch* statement can be used. The *switch* statement is used to execute a block of statements depending on the value of a variable or an expression. The syntax of the *switch* statement is as follows:

```
switch(<expression>) {
case<label 1>:
          statement(s);
          break;
case<label 2>:
          statement(s);
          break;
case<label 3>:
          statement(s);
          break;
   // you can have any number of case statements
default :
          statement(s);
          break;
}
```

Let us discuss about the above syntax of the switch statement.

- The control statement *switch* begins with the switch keyword followed by one block which contains different cases. Each case handles the statements corresponding to an option i.e. <label 1>, <label 2> etc. (a satisfied condition) and ends with the *break* statement which transfers the control out of the *switch* structure to the original program.
- The compiler checks the values of the expression or variable. If this value matches with any one of the labels given in <case> value, then that statement block will be executed.

- The braces { } can be omitted when there is only one statement available in the statement block.
- Here the variable between the parentheses following the *switch* keyword is used to test the condition and is called the control variable.
- *Break* is a statement which will transfer the control to the end of *switch* statement. (The details about the *break* statement will be discussed in the next section)
- The *default* block is optional i.e. this may be omitted while writing a program.

STOP TO CONSIDER

The control variable of the switch statement can only be of the type int, long or char. ***Switch*** statement is compact and can be used to replace a nested ***if*** statement.

Example 3.13: Using switch statement write a program to display the day of a week. When user types 1, Monday should be displayed, for 2 Tuesday….etc.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int choice;
        clrscr();
        printf("Enter your choice between 1 and 7:\t");
        scanf("%d", &choice);
        switch (choice)
        {
                case 1:
                                printf("Monday");
                                break;
                case 2:         printf("Tuesday");
                                break;
                case 3:
                                printf("Wednesday");
                                break;
                case 4:
                                printf("Thursday");
                                break;
```

Space for learners notes

Space for learners notes

```
                case 5:
                                printf("Friday");
                                break;
                case 6:
                                printf("Saturday");
                                break;
                case 7:
                                printf("Sunday");
                                break;
                default:
                        printf("Invalid choice. Enter your choice between 1 and 7");
                        break;
        }
        getch();
}
```

Example 3.14: Write a program which will read two numbers from the user. Now perform the addition, subtraction, multiplication and division operations according to the need of the user.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,choice;
        clrscr();
        printf("Enter two numbers:\t");
        scanf("%d%d",&a,&b);
        printf("\nEnter your choice between 1 and 4:\t");
        printf("\n[1]. Addition\n");
        printf("[2]. Subtraction\n");
        printf("[3]. Multiplication\n");
        printf("[4]. Division\n");
        scanf("%d", &choice);

switch (choice)
{
```

```
case 1:
        printf("The addition of %d and %d is: %d",a,b,a+b);
        break;

case 2: printf("The Subtraction of %d and %d is: %d",a,b,a-b);
        break;
case 3:
        printf("The Multiplication of %d and %d is: %d",a,b,a*b);
        break;
case 4:
        printf("The Division of %d and %d is: %d",a,b,a/b);
        break;
default:
        printf("Invalid choice. Enter your choice between 1 and 4");
        break;
}
getch();
}
```

STOP TO CONSIDER

The ***default*** block is executed when none of the case labels matches with the value of the expression/variable

CHECK YOUR PROGRESS 2

4. State whether the following statements are true or false:
   a) Every if statement can be converted into an equivalent switch statement.
   b) 'default' case is mandatory in a switch statement.
   c) An if statement may contain compound statements only in the else clause.
   d) A break statement must be used following the statements for each case in a switch statement.
   e) The control variable of the switch could be of type int, long or char.

3.4 LOOP CONTROL STATEMENT

In programming, there may arise some situations where a repetitive work has to be carried out. For example, suppose we want to display the sentence "IDOL, Gauhati University" 100 times. What will we do ? We can display the sentence by writing 100 printf() statements. But, it will be a time consuming process. Similarly, suppose we want to display the numbers between 1 and 1000. So far we have learnt only one

Space for learners notes

Space for learners notes

solution, displaying the numbers with 1000 printf() statements. But, this is not a solution. The solution is, we have to use loop.

Loop control structures are used to execute and repeat a block of statements depending the value of a condition. This condition causes the termination of the loop. In C, there are three types of loop control statements.

- for loop
- while loop
- do while loop

In the following sections, we will discuss the use of these three loops with some examples.

3.4.1 The *for* loop

The *syntax* of **for** *loop* is:

```
        for (expression-1; expression-2; expression-3)
        {
                statement 1;        ⎫
                statement 2;        ⎪
                ..................  ⎬  body of the loop
                ..................  ⎪
                statement n;        ⎭
        }
```

- **expression-1** contain *initialization*statement(s).

  The *statement(s)* are *assignment statement(s)* used to set the *loop* control *variable/variables*. These *statement/statements* will *execute* before the first *iteration* of the *loop*.
- **expression-2** contain *condition(s)*.

  The *condition(s)*, that determine the termination of the *loop*. Before every *iteration*, the *condition(s)* is/are *checked* and if found *true* then the next *iteration* will take place.
- **expression-3** contain *increment* or *decrement statement(s)*.

  The *statement/statements* denote how to change the states of the control *variable(s)* after each *iteration*.
- the section within "{" and "}" is called as the *body* of the *loop*.

Example 3.15: Write a C program to display all the numbers between 1 and 100.

Solution:

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

```
void main()
{
        int i;
        clrscr();
        printf("The nos. between 1 and 100 are: \n");
        for(i=1; i<=100; i++)
        {
                printf("%d\t", i);
        }
        getch();
}
```

Explanation:

In the "for" *loop*,

- in **expression-1**, the *variable* **"i"** is initialized to 1 as the starting no. of the range is 1.
- in **expression-2**, the *condition* is set as "i<=100" because the *loop* will continue as long as the value of "i" remains less than or equal to 100.
- in **expression-3**, the value of "i" is incremented by 1 after each *iteration*.
- in the *body* of the *loop*, the "printf()" will display the current value of "i" with a space in between.

Thus the numbers between 1 and 100 will be displayed.

Example 3.16: Write a C program to display all the numbers between 1 and 100 those are divisible by 3.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
clrscr();
printf("The numbers, divisible by 3, between 1 and 100 are: \t");
for(i=1; i<=100; i++)
{
        if((i%3)==0)
        {
                printf("%d \t", i);
        }
```

Space for learners notes

Space for learners notes

```
}
getch();
}
```

Explanation:

In the "for" *loop*,

- in **expression-1**, the *variable* "i" is initialized to 1 as the starting no. of the range is 1.
- in **expression-2**, the *condition* is set as "i<=100" because the *loop* will continue as long as the value of "i" remains less than or equal to 100.
- in **expression-3**, the value of "i" is incremented by1 after each *iteration*.

In the *body* of the *loop*,

- the remainder of the division, "i" by 3 is calculated and compared equality to 0 using "if" statement.
- and if the condition satisfies, i.e. the current value of "i" is divisible by 3, then "printf()" will display the current value of "i" with a space in between. (\t means a space with tab)

Thus the numbers, those are divisible by 3, between 1 and 100 will be displayed.

Example 3.17: Write a program to find the sum of first *n* natural numbers where n is entered by user. (Note: 1,2,3... are called natural numbers.)

Solution:

```
#include <stdio.h>
#include<conio.h>
void main()
{
int n, i, sum=0;
clrscr();
printf("Enter the value of n: \t");
scanf("%d",&n);
for(i=1; i<=n; i++) //for loop terminates if i>n
{
sum=sum + i;
}
printf("The summation of the numbers between 1 and %d is: %d", , n, sum);
getch();
}
```

Output:

```
Enter the value of n: 19
The summation of the numbers between 1 and 19 is: 190
```

Explanation: In this program, the user is asked to enter the value of *n*. Suppose you entered 19 then; i is initialized to 1 at first. Then, the test expression in the for loop, i.e., (i<= n) becomes true. So, the code in the body of for loop is executed which makes *sum* to 1. Then, the expression i++ is executed and again the test expression is checked, which becomes true. Again, the body of for loop is executed which makes *sum* to 3 and this process continues. When count is 20, the test condition becomes false and the for loop is terminated.

Example 3.18: Write a program to display all the even and odd numbers between 10 and 100. Also display the summation of all the even and odd numbers separately within that range.

Solution:

```
#include <stdio.h>
#include<conio.h>
void main()
{
int i, sum=0;
clrscr();
for(i=10; i<=100; i++)
{
if (i%2==0)
printf("\nThe even numbers between 10 and 100 are: %d\t", i);
sum=sum + i;
}
printf("\nThe summation of all the even numbers between 1 and 100 is: %d", sum);
/* sum is again initialised to 0 to calculate the summation of all odd numbers between
1 and 100 */
sum = 0;
for(i=10; i<=100; i++)
{
if (i%2!=0)
printf("\nThe odd numbers between 10 and 100 are: %d\t", i);
sum=sum + i;
}
printf("\nThe summation of all the odd numbers between 1 and 100 is: %d", sum);
getch();
}
```

Space for learners notes

Space for learners notes

Example 3.19: Write a program to display all the even and odd numbers between a range of numbers where the starting number and the ending number of that range will be provided by the user. Also display the summation of all the even and odd numbers separately within that range.

Solution:

This program is same as the Example 3.18. Here the only modification is that the starting number and ending number of the *for loop* is not fixed. Those two numbers of the range will be inputted by the user. Let us solve the problem as below:

```
#include <stdio.h>
#include<conio.h>
void main()
{
int start_no, end_no, i, sum=0;
clrscr();
for(i=start_no; i<=end_no; i++)
{
if (i%2==0)
printf("\nThe even numbers between %d and %d are: %d\t", start_no, end_no, i);
sum=sum + i;
}
printf("\nThe summation of all the even numbers between %d and %d is: %d", start_no,
end_no, sum);
/* sum is again initialised to 0 to calculate the summation of all odd numbers between
starting number and ending number of the range */
sum = 0;
for(i=start_no; i<=end_no; i++)
{
if (i%2!=0)
printf("\nThe odd numbers between %d and %d are: %d\t", start_no, end_no, i);
sum=sum + i;
}
printf("\nThe summation of all the odd numbers between %d and %d is: %d", start_no,
end_no, sum);
getch();
}
```

Example 3.20: Write a program to find the factorial of a number.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n, i;
long int fact;
printf("Enter an number: \t");
scanf("%d",&n);
if(n==0)
        printf("\nFactorial of 0 is 1\n");
else
{
        fact = 1;
        for(i=1; i<n; i++)
        fact = fact * i;
        printf("The factorial of %d is %d:", n, fact);
}
getch();
}
```

Output:

```
Enter an integer: 3
Factorial = 6
```

Explanation: For any positive number *n*, its factorial is calculated as factorial = 1*2*3*4...*n

Example 3.21: Write a program to check whether a given number is prime or not.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n, i, flag=0;
clrscr();
printf("Enter a positive integer: ");
scanf("%d",&n);
for(i=2;i<=n/2;++i)
```

Space for learners notes

Space for learners notes

```
{
if(n%i==0)
{
flag=1;
break;
}
}
if(flag==0)
printf("%d is a prime number.",n);
else
printf("%d is not a prime number.",n);
getch();
}
```

Output

```
Enter a positive integer: 5
5 is a prime number.
```

Explanation: This program takes a positive integer from user and stores it in variable *n*. Then, for loop is executed which checks whether the number entered by user is perfectly divisible by *i* or not starting with initial value of *i* equals to 2 and increasing the value of *i* in each iteration. If the number entered by user is perfectly divisible by *i* then, *flag* is set to 1 and that number will not be a prime number but, if the number is not perfectly divisible by *i* until test condition i<=n/2 is true means, it is only divisible by 1 and that number itself and that number is a prime number.

3.4.2 The *while* loop

The *syntax* of **while** *loop* is:

```
        expression-1;
        while(expression-2)
        {
                statement-1;
                statement-2;
                ....................
                ....................                    body of the loop
                statement-n;
                expression-3;
        }
```

- **expression-1** contain *initialization statement(s)* which is/are is mentioned before the starting of the **while** loop.

The *statement(s)* are *assignment statement(s)* used to set the *loop* control *variable(s)*. These *statement(s)* will *execute* before the first *iteration* of the *loop*.

- **expression-2** contain *condition(s)*.

  The *condition(s)*, that determine the termination of the *loop*. Before every *iteration*, the *condition(s)* is/are *checked* and if found *true* then the next *iteration* will take place.
- **expression-3** contain *increment/decrement statement(s)*.

  The *statement(s)* denote how to change the states of the control *variable(s)* in each *iteration*.
- The section within "{" and "}" is called as the *body* of the *loop*.

Space for learners notes

Example 3.22: Write a program to display all the numbers between two input numbers using while loop.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int start_no, end_no, i;
clrscr();
printf("Enter the Starting No:\t");
scanf("%d", &start_no);
printf("Enter the Ending No:\t");
scanf("%d", &end_no);
printf("The numbers between %d and %d are:\n", start_no, end_no);
i=start_no;
while(i<=end_no)
{
        printf("%d\t",i);
        i++;
}
getch();
}
```

Explanation:

- Two numbers are taken as input using the "scanf()" function and stored into the variables "start_no" and "end_no".

Space for learners notes

- In expression-1, the *variable* "i" is initialized to the value of the variable "start_no" as the starting no. of the range is in this variable.
- In expression-2, the *condition* is set as "i<=endno" because the *loop* will continue as long as the value of "i" remains less than or equal to the value of the variable "end_no".
- Inside the body of the loop, the "printf()" will display the current value of "i" with a space(\t is used for tabbed space). In expression-3, the value of "i" is incremented by 1.

Example 3.23: Write a program to display sum of all the digits of a number entered by the user.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num, rem, quo,sum=0;
printf("Enter a number:\t");
scanf("%d",&num);
while(num>0)
{
        rem = num%10;
        quo = num/10;
        sum = sum + rem;
        num = quo;
}
printf("The sum of digits of the number is %d", num, sum);
getch();
}
```

Output:

```
Enter a number: 12345
The sum of digits of the number is: 15
```

Example 3.24: Find factorial of a number using while loop.

Solution:

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

```
int num,fact;
printf("Enter a number.\t");
scanf("%d",&num);
fact=1;
while(num>0)
{
fact=fact*num;
—num;
}
printf("Factorial is: %d",num, fact);
getch();
}
```

Output:

```
Enter a number.5
Factorial=120
```

Example 3.25: Write a program to display the Fibonacci series up to n number of terms using while loop.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int count, n, t1=0, t2=1, display=0;
printf("Enter number of terms:\t");
scanf("%d",&n);
/* Displaying first two terms */
printf("Fibonacci Series: %d+%d+", t1, t2);
count=2; // count=2 because first two terms are already displayed.
while(count<n)
{
display=t1+t2;
t1=t2;
t2=display;
++count;
printf("%d+",display);
}
getch();
}
```

Space for learners notes

Space for learners notes

Output

```
Enter number of terms: 10
Fibonacci Series: 0+1+1+2+3+5+8+13+21+34+
```

Suppose, instead of number of terms, you want to display the Fibonacci series until the term is less than certain number entered by user. Then, this can be done using the code as below:

```
/* Displaying Fibonacci series up to certain number entered by user. */
#include<stdio.h>
#include<conio.h>
void main()
{
int t1=0, t2=1, display=0, num;
printf("Enter an integer: \t");
scanf("%d",&num);
/* Displaying first two terms */
printf("Fibonacci Series: %d+%d+", t1, t2);
display=t1+t2;
while(display<num)
{
printf("%d+",display);
t1=t2;
t2=display;
display=t1+t2;
}
getch();
}
```

Output

```
Enter an integer: 200
Fibonacci Series: 0+1+1+2+3+5+8+13+21+34+55+89+144+
```

Example 3.26: Write a program to display the number of digits of an input number.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
long int n, count=0;
printf("Enter an integer: \t");
scanf("%ld",&n);
while(n!=0)
{
n=n/10;
++count;
}
printf("Number of digits: %d", count);
getch();
}
```

Output

```
Enter an integer: 34523
Number of digits: 5
```

Explanation: This program takes an integer from user and stores that number in variable *n*. Suppose, user entered 34523. Then, while loop is executed because n!=0 will be true in first iteration. The codes inside while loop will be executed. After first iteration, value of n will be 3452 and *count* will be 1. Similarly, in second iteration *n* will be equal to 345 and *count* will be equal to 2. This process goes on and after fourth iteration, n will be equal to 3 and *count* will be equal to 4. Then, in next iteration n will be equal to 0 and *count* will be equal to 5 and program will be terminated as n!=0 becomes false.

Example 3.27: Write a program to display character from A to Z using while loop.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char c;
clrscr();
c='A';
printf("The letters from A to Z are:\n");
while(c<='Z')
{
printf("%c ",c);
++c;
}
getch();
```

Space for learners notes

Space for learners notes

```
}
```

Output

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Example 3.28: Write a program to display the reverse of a number input by the user.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
long int num, rem,reverse=0;
clrscr();
printf("Enter a number:\t");
scanf("%ld",&num);

while(num !=0)
{
 rem = num%10;
 reverse = reverse*10 + rem;
 num = num/10;
}
printf("\nThe reverse number is %ld",reverse);
getch();
}
```

3.4.3 The *do while* loop

The *syntax* of **do-while** *loop* is:

```
        expression-1;
        do
        {
                Statement 1;
                Statement 2;
                ..................
                ..................                      body of the loop
                Statement n;
                expression-3;
        }while(expression-2);
```

- **expression-1** contain *initialization statement(s)* which is/are mentioned before the starting of the **while** loop.

The *statement(s)* are *assignment statement(s)* used to set the *loop* control *variable(s)*. These *statement(s)* will *execute* before the first *iteration* of the *loop*.

- **expression-2** contain *condition(s)*.

  The *condition(s)*, that determine the termination of the *loop*. After every *iteration*, the *condition(s)* is/are checked and if found *true* then the next *iteration* will take place.
- **expression-3** contain *increment/decrement statement(s)*.

  The *statement(s)* denote how to change the states of the control *variable(s)* in each *iteration*.
- The section within "{" and "}" is called as the *body* of the *loop*.

Space for learners notes

Example 3.29: Write a program to check whether an input character is a vowel or not. The program should continue as many times the user wishes.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char c, ans;
        do
        {
        clrscr();
        printf("Enter the character to check:\t");
        scanf("%c", &c);
        switch(c)
        {
                case 'a':
                case 'A':
                case 'e':
                case 'E':
                case 'i':
                case 'I':
                case 'o':
                case 'O':
                case 'u':
                case 'U': printf("The input character is a vowel");
                        break;
                default: printf("The input character is not a vowel");
```

Space for learners notes

```
        }
        printf("\nDo you want to continue(y/n):");
        scanf("%c", &ans);
        }while(ans == 'y');
        getch();
}
```

Explanation:

- Inside, the **do-while** loop
    - First, the character to be checked is taken input in variable "c".
    - In the "switch" statement, since the output will the same for all the vowels [i.e. 'a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U'], the cases with all the vowels(small letter and capital letter) are mentioned serially and in the last case, i.e., "case 'U', the display statement is mentioned for displaying "The input character is a vowel". And then because of the "break" statement, the "switch" statement will end.

      The case "default" will satisfy if all the above cases do not satisfy and "The input character is not a vowel" will be displayed and the "switch" statement will end.
    - After "switch" statement, the user is asked whether he/she wishes to continue by taking a character input to variable "ans".
- In the condition inside "while" the value of variable "ans" is compared with character 'y' for equality and if it satisfies then the "do-while" loop will continue otherwise the "do-while" loop will terminate and thus the program ends.

3.5 COMPARISION OF THE LOOP STATEMENTS

Space for learners notes

	for loop	**while** loop	**do-while** loop
1.	A **for** loop is used to execute a block of statements depending on the condition which is evaluated at the beginning of the loop.	A **while** loop is used to execute a block of statements depending on the condition which is evaluated at the beginning of the loop.	A **do-while** loop is used to execute a block of statements depending on the condition which is evaluated at the end of the loop.
2.	The block of statements will not be executed when the **condition does not satisfy**, i.e., value of the condition is **false**.	The block of statements will not be executed when the **condition does not satisfy**, i.e., value of the condition is **false**.	The block of statements will not be executed when the **condition does not satisfy**, i.e., value of the condition is **false** but the block will execute at-least once irrespective of the value of the condition.
3.	Variable(s) is/are initialized at the beginning of the loop which is/are used to control the loop.	Variable(s) is/are initialized before the starting of the loop which is/are used to control the loop.	Variable(s) is/are initialized before the starting of the loop which is/are used to control the loop.
4.	The statements to change the state of the control variable(s) is/are mentioned within "( )" in expression-3.	The statements to change the state of the control variable(s) is/are mentioned just before the end of the body of the loop.	The statements to change the state of the control variable(s) is/are mentioned just before the end of the body of the loop.

Space for learners notes

3.6 NESTED LOOP

A loop may contain another loop within its body. This form of loop inside a loop is called *nested loop.* In a nested loop, the inner loop must terminate before the outer loop can be ended.

Example 3.30: Write a program to display the multiplication table of 1, 2 and 3.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,j;
        clrscr();
        printf("The multiplication table from 1 to 3 are:\n");
        for(i=1;i<=3;i++)
        {
                for(j=1;j<=10;j++)
                printf("%d X %d = %d\n", i,j,i*j);
                printf("\n");
        }
        getch();
}
```

3.7 GOTO STATEMENT

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

```
        goto label;
```

where ***label*** is a valid indentifier to indicate the destination where a control can be transferred.

Example 3.31: Using goto statement, write a program to display the larger number of two numbers entered by the user.

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

```
int a,b;
clrscr();
printf("Enter two numbers:\t")
scanf("%d%d",&a,&b);
if(a>b)
        goto label1;
else
        goto label2;
label1:
        printf("The larger number is:%d\n",a);
        goto end;
label2:
        printf("The larger number is:%d\n",b);
end:
        return 0;
}
```

Output:

```
Enter two numbers: 10 20
The larger number is: 20
```

3.8 BREAK STATEMENT

The *break* statement causes an immediate exit from the innermost loop. When the keyword *break* is encountered inside any loop, control automatically passes to the statement after the loop. The break statement can also be used with *switch* statement that we studied earlier.

Example 3.32:

```
#include <stdio.h>
#include <conio.h>
void main(){
int i, num;
clrscr();
for (i=1;i<=10;i++)
{
printf("\n\nEnter a number:\t");
scanf("%d",&num);
if(num<0)
{
```

Space for learners notes

Space for learners notes

```
printf("\nYou have entered a -ve number.\n");
break;
}
printf("\nThe value of i in the loop is: %d", i);
printf("\nThe number you have entered is: %d", num);
}
printf("\nGood bye");
getch();
}
```

Output:

```
Enter a number: 10
The value of i in the loop is:1
The number you have entered is: 10
Enter a number: 20
The value of i in the loop is: 2
The number you have entered is: 20
Enter a number: -5
You have entered a -ve number.
Good Bye
```

Explanation:

Here is this case, when we put the value as -5, the statement "if(num<0)" returns true, so the statement "You have entered a -ve number." is displayed. Since we have used a ***break*** statement after that statement, the program control terminates the loop immediately. So, the statements i.e. printf("\nThe value of i in the loop is: %d", i); and printf("\nThe number you have entered is: %d", num);are not displayed and the statement printf("\nGood bye");is displayed.

3.9 CONTINUE STATEMENT

Sometimes we want to take the control to the beginning of the loop by passing the statements inside the loop which have not yet been executed. The *continue* statement forces the next iteration of the loop to take place, skipping any statement(s) following the *continue* statement in the body of the loop. The syntax of the *continue* statement is:

```
continue;
```

STOP TO CONSIDER

continue statement is not used with switch statement

Example 3.33:

```
#include <stdio.h>
#include <conio.h>
void main(){
int i, value;
clrscr();
for (i=1;i<=4;i++)
{
printf("\n\nEnter a number:\t");
scanf("%d", &value);
if (value<=0)
{
printf("\nZero or -ve value found\n");
continue;
}
printf("\nThe value of i in the loop is: %d", i);
printf("\nThe entered number is: %d", value);
}
getch();
}
```

Output:

```
Enter a number: 10
The value of i in the loop is: 1
The entered number is: 10
Enter a number: 20
The value of i in the loop is: 2
The entered number is: 20
Enter a number: -5
Zero or -ve value found
Enter a number: 30
The value of i in the loop is: 4
The entered number is: 30
```

Explanation:

In the above example, when we put the value as -5, the *if* condition returns true and the statement inside the *if* block " Zero or -ve value found " is displayed. But since there is a *continue* statement after that statement, the program control skips the next two printf() statements and goes to the next iteration of the loop.

Space for learners notes

Space for learners notes

3.10 EXIT() FUNCTION

The function exit() is used to terminate the program execution immediately. The syntax is:

```
exit(status);
```

where 'status' is the termination value returned by the program and is an integer. Normal termination usually returns 0.

CHECK YOUR PROGRESS 3

5. State whether the following statements are ***true*** or ***false***
   a) The while and for loops cannot be nested loops the way if statement can be nested.
   b) Loop is a mechanism to execute a set of statements a number of times.
   c) The break statement help immediate exit from any part of the loop.
   d) A while loop may always be converted to an equivalent for loop.
   e) In a C program, use of goto statement is generally not recommended.
   f) You can use one break statement in one loop.
   g) The exit() function causes an exit from a function.
   h) It is not possible to have a switch statement nested within while or for loops.
   i) A continue statement causes an exit from a loop.
   j) A do while loop is useful when the body of the loop will be executed at least once.
   k) Multiple increment expressions in a for loop expression are separated by commas.
   l) If a loop does not contain any statement in its loop body it is said to be an empty loop.
   m) A loop can contain another loop in its body.
   n) The while loop evaluates a test expression before allowing entry into the loop.
   o) In nested loops, the inner loop must terminate before the outer loop terminates.
   p) Statements inside a do while loop will be executed at least once.
   q) The continue statement is used to skip some statements within a loop and start next iteration.
   r) The break statement is used when it is required to exit from a loop other than by testing of termination condition.
6. Fill in the blanks:
   a) The initialisation, testing and incrimination can be done in the ________ statement itself.

b) Nesting can be done upto ______ level for while loop.

c) Example of an infinite loop is ________

d) A ________ is used to separate the three parts of the loop expression in a for loop.

e) When the ________ statement is executed, the program skips the remaining statements in the loop and goes back to test the loop condition.

f) An infinite for loop has _______ missing expression.

g) A ________ loop can also be an empty loop, if it contains just a null statement in its body.

h) The ________ is executed at least once always before it evaluates the test expression.

i) ________ statement exits from some deeply nested structure at once.

j) ________ function forces exit from a program.

k) ________ statement forces immediate exit from any loop.

l) The ________ statement is used to skip some statements within a loop and start next iteration.

Space for learners notes

3.11 SUMMING UP

There are three ways for taking decisions in a program. First way is to use the **if else** statement, second way is to use the conditional operators and third way is to use the **switch** statement. The default scope of the **if** statement is only the next statement. So, to execute more than one statement they must be written in a pair of braces.

An **if** block need not always be associated with an **else** block. However, an **else** block is always associated with an **if** statement.

Loop structures are used to execute a statement/block of statements repeatedlly a number of times. Three types of loops are used in C: ***for***, ***while*** and ***do while***. In both ***for*** and ***while*** loop, the condition is checked before each iteration of the loop. But in case of ***do while*** loop, the condition is checked after each iteration of the loop.

The ***goto*** statement transfers control to a label. The ***break*** statement terminates the execution of the nearest enclosing ***do***, ***for***, ***while*** or ***switch*** statement in which it appears.

The continue statement passes control to the next iteration of the nearest enclosing ***do***, ***for*** or ***while*** statement in which it appears bypassing any remaining statements in the ***do***, ***for*** or ***while*** statement body.

3.11 ANSWERS TO CHECK YOUR PROGRESS

1. The *if* statement is a control statement that tests a particular condition. Whenever, the evaluated condition comes out to be true, then that action or the set of actions

Space for learners notes

are carried out. Whereas, if else statement is used to execute a statement block or a single statement depending on the value of a condition. If the condition evaluates to true, then the statement(s) inside the if block will be executed, otherwise, the statement(s) inside the else block will be executed.

2. An if statement may have another ***if*** statement in the true condition block and false condition block. This compound statement is called ***nested if*** statement.

3. 20 5

4.

a) False

b) False

c) False

d) False

e) True

5.

a) False

b) True

c) True

d) True

e) True

f) False

g) False

h) False

i) False

j) True

k) True

l) True

m) True

n) True

o) True

p) True

q) True

r) True

6.

a) for

b) 3

c) while(1)

d) ;

e) continue

f) test

g) while or for

h) do while

i) goto

j) exit()

k) break

l) continue

Space for learners notes

3.13 POSSIBLE QUESTIONS

Short answer type questions:

1. What is the effect of absence of break in switch case statement? What is the purpose of default?
2. What is the similarity and difference between break and continue statement?
3. What is the function of break statement in a loop?
4. Why the use of goto statement should generally be avoided in a C program?
5. Differentiate between while and do while loop.
6. Differentiate between break and exit ().

Long answer type questions:

1. Explain the various if structures with suitable examples.
2. Differentiate between if else and switch structures with an example.
3. What is nested if statement? Explain with an example.
4. Write a program in C to display the smallest of three numbers entered by the user.
5. Explain the loop control structures used in C with examples.
6. Write a program in C to convert a binary number into a decimal number.
7. Write a program in C to convert a decimal number into its equivalent binary number.
8. Write a program to check whether a number is palindrome or not.
9. Write a program to display the multiplication table of 5.
10. Write a program to display all the numbers divisible by 7 between 20 and 200.
11. Write a program to generate the first 100 positive integers divisible by 5.
12. Write a program in C to evaluate the series:

    Sum = 1 + 1/2 + 1/3 + ... + 1/n
13. Write a program in C to display the prime numbers between 10 and 100.

Space for learners notes

14. Write a program in C to display the factorial of all the numbers between 1 and 10.
15. Write a program in C to evaluate the series:

    1 + 1/3 +1/5 + ...+ n

3.14 FURTHER READINGS

1. Kanetkar, Yashavant P. *Let us C*. BPB publications, 2016.
2. E Balagurusamy .*Programming in ANSI C.* Tata McGraw-Hill publications, 2006.

UNIT 4 ARRAYS AND STRINGS

Space for learners notes

CONTENTS

4.1 INTRODUCTION

In unit 2, we have learnt how to declare a variable and input a value. Now sometime in a C program, more than one similar type of data is required as inputs. In such cases, it may happen that the number of required input data is very large or it is dependent on the user input. So in such cases, it is not possible to declare different variables for all the required input data. Here we can use the concept of Array in C programming.

Space for learners notes

4.2 OBJECTIVE

After going through this unit student will able to:

- What are Arrays?
- Different types of Arrays.
- How to declare and initialize an Array?
- Different operations on one dimensional array and two dimensional array.
- What is a string?
- How to input and display a string?
- Different operations on strings.
- What is array of strings?
- Different library functions on strings.

4.3 DEFINITION OF ARRAY

An array is a collection of homogeneous pieces of data that are all identical in type and stored in consecutive memory locations. For example in Fig.4.1, A is an integer array storing 10 integer numbers.

	0	1	2	3	4	5	6	7	8	9
A	23	45	32	67	8	12	123	89	90	22

Fig.4.1: Example of integer array storing 10 integers

Now let us assume that the base address of A is 5012. Base address of an array is the memory address of the first element in the array. So here 5012 is the memory address of 23. Now according to the definition of array, the memory address of the second element of A is 5014 as the memory size of int type variable is 2 bytes. In this way, the memory address of the third and fourth element of A is 5016 and 5018 respectively as the array elements are stored in consecutive memory locations.

4.4 TYPES OF ARRAY AND DECLARATION

There are two types of array available in C programming that are explained as follows:

4.4.1 One dimensional array:

In one dimensional array, data is stored row or column wise and are hold in consecntive memory locations.

The declaration syntax of one dimensional array is

Datatype arraynm [N];

Here datatype specifies type of the data to be stored in the array and arraynm is the name of the array. [N] specifies that arraynm is a one dimensional array and it can store maximum N number of elements.

For example: int arr[30] ;

Here, int specifies that the array will store integer type of data and arr is the name of the array. The array can store maximum 30 number of integer type data.

Space for learners notes

4.4.2 Multi-dimensional array

In multi-dimensional array, data is held both row and column wise.

The declaration syntax of multi dimensional or N dimensional array is

Datatype arraynm [size1][size2][size3]........[sizeN]

For example, declartion of a 2 dimensional array is

float arrtwo [30][20] ;

Here [30][20] means arrtwo is a 2 dimensional array and it can store maximum 30×20 = 600 numbers of float type data. A two dimensional array also can be called as a matrix.

Again example of declaring a three dimensional array is

int arrthree[10][20][10];

Here, arrthree is a three dimensional array which can store maximum 10X20X10 = 2000 number of integer type data.

4.5 OPERATIONS ON ONE DIMENSIONAL ARRAY

There are different operations that can be performed on one dimensional arrays as explained in the following sub-sections.

4.5.1 Initialization

A one dimensional array can be initialized by using the following statements.

```
int arr1[5] = {4 , 7 , 8 , 23 , 56} ;
int arr2[ ] = {2 , 7 , 1 , 9};
char arr3[ ] = {'A', 'H', 'J', 'B'};
```

By initializing a one dimensional array, we can store some initial values as array elements into the array at compile time. From the above statements, the array

Space for learners notes

arr1 is initialized with the values 4 , 7 , 8 , 23 , 56 as first, second, third, fourth and fifth element of arr1 respectively. Now if an array is not initialized then it contains garbage values because by default the storage class of array is auto. So if the storage class of an array is declared to be static then all the array elements will be initialized to zero.

4.5.2 Read and Access of Array Elements

We have to learn how to access individual element in a one dimensional array and how array elements can be read from standard input device. Accessing of array elements can be performed with the number in the brackets (for example: [4]) following the array name. This number is called as subscript. This number specifies the element's position in the array. In C programming, all the array elements are numbered, starting with subscript value 0. So an array of size 20 has subscript values starting from 0 to 19. So if we want to access the 5th element of an array 'arr' then we can use 'arr[4]' . Now we can read and display the 5th element of an integer array 'arr' with the help of the following programming statements.

```
int arr[30];
scanf( "%d" , &arr[4] ) ; /* an int type data is read from the standard input device
                              into the 5th position of arr */
printf(" %d", arr[4] ) ;   /* the 5th element of arr is displayed in the standard output
                              device*/
```

Now from fig. 4.1, A[0] will refer to the first element of A which is 23 and in this way A[9] will refer to the 10th element of A which is 22.

Here we have an important point that is what happens when the subscript value used in the time of reading an array element is greater than or eqtial to the size of the array. In such cases, for C programming, data will be entered into the memory space outside the memory space allocated for the array. Sometimes this entered data will replace other important data outside memory space of the array and in some cases the system may stop responding. So there should be a conditional statement in our C programs to check that the subscript value never exceed the array size in the time of reading array elements.

STOP TO CONSIDER:

Direct access or random access of array elements is possible because the array elements are stored in consecutive memory locations.

Now, a C program to input and display n elements in an integer array is given below.

```
# include <stdio.h>
# include <conio.h>
void main()
{
   int arr[30];
   int i , n;
   clrscr();
  printf("\nHow many numbers you want to enter(maximum 30)=" );
   scanf("%d",&n);

   if( n>30 )
   {
        printf("\nYour entered quantity of numbers exceeds the size
of the array");
   }
   else
   {
        for( i = 0 ; i < n ; i++)
        {
           printf("\nEnter the %dth number =", i+1);
           scanf("%d", &arr[i]);
        }
        for( i = 0 ; i < n ; i++)
        {
           printf("\nThe %dth number in the array is= %d", i+1 , arr[i]);
        }
   }
   getch();
}
```

Space for learners notes

4.5.3 Searching and Sorting

Now searching a particular element in an array is another important operation performed on arrays. This can be performed by comparison operation between the element to be searched and elements available in the array. Two fundamental searching algorithms are linear search and binary search.

Arranging array elements in ascending or descending order in an array is called the sorting operation. There are different algorithms available for sorting operation. For example: Babble sort, Selection sort, Insertion sort etc.

A C program to search an element in an integer array using linear search technique and display the subscript value where the element is present in the array is given as follows.

Space for learners notes

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int arr[30];
    int i , n , sno , flag = 0;
    clrscr();
   printf("\nHow many numbers you want to enter(maximum 30) =" );
   scanf("%d", &n);
    if( n > 30 )
    {
          printf("\nYour entered quantity of numbers exceeds the size of
the array");

    }
   else
    {
          for( i = 0 ; i < n ; i++)
          {
                  printf("\nEnter the %dth number=", i+1);
                  scanf("%d", &arr[i]);
          }
          printf("\n Enter the number to be searched in the array =" );
          scanf("%d", &sno);
          for(i = 0 ; i < n ; i++)
          {
                    if(sno = = arr[i])
                    {
                              printf("\n%d is present in the array in the array",
sno);
                              printf("\n Subscript value of the searched
element is = %d", i)
                              flag = 1;
                              break;
                    }
          }
        if(flag = = 0)
        {
            printf("\n%d is not present in the array",sno);
        }
  }
  getch();
}
```

Example 4.1: Write a C program to search a specific data in a one dimensional array using binary search algorithm.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int arr[50] , i , n , start , mid , end , src_data;
        clrscr();
        printf("\n Enter the total number of data in the array:");
        scanf("%d", &n);
        if(n <= 50)
        {
                printf("\n Enter data into the array:");
                for( i = 0 ; i < n ; i++)
                {
                        printf("\n Enter %dth data:", i+1);
                        scanf("%d", &arr[i]);
                }
                printf("\n Enter the data to be searched:");
                scanf("%d", &src_data);

                printf("\n The array data are:\n");
                for(i = 0 ; i < n ; i++)
                        printf("\t%d",arr[i]);
                start = 0;
                end = n-1;
                while(start <= end)
                {
                   mid = (start + end)/2;
                  if(arr[mid] = = src_data)
                   {
                        printf("\n %d is available in the array", src_data);
```

Space for learners notes

Space for learners notes

```
                printf("\n Subscript value of the searched element is = %d",
                mid)
                break;
            }
          else if(arr[mid] < src_data)
                start = mid+1;
           else
                end = mid-1;
         }

         if(start > end)
                printf("\n %d is not available in the
                array",src_data);
    }
    else
    {
         printf("\n The total number of data exceed the size of
         the array");
    }
    getch();
}
```

Example 4.2: Write a C program to find out the minimum and the maximum of the numbers present in an integer array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int arr[30];    //arr is a one dimensional integer array with size 30
   int i , n , min , max;
   clrscr();

   printf("\nHow many numbers you want to enter(maximum 30)=" );
   scanf("%d", &n);
   if(n > 30)
    {
      printf("\nYour entered quantity of numbers exceeds the size
of the array");
```

```
    }
    else
    {
        for( i = 0; i < n ; i++)
        {
         printf("\nEnter the %dth number =", i+1);
           scanf("%d",&arr[i]);
        }
        min = arr[0];
        max = arr[0];
        for( i = 1 ; i < n ; i++)
        {


            if(arr[i] < min)
                  min = arr[i];
            if(arr[i] > max)
                  max = arr[i];
        }
        printf("\nThe minimum of the numbers present in the
array is = %d",min);
        printf("\nThe maximum of the numbers present in the array is
= %d",max);

    }
    getch();
}
```

Example 4.3:

Write a C program to find out the summation of all the numbers present in an integer array.

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

void main()
{
   int arr[30];
   int i , n , sumarr = 0;
   clrscr();
```

Space for learners notes

Space for learners notes

```
        printf("\nHow many numbers you want to enter(maximum 30) =" );
        scanf("%d", &n);
    if(n > 30)
    {
        printf("\nYour entered quantity of numbers exceeds the size of the
        array");
    }

    else
    {

      for(i = 0 ; i < n ; i++)
      {
         printf("\nEnter the %dth number =" , i+1);
         scanf("%d", &arr[i]);
         sumarr = sumarr + arr[i];
      }

        printf("\nThe summation of the numbers present in the
        array is=%d",sumarr);
     }
     getch();
    }
```

Example 4.4: Write a C program to sort some integer numbers stored in a one dimensional array using selection sort algorithm.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int arr[50] , i , j , n , index , min;
        clrscr();
        printf("\n Enter the total number of data in the array:");
        scanf("%d", &n);
        if(n <= 50)
        {
                printf("\n Enter data into the array:");
                for(i = 0 ; i < n ; i++)
```

```
            {
                printf("\n Enter %dth data:", i+1);
                scanf("%d", &arr[i]);
            }
            printf("\n Before sorting the array data are:\n");
            for(i = 0 ; i < n ; i++)
                printf("\t%d", arr[i]);

            for(i = 0 ; i < n-1 ; i++)
            {
                index = i;
                min = a[i];

                for(j = i+1 ; j <= n-1 ; j++)
                {

                    if ( min > a[j] )
                    {
                        min = a[j];
                        index = j;
                    }
                }

                a[index] = a[i];
                a[i] = min;
            }
            printf("\n After sorting the array data are:\n");
            for(i =0 ; i < n ; i++)
                printf("\t%d",arr[i]);
   }
   else
   {
            printf("\n The total number of data exceed the size of the
array");
    }
   getch();
}
```

Space for learners notes

Space for learners notes

Example 4.5: Write a C program to sort some integer numbers stored in a one dimensional array using bubble sort algorithm

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int arr[50] , i , j , n , temp;
        clrscr();
        printf("\n Enter the total number of data in the array:");
        scanf("%d", &n);
        if(n <= 50)
        {
                printf("\n Enter data into the array:");
                for(i = 0 ; i < n ; i++)
                {
                        printf("\n Enter %dth data:", i+1);
                        scanf("%d", &arr[i]);
                }
                printf("\n Before sorting the array data are:\n");
                for(i = 0 ; i < n ; i++)
                        printf("\t%d", arr[i]);

                for (i = 0; i < n - 1;  i++)
                {
                    for(j = 0 ; j < n-i-1 ; j++)
                    {
                 if (arr[j] > arr[j + 1])
                  {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                   }
                      }
                }
                printf("\n After sorting the array data are:\n");
                for(i = 0 ; i < n ; i++)
                        printf("\t%d", arr[i]);
   }
   else
   {
                  printf("\n The total number of data exceed the size of the
array");
   }
   getch();
}
```

Example 4.6: Write a C program to sort some integer numbers stored in a one dimensional array using Insertion sort algorithm

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int arr[50] , i , j , n , key;
        clrscr();
        printf("\n Enter the total number of data in the array:");
        scanf("%d", &n);
        if (n <= 50)
        {
                printf("\n Enter data into the array:");
                for(i = 0 ; i < n ; i++)
                {
                        printf("\n Enter %dth data:", i+1);
                        scanf("%d",&arr[i]);
                }
                printf("\n Before sorting the array data are:\n");
                for(i = 0 ; i < n ; i++)
                        printf("\t%d", arr[i]);

                for(i = 1 ; i < n ; i++)
                {
                key = a[i];
                j = i-1;
                while (j >= 0 && a[j] > key)
                {
                        a[j+1] = a[j];
                        j--;
```

Space for learners notes

Space for learners notes

```
                    }
                    a[j+1]=key;
                    }
                    printf("\n After sorting the array data are:\n");
                    for(i = 0 ; i < n ; i++)
                            printf("\t%d", arr[i]);
    }
    else
    {
                    printf("\n The total number of data exceed the size of the
array");
    }
    getch();
}
```

STOP TO CONSIDER:

Let us declare an one dimensional array as int Arr[20]. Then Arr and &Arr[0] will provide the base address of the array.

4.6 OPERATIONS ON TWO DIMENSIONAL ARRAYS

4.6.1 Initialization

Two dimensional arrays can be initialized as follows:

```
int arrtwo1[4][3]= {
                {4, 8 , 9 },
                        {7, 9, 21},
                        {1, 8, 19},
                        {71, 6, 2}
                };
int arrtwo2[4][3] = { 4 , 8 , 9 , 7, 9, 21 , 1 , 8 , 19 , 71 , 6 , 2};
int arrtwo3[ ][3] = { 4 , 8 , 9 , 7, 9 , 21 , 1 , 8 , 19 , 71 , 6 , 2};
```

Fig. 4.2: illustrates this process.

Here, three ways of initializing a two dimensional array are shown above. In case of initializing two dimensional arrays, it is necessary to mention the second dimension of the array, otherwise it will not work in C programming. So, two dimensional array initializations as shown below will not work in C programming.

```
int arrtwo[ ][ ] = { 4 , 8 , 9 , 7, 9, 21 , 1 , 8 , 19 , 71 , 6 , 2};
int arrtwo[4][ ] = { 4 , 8 , 9 , 7, 9, 21 , 1 , 8 , 19 , 71 , 6 , 2};
```

	0	1	2
0	4	8	9
1	7	9	21
2	1	8	19
3	71	6	2

Fig. 4.2: Diagrammatic representation of the array arrtwo1 declared above

4.6.2 Read and Access Array Elements

We need two subscript values to access a specific cell of a two dimensional array. Here first subscript value will represent the row index and the second subscript value will represent the column index of the specific cell of a two dimensional array. For example, consider the two dimensional array arrtwo1 declared above whose diagrammatic representation is given in fig. 4.2. Here arrtwo1 [0][0] will refer to the first element in the array with row index 0 and column index 0 which is 4. Again arrtwo1[3][0] will refer to 71.

So arrtwo1[i][j] will give the array element with ith row number and jth column number of the array arrtwo1.

Now to display the array element with row number 3 and column number 2 of array arrtwo1 on the standard output device, the following programming statement in C can be used:

```
printf("%d",arrtwo1[3][2]);
```

So the output of the statement will be 2 .

Again to read a new array element from the standard input device into the cell with row number 3 and column number 2 of array arrtwo1, the following programming statement can be used.

```
scanf("%d", &arrtwo1[3][2]);
```

So result of this statement will be a new data from standard input device will replace the existing array element of arrtwo1 with row number 3 and column number 2 as arrtwo1 is initialized.

Example 4.7: Write a C program to find out the summation of all the numbers of a matrix with integer values.

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

Space for learners notes

Space for learners notes

```
          int i , j , row_no , col_no, matrix[20][20] ,sum = 0;
          clrscr();

          printf("\n Please enter the number of rows =");
          scanf("%d", &row_no);
          printf("\n Please enter the number of columns =");
          scanf("%d", &col_no);
          printf("\nPlease enter the matrix:");

          for (i = 0 ; i < row_no ; i++)
          {
                    for (j = 0; j < col_no ; j++)
                    {
                              printf("\nPlease enter the (%d,%d) th
                              data=", i , j);
                              scanf("%d", &matrix[i][j]);
                    }
          }
          for (i = 0 ; i < row_no ; i++)
          {
                    for (j = 0; j < col_no ; j++)
                    {
                              sum = sum+ matrix[i][j];
                    }
          }
        printf("\n The required summation is = %d", sum);
          getch();
}
```

Example 4.8: Write a C program to find out the summation of all the diagonal elements of a symmetric matrix with integer values.

```
#include <stdio.h>
#include <conio.h>
void main()
{
          int i , j , row_no , col_no, matrix[20][20] ,sum = 0;
          clrscr();
printf("\n Please enter the number of rows = ");
scanf("%d", &row_no);
printf("\n Please enter the number of columns = ");
scanf("%d", &col_no);

if(row_no != col_no)
{
    printf("\n Wrong input. Symmetric matrix required here");
}
else
{
            printf("\nPlease enter the matrix:");

            for (i = 0 ; i < row_no ; i++)
            {
                        for (j = 0; j < col_no ; j++)
                        {
                        printf("\nPlease enter the (%d,%d)
                        th data =", i , j);
                           scanf("%d", &matrix[i][j]);
                        }
            }
 for (i = 0 ; i < row_no ; i++)
            {
                        for (j = 0; j < col_no ; j++)
                        {
                                    if(i = = j)
                                    {
                                                sum = sum+ matrix[i][i];
                                    }
                        }
            }
 printf("\n The required summation is = %d", sum);
     }
    getch();
}
```

Space for learners notes

Space for learners notes

Example 4.9: Write a C program to add two matrix containing integer data

```
#include <stdio.h>
#include <conio.h>
void main()
{
          int i , j , row_no , col_no ;
          int matrix1[20][20] , matrix2[20][20] , sum_matrix[20][20];
          clrscr();

          printf("\n Please enter the number of rows = ");
          scanf("%d", &row_no);
          printf("\n Please enter the number of columns = ");
          scanf("%d", &col_no);

          printf("\nPlease enter the first matrix:");

          for (i = 0 ; i < row_no ; i++)
          {
                    for (j = 0; j < col_no ; j++)
                    {
                              printf("\nPlease enter the
(%d,%d) th data of matrix1=", i , j);
                              scanf("%d", &matrix1[i][j]);
                    }
          }
          printf("\nPlease enter the second matrix:");
          for (i = 0 ; i < row_no ; i++)
          {
                    for (j = 0; j < col_no ; j++)
                    {
                            printf("\nPlease enter the
                    (%d,%d) th data of matrix2 = ", i , j);
                              scanf("%d", &matrix2[i][j]);
                    }
          }

          for (i = 0 ; i < row_no ; i++)
          {
            for (j = 0 ; j < col_no ; j++)
            {
                        sum_matrix[i][j] = matrix1[i][j] +
                        matrix2[i][j];

            }
}

printf("\n The first matrix is:\n");

for (i = 0 ; i < row_no ; i++)
{
            for (j = 0; j < col_no ; j++)
            {
                        printf("\t%d",matrix1[i][j]);
            }
            printf("\n");
}

printf("\n The second matrix is:\n");

for (i = 0 ; i < row_no ; i++)
{
            for (j = 0; j < col_no ; j++)
            {
                        printf("\t%d",matrix2[i][j]);
            }
            printf("\n");
}

printf("\n The resultant matrix after summation is:\n");

for (i = 0 ; i < row_no ; i++)
{
            for (j = 0; j < col_no ; j++)
            {
                        printf("\t%d", sum_matrix[i][j]);
```

Space for learners notes

Space for learners notes

```
                        }
                        printf("\n");
                }
                getch();

}
```

Example 4.10: Write a C program to multiply two matrix containing integer data

```
#include <stdio.h>
#include <conio.h>
void main()
{
     int i , j , k , r1 , r2 , c1 , c2 , sum = 0;
    int matrix1[20][20] , matrix2[20][20] , matrix_mult[20][20];

    printf("\nPlease enter the number of rows of the first matrix=");
     scanf("%d", &r1);
    printf("\nPlease enter the number of columns of the first matrix =");
     scanf("%d", &c1);

    printf("\nPlease enter the number of rows of the second matrix =");
     scanf("%d", &r2);
    printf("\nPlease enter the number of columns of the second matrix
=");
     scanf("%d", &c2);

      if (c1 != r2)
        printf("\n Matrix multiplication for these dimensions of
matrices is not possible");

      else
      {
              printf("\nPlease enter the first matrix:");

              for (i = 0 ; i < r1 ; i++)
              {
                      for (j = 0; j < c1 ; j++)
                      {
```

```
                    printf("\nPlease enter the
              (%d,%d) th data of  matrix1 = ", i , j);
                    scanf("%d", &matrix1[i][j]);
              }
        }

        printf("\nPlease enter the second matrix:");

        for (i = 0 ; i < r2 ; i++)
        {
              for (j = 0; j < c2 ; j++)
              {
                    printf("\nPlease enter the (%d,%d)
th data of matrix2 = ", i , j);
                    scanf("%d", &matrix2[i][j]);
              }
        }

       for (i = 0; i < r1; i++)
       {
              for (j = 0; j < c2; j++)
              {
                    for (k = 0; k < r2; k++)
                    {
                         sum = sum + matrix1[i][k] *
matrix2[k][j];
                    }

                   matrix_mult[i][j] = sum;
                    sum = 0;
              }
        }


      printf("\n The first matrix is:\n");

       for (i = 0 ; i < r1 ; i++)
        {
```

Space for learners notes

Space for learners notes

```
                for (j = 0; j < c1 ; j++)
                {
                        printf("\t%d",matrix1[i][j]);
                }
                printf("\n");
        }

    printf("\n The second matrix is:\n");

        for (i = 0 ; i <r2 ; i++)
        {
                for (j = 0; j < c2 ; j++)
                {
                        printf("\t%d", matrix2[i][j]);
                }
                printf("\n");
        }

    printf("\n The resultant matrix after multiplication is:\n");

        for (i = 0 ; i < r1 ; i++)
        {
                for (j = 0; j < c2 ; j++)
                {
                        printf("\t%d", matrix_mult[i][j]);
                }
                printf("\n");
        }
    }

  getch();
}
```

STOP TO CONSIDER:

Let us declare a two dimensional array as int Arr2[20][30]. Then Arr2, Arr2[0] and &Arr2[0][0] will provide the base address of the array.

CHECK YOUR PROGRESS

Space for learners notes

1. **Multiple choices**

   (a) In C programming, the subscript value of an array is starting from_____.

   (i) 0
   (ii) 1
   (iii) Compiler dependent
   (iv) None of the above

   (b) int arr[20];

   The meaning of the above statement is_____.

   (i) arr is a integer variable
   (ii) arr is a integer array capable of storing 19 integer numbers
   (iii) arr is an array capable of storing 20 data
   (iv) arr is an integer array capable of storing 20 integer numbers

   (c)
```
int arr[5] = {5,2,0,1,4};
arr[3] = arr[1] + arr[4];
for(i = 0 ; i < 5 ; i++)
printf(" %d", arr[i] );
```
   The output of the above statements is

   (i) 5 2 0 1 4
   (ii) 5 2 0 4 4
   (iii) 5 1 0 6 4
   (iv) 5 2 0 6 4

   (d) If arr is a character array and the memory address of arr[0] is 203 then memory address of arr[3] is _____.

   (i) 204
   (ii) 205
   (iii) 206
   (iv) None of the above

   (e) If you don't initialize an array what will be the elements set to?

   (i) 0
   (ii) an undetermined value
   (iii) a floating point number
   (iv) the character constant '\0'

   (f) What will happen if you try to put so many values into an array when you initialize it that the size of the array is exceeded?

   (i) Nothing
   (ii) possible system malfunction

Space for learners notes

(iii) Error message

(iv) Other data may be overwritten

(g) What will happen if you put too few elements in an array when you initialize it?

(i) Nothing

(ii) possible system malfunction

(iii) Error message

(iv) Unused spaces will be filled with 0's or garbage.

2. **State whether true or false**

(a) To declare an integer array we have to write int arr = size(20);

(b) Array can be used to store different types of data.

(c) In C programming ,the subscript value of an array is starting from 0.

(d) The subscript value of the last element of an array of size 10 is 10 in C programming.

(e) In C programming, an array cannot be initialized.

4.7 DEFINITION OF STRING

String is a collection of some characters stored in a one dimensional character array. A string is always terminated by '\0' which is called NULL character. The ASCII value of '\0' is zero. For example in fig.4.3, A is a character array with array size 10 and it stores the string "Welcome".

	0	1	2	3	4	5	6	7	8	9
A	W	E	l	c	o	m	e	\0		

Fig.4.3: Example of a string

STOP TO CONSIDER:

The maximum length of any string that will be stored in a character array, stm[N] is N-1 as string must be terminated by the NULL('\0') character.

4.8 INPUT AND DISPLAY A STRING

At first we require a character array to input a string. Now we can use different library functions like scanf(), gets(), getchar() to input a string. For example:

```
char str[40];
```

(a) scanf() can be used as:

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

(b) gets() can be used as:

```
gets(str);
```

(c) getchar() can be used as:

```
int i = 0;
char ch;
while((ch = getchar()) != '\n')
{
        str[i] = ch;
        i++;
}
str[i] = '\0';
```

Here str is the character array where we have input a string using scanf(), gets() and getchar(). But scanf() is not capable to input a multi word string, so in case of multi word string we can use gets() or getchar(). getchar() is a input function which can be used to input a character. So getchar() can be used to input all the characters of a string one by one with the help of a loop control and at the end , the NULL character(\0) is entered.

Now to display a string we can use different library function like printf(), puts() . For example:

(a) printf() can be used as:

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

(b) puts() can be used as:

```
puts(str);
```

STOP TO CONSIDER:

The character display function, putchar() can also be used to display strings. All the characters of a string can be displayed by using putchar() inside a loop control.

4.9 OPERATIONS ON STRINGS

There are different operations performed on strings that are discussed as follows.

(a) A string is initialized as:

```
char str[ ] = "Welcome";
Or
char str[ ] = {'W','e','l','c','o','m','e','\0'};
```

Space for learners notes

Space for learners notes

Here in the first declaration, '\0' is not necessary. C compiler inserts the NULL character (\0) automatically.

(b) Length of a string can be estimated by just finding the subscript value of the NULL character (\0) in the character array where the string is stored. So searching operation is performed for the NULL character (\0) in the string and the subscript value of the NULL character (\0) is the required length of the string.

A C program to find out the length of a string is given below.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        char ch, str[30];
        int slen = 0;
        clrscr();
        printf("\nEnter a string =");
        gets(str);
        while(str[slen]! = '\0')
        {
            slen++;
        }
        printf("\n The length of the string is =%d",slen);
        getch();
}
```

(a) To concatenate one string at the end of another string, we have to find out the subscript value of the NULL character that is stored in the string where concatenation will be performed. Then we have to assign each character of the string that is to be concatenated to the other string in consecutive position starting from the estimated subscript value.

A C program to concatenate a string at the end of an another string is given below

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char str1[30], str2[30];
        int slen = 0, i = 0;
        clrscr();
        printf("\nEnter the first string =");
        gets(str1);
        printf("\nEnter the second string =");
        gets(str2);
        while(str1[slen]! = '\0')
        {
          slen++;
        }
        while(str2[i]! = '\0')
        {
          str1[slen] = str2[i];
          slen++;
          i++;
        }
        str1[slen] = '\0';
        printf("\n After concatenation the first string is=");
        puts(str1);
        getch();
}
```

(c) To copy a string to an another string we have to do just assign each character of the first string to the second string in consecutive subscript value positions starting from 0 to the subscript value where the NULL character(\0) is stored in the first string.

A C program to copy one string to another string is given below.

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

void main()
{
        char str1[30], str2[30];
        int i = 0;
        clrscr();
        printf("\nEnter the first string =");
        gets(str1);
        printf("\nEnter the second string =");
        gets(str2);
        while(str2[i]! = '\0')
        {
```

Space for learners notes

Space for learners notes

```
            str1[i] = str2[i];
            i++;
        }
        str1[i] = '\0';
        printf("\nAfter copy the first string is =");
        puts(str1);
        getch();
}
```

Example 4.11:

Write a C program to search and find out the number of occurrences of a specific character in a string.

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

void main()
{
  char str[30] ;
  char ch ;
  int i = 0, chcount = 0 ;
  clrscr();
  printf("\nEnter a string = ") ;
  gets(str) ;
  printf("\nEnter the character to be searched= ") ;
  scanf("%c", &ch) ;

  while(str[i] != '\0' )
   {
     if( str[i] == tolower(ch)  || str[i] == toupper(ch) )
          chcount++;
       i++;
   }
  if(chcount == 0 )
     printf("\n the character '%c' is not present in the string",ch) ; else
    printf("\nThe character is present in the string %d no. of times",chcount);
  getch();
 }
```

Example 4.12:

Write a C program to count the number of vowels present in a string.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        char str[ 30 ];
        int i = 0,vcount = 0;
        clrscr();
        printf("\nEnter a string =");
        gets( str );
        while( str[i] != '\0' )
        {
                switch( str[i] )
                {
                        case 'a':
                        case 'A':
                        case 'e':
                        case 'E':
                        case 'i':
                        case 'I':
                        case 'o':
                        case 'O':
                        case 'u':
                        case 'U': vcount++;
                }
                i++;
        }

        if ( vcount == 0 )
          printf("\n No vowel present in the string");
        else
            printf("\nThe number of vowel present in the string is
=%d",vcount);

        getch();
}
```

Space for learners notes

Space for learners notes

4.10 ARRAY ON STRINGS

Till now, we have learnt to read and display single strings by using one dimensional character arrays. But to read multiple strings, we require two dimensional character arrays where the first subscript value of the arrays indicates the total number of strings and the second subscript value indicate the maximum length of each strings. This is also called as array of strings. For example, let us consider a two dimensional character array Multi_Strn[20][40]. Now Multi_Strn[20][40] can be used to read 20 strings where maximum length of each string can be 39.

A C program is shown below where N number of employees' names is read and displayed.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        char Emp_names[50][100];
        int N, i;
        clrscr();
        printf("\n Enter the total number of names = ");
        scanf("%d",&N);
        if(N > 50)
        {
                printf("\n Maximum 50 names is possible");
        }
        else
        {
                printf("\n Enter names of %d number of employees::", N);
                for(i = 0 ; i < N ; i++)
                {
                        printf("\n Enter %dth name =", i+1);
                        gets(Emp_names[i]);
                }
                printf("\n The list of employee names is:\n");
                for(i = 0 ; i < N ; i++)
                {
                        puts(Emp_names[i]);
                        printf("\n");
                }
        }
        getch();
}
```

In the above program, Emp_names[50][100] is declared as two dimensional character array to store maximum 50 number of employee names. Here Emp_names[i] point to the ith employee name.

Space for learners notes

4.11 STRING LIBRARY FUNCTIONS

Some of useful library functions on strings and their functionalities are given in the following table (TABLE 4.1).

TABLE 4.1: TABLE FOR STRING LIBRARY FUNCTIONS AND THEIR FUNCTIONALITIES

String Library Function	Functionality
strlen(strn)	Returns the length of the string strn
strcpy(strn1,strn2)	Copies the string strn2 to the string strn1
strncpy(strn1,strn2,N)	Copies first N characters of the string strn2 to the string strn1
strcat(strn1,strn2)	Concatenate the string strn2 at the end of the string strn1
strcmp(strn1,strn2)	Compares the two strings strn1 and strn2. If it returns 0 then strn1 and strn2 are equal .If it returns a positive value then strn1 is greater than strn2. If it returns a negative value then strn2 is greater than the strn1.
strncmp(strn1,strn2,N)	Compares first n characters of two strings strn1 and strn2
strcmpi(strn1,strn2)	Compares two strings strn1 and strn2 without regard to case
strlwr(strn)	Converts the string strn to lowercase
strupr(strn)	Converts the string strn to uppercase
strdup(strn)	Returns a pointer to a string that is duplicate of the string strn
strchr(strn,chr)	Returns a pointer to the first occurrence of the character chr in the string strn. If chr is not available in strn then it returns NULL.
strrchr(strn,chr)	Returns a pointer the last occurrence of the character chr in the string strn. If chr is not available in strn then it returns NULL.
strstr(strn1,strn2)	Returns a pointer to the first occurrence of a string strn2 in the string strn1. If strn2 is not available in strn1 then it returns NULL.
strrev(strn)	Reverses the string strn
strset(strn,chr)	Sets all characters of the string strn to the character chr
strnset(strn,chr,N)	Sets first N characters of the string strn to the character chr

Space for learners notes

STOP TO CONSIDER:

It is necessary to include the header file 'string.h' to use mentioned string library functions.

Now consider the following programming statements.

```
char strn1[ ] = "Gauhati University";
char strn2[ ] = "Welcome to IDOL";
int slen;
slen = strlen(strn1);
printf("\n Length of the string stored in strn1 is = %d",slen);
slen = strlen(strn2);
printf("\n Length of the string stored in strn2 is = %d",slen);
strcpy(strn1,strn2);
printf("\n String stored in strn1 is =");
puts(strn1);
printf("\n String stored in strn2 is =");
puts(strn2);
printf("\n %d", strcmp(strn1,strn2));
```

Now the output of the above programming statements is:

```
Length of the string stored in strn1 is = 18
Length of the string stored in strn2 is = 15
String stored in strn1 is = Welcome to IDOL
String stored in strn2 is = Welcome to IDOL
0
```

The first line of the output gives the length of the string "Gauhati University" stored in the character array strn1. The second line of the output gives the length of the string "Welcome to IDOL" stored in the character array strn2. To estimate these lengths, the string library function strlen() is used.

In the above programming statements, string library function strcpy() is used to copy the string stored in strn2 to the string in strn1. As a result, string "Welcome to IDOL" replaces the string "Gauhati University" in strn1. Due to this, the third line of the output displays the string stored in strn1 which is "Welcome to IDOL". The fourth line of the output displays the string stored in strn2 which is also "Welcome to IDOL".

In the above programming statements, string library function strcmp() is also used to compare the strings stored in strn1 and strn2. Now at this moment, both strn1 and strn2 store the same string. So strcmp(strn1,strn2) returns 0 and as a result the fifth line of the output provide 0.

STOP TO CONSIDER:

In case of multi word strings, the blank spaces between two words are also considered as characters in estimation of lengths of the strings. For example, the length of the string "Gauhati University" is 18.

Space for learners notes

Example 4.13: Write a C program to read N number of employee names and perform sorting operation using Babble sort technique to arrange these names in alphabetical order. Use string library functions as required in the program.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{
        char Emp_names[50][100], temp[100];
        int N, i, j;
        clrscr();
        printf("\n Enter the total number of names = ");
        scanf("%d",&N);

        if(N > 50)
        {
                printf("\n Maximum 50 names is possible");
        }
        else
        {
                printf("\n Enter names of %d number of employees::", N);
                for(i = 0 ; i < N ; i++)
                {
                        printf("\n Enter %dth name = ",i+1);
                        gets(Emp_names[i]);
                }
                printf("\n The list of employee names before sorting is:\n");
                for(i = 0 ; i < N ; i++)
                {
                        puts(Emp_names[i]);
                        printf("\n");
                }
                for(i = 0 ; i < N-1 ; i++)
                {
```

Space for learners notes

```
                for(j = 0 ; j < N-1-i ; j++)
                {
                        if(strcmp(Emp_names[j]
                        ,Emp_names[j+1]) > 0)
                        {
                                strcpy(temp ,
                        Emp_names[j]);
                        strcpy(Emp_names[j] ,
                        Emp_names[j+1]);
                                strcpy(Emp_names[j+1] , temp);
                        }
                }
        }
        printf("\n The list of names after sorting is:\n");
        for(i = 0 ; i < N ; i++)
        {
                puts(Emp_names[i]);
                printf("\n");
        }
    }
    getch();
}
```

CHECK YOUR PROGRESS

3. Multiple choices

(a) In C programming, a string is terminated by

(i) '\0'

(ii) '\n'

(iii) A blank

(iv) None of the above

(b) A string is stored in a

(i) Character array

(ii) Integer array

(iii) Both (i) and (ii)

(iv) None of the above

(c) A string is initialized as

(i) char st1[ ] = "IDOL";

(ii) char st1 = "IDOL";

(iii) char st1[ ] = {'I','D','O','L','\0'};

(iv) Both (i) and (iii)

(d) char name[20] = "Welcome to IDOL" ;

name[7] = '\0';

printf("%s", name);

The output of the above statements is

(i) Welcome

(ii) Welcome t

(iii) IDOL

(iv) None of the above

(e) Which one of the following is appropriate for reading a multi word string ?

(i) printf( )

(ii) scanf( )

(iii) gets( )

(iv) Both (ii) and (iii)

(f) If strcmp(s1,s2) returns -12 then it means

(i) s1 and s2 are equal strings

(ii) s1 is greater than s2

(iii) s2 is greater than s1

(iv) None of the above

(g) char str[20] = "Welcome";

for(int i = strlen(str) – 1 ; i>=0 ; i—)

printf("%c", str[i]);

The output of the above statements is

(i) Welcome

(ii) emocleW

(iii) Welcom

(iv) Error message from compiler

(h) strcpy(s1,s2);

The above statement means

(i) Copies the string s2 to the string s1

(ii) Copies the string s1 to the string s2

(iii) Copies the first n characters of the string s2 to the string s1

4. State whether true or false

(a) The length of a string is equal to the subscript value of the position where the NULL character is stored in the character array.

(b) Strings cannot be initialized.

Space for learners notes

Space for learners notes

(c) A string with multiple words cannot be entered by scanf( ).

(d) strlwr( ) converts a string to its lower case.

(e) strcat(str1,str2) concatenates the string str1 at the end of the string str2.

4.12 SUMMING UP

We have learnt about arrays and strings from this unit. An array is a collection of similar type of data which are stored in consecutive memory locations. The declaration of an array has three parts, (a) type of the variable, (b)array name and (c) within brackets ([ ]) the size of the array means how many elements can be stored in the array.

Initialization of a one dimensional array can be implemented as follows:

```
int arr[5]={ 12,23,34,45,56};
```

Three ways of initializing a two dimensional array are given as follows:

```
➢    int arrtwo[2][3]= {
               {2, 18 , 7 },
               {43, 91, 1}
     };
➢    int arrtwo[2][3] = { 2 , 18 , 7, 43, 91 , 1 };
➢    int arrtwo[ ][3]= { 2 , 18 , 7, 43 , 91 , 1 };
```

Insertion and searching operation on an array can be performed with the name of the array and the subscript values.

String is a collection of some characters stored in a character array. A string is always terminated by \0 which is called NULL character. In general, gets( ) and scanf( ) are the two library functions used to read a string using standard input device. But scanf( ) is not capable of entering multi word strings. Multiple strings can be stored using two dimensional character array where the first subscript value of the array indicates the total number of strings and the second subscript value indicate the maximum length of each strings. This is also referred as array of strings.

Some useful library functions on strings are strlen( ), strcpy( ), strncpy( ), strcat( ), strlwr ( ), strupr( ), strcmp( ), strncmp( ), strcmpi( ), strdup( ), strchr( ), strrchr( ), strstr( ), strrev( ), strset( ), strnset( ). We have to include the header file 'string.h' to use these functions.

ANSWER TO CHECK YOUR PROGRESS

1. (a)(i) (b)(iv) (c)(iv) (d)(iii)(e) (ii)(f)(iv)(g)(iv)
2. (a) false (b) false (c) true (d) false (e) false
3. (a)(i) (b)(i)(c)(iv)(d)(i)(e) (iii)(f)(iii) (g)(ii) (h)(i)

4. (a) true (b) false (c) true (d) true (e) false

Space for learners notes

4.13 POSSIBLE QUESTIONS

1. Define array. Explain different types of array available in C programming. Give suitable examples.
2. Why concept of array is very important in programming?
3. Write a C program to construct a new array by merging two sorted integer array where the elements in the new array will also be sorted.
4. Write a C program to input a new element into an array at the position entered by the user.
5. Write a C program to calculate the summation of two integer arrays.
6. Write a C program to find out the number of even and odd numbers present in an integer array.
7. Write a C program to calculate the summation of all the even and odd numbers present in an integer array.
8. Write a C program to estimate the transpose of a input matrix.
9. Define string. Write down the differences between string and a character array.
10. Write a C program to check a string is palindrome or not.
11. Write a C program to reverse a string without using string library functions.
12. Write a C program to replace a particular character in a string by a character entered by the user.

4.14 REFERENCES AND SUGGESTED READINGS

➢ Kanetkar, Y. P., *Let us C*. BPB publications, 2016

➢ Gottfried B., Chhabra J. K, *Programming with C*, Schaum's Outlines Series, Tata McGraw Hill Publications, 2011

➢ Balagurusamy E., *Programming in ANSI C*, Tata McGraw Hill Publications, 2006

➢ Venugopal, K. R., Prasad S.R, *Mastering C*. Tata McGraw-Hill Education, 2007.

[illegible]

[illegible]

[illegible]

UNIT 5 FUNCTIONS

Space for learners notes

CONTENTS

5.1 INTRODUCTION

Functions are one of the important building blocks of C language. Function performs some tasks and it can be used in a program several times. The task of a function is pre-defined and so it can perform only the task for which it is designed. In the previous Units the most common functions encountered so far are: **printf()** and **scanf()** used for the purpose of output and input respectively.

5.2 OBJECTIVES

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

- understand why function is necessary and its advantages,
- understand the components of a function – function prototype, function definition and function call statement, return-type and argument(s) of a function,
- integrate a function into a program,

Space for learners notes

- know the differentiate between function call by value and function call by address,
- understand the concept of recursive function and its use.

5.3 WHAT IS A FUNCTION?

A **function** can be defined as a group of statements that perform a task. A function may be **called** (used) from anywhere in a program for any number of times. There are two categories of Functions and they are: **library functions** and **user-defined functions.**

Library Functions are those functions that are implemented in the C library. Prototypes of these functions are declared in several header files (files with extension. h) Library functions are grouped according to their uses and different header files are defined to hold their prototypes. Prototypes of Mathematical functions are declared in the file 'math. h'. Prototypes of functions dealing with strings are declared in the file 'string. h'. We can just use a function in our program whenever the tasks implemented in that function are to be performed. In the earlier Units you have come across different library functions, e.g.,

A **User Defined Function** is a function which is implemented by a user (mainly a programmer). So, now onwards we will discuss about **User Defined Functions. main()** is a special **user-defined function** which is mandatory to be implemented in every program as the execution of a program starts from it.

A program can have more than one user-defined function. Conventionally, functions are so designed that each one of them performs some independent task and later integrated in a single program.

STOP TO CONSIDER

In a statement of a C program if a word contains '( )' at the end then that word with '( )' is a function e.g., in the statement '**x=summation();**' then you remain sure that **summation()** is a function (may be user defined one or a library function).

The following are some advantages of using functions:

1. By defining functions a programmer can divide the entire task of the program into simple subtasks.
2. In a program, a task containing multiple statements may have to repeat a no. of times. In a function the task-code can be implemented and wherever in the program, the task is required to be performed, just use that function. Thus it reduces the size of the program instead of implementing the same set of code again and again in the program.
3. In C, a function defined for a particular task can be shared or used by different programs.

4. The advantage of implementing the repetitive code as a function is that whenever there is a requirement of modification in the task-code, you just modify the code inside the function and the modification will be reflected in every use of the function.

Space for learners notes

5.4 STRUCTURE OF A C FUNCTION

As already mentioned, a function is a group of statements, which perform a particular task; so there are rules for its declaration, definition and use. From the previous units, it is clearly understood how library (built-in) function can be used in our program to perform a particular task for which it is designed.

A user defined function can occur in a program in the following ways.

- Function Declaration (or Function prototype)
- Function Definition:: Formal Arguments
- Function Call:: Actual Arguments

The sections, to follow, will dwell upon the ways one by one. Consider the following program, ***Program-1***, where in the "**main()**" function, two integers are taken as input, then calls the user defined function "**sum()**" passing the two input integers and gets the summation in return. Then the summation is displayed on to the screen.

Program-1:

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

int sum(int, int);  ←——  Function Declaration or Prototype of function sum()

void main()  ←——  Definition of main() function
{
        clrscr();
        int a, b, result;
        printf("Enter a number:")
        scanf("%d", &a);
        printf("Enter another number:")
Calling sum() function [a, b are Actual Arguments.
        scanf("%d", &b);
        result=sum(a, b);  ←——  Calling sum() function [a, b are Actual Arguments.
        printf("The Summation=%d", result);
}
int sum(int x, int y)  ←——  Definition of function sum() [x, y are Formal Arguments.
```

Space for learners notes

```
{
        int s;
        s=x+y;
        return (s); ◄——— Definition of sum () function
}
```

5.5 FUNCTION DECLARATION

Like variables, the declaration of a function is necessary before it is used. The function declaration is formally known as **Function Prototype**. As the name **prototype** means **model/blueprint**, the function prototype means the blueprint for the function which basically describes/informs the compiler about the return type, function name and data-type of the **parameters/arguments** passed to it. Except the **"main()"** and the library functions, all other user defined functions should have a prototype.

Before going into detail about function let's first try to understand what is **parameter** or **argument** to a function. Consider the following statement,

```
        printf("The average=%f", avg);
```

where, **avg** is a variable of type float. In this **printf()** function call there are two data-items inside the ( ) brackets and they are **"The average=%d"** , **avg**. These two data-items are termed as **parameters** or **arguments** to the function **printf()**.

The syntax for function declaration/prototype is:

return-type function-name (parameter-type-list);

You may get confused with the word **return-type**, well, it means data-type!!! Yes, only valid data-types (built-in/user defined) can be used as return-type for a function. Return-type basically describes the kind of the data/value, a function can return. If, a function should return an integer data/value then the return-type should be one among **int/short int/long int**. If a function does not return any value then the return-type should be **void**.

function-name is the name given to a function. The rules for naming a function are the same as that for a variable.

The **parameter-type-list** is the list of data-types for the data/values to be passed to the function as parameters, each separated by ','. Sometimes, along with the dfita-type a parameter name is given for each of the parameters in the list but this is optional.

The function prototype statement should be terminated by a semi-colon. From the ***Program-1***, the declaration of the function **sum()** is illustrated in ***Fig-5.1***.

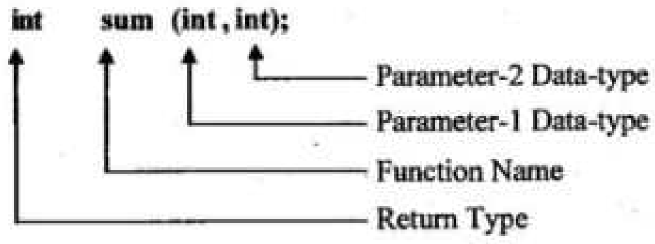


Fig-5.1

The above prototype tells the compiler that the function **sum()** takes two integers as arguments and returns an integer. The values/data which a function takes are called parameters or arguments. A function in C can take as many arguments as it needs (or no arguments at all) but can return only one value.

Example-1: Write a function prototype which takes a character as parameter and returns nothing.

```
void fun_1 (char);
```

or, we can write as

```
void fun_1 (char param);
```

Explanation:

- ✓ Since, the function returns nothing the return-type is mentioned as **void**.
- ✓ **fun_1** is the name of the function.
- ✓ **'char'** or **'char param'** is mentioned within **( )** brackets as it is said that the function takes a character as parameter.

Example-2: Write a function prototype which takes two floating point numbers as parameters and returns their summation.

```
float summation (float, float);
```

or, we can write as

```
float summation (float num1, float num2);
```

Explanation:

- ✓ Since it is said that the function returns the summation of two floating point numbers so, the return-type is mentioned as **float**.
- ✓ **summation** is the name of the function.
- ✓ **'float'**, **'float'** (or **'float num1'**, **'float num2'**) are mentioned within **( )** brackets as it is said that the function takes two floating point numbers as parameters.

STOP TO CONSIDER

If the definition of a particular function (e.g. **sum**) is mentioned after the function, from where the 1st function is called upon (e.g. **main**), then the function declaration/prototype 1st function (**sum**) is mandatory. But, if the 1st function (**sum**) is defined before the 2nd function (**main**) then the declaration/prototype is optional.

Space for learners notes

Space for learners notes

5.6 FUNCTION DEFINITION: FORMAL PARAMETERS & THE return STATEMENT

The definition of a function tells exactly what the function is written for. A **Function Definition** comprises of the **function name, return-type,** ***number of parameters with their types*** and ***its body***. A **Function** is a block of statements that will be executed when the function is called. The syntax for function definition is:

```
return-type function-name(type param1, type param2,.....)
{
        //Statements
}
```

In the above syntax,

- **return-type** is the **data type** of the value/data to be returned by the function.
- **function-name** is the name of the function.
- "**type param1, type param2, ...**" is the list of parameters to be passed to the function. Here, unlike in function prototype, name of the parameter along with its type is mandatory for each of the parameters for the function.

STOP TO CONSIDER

You may notice in the ***syntax*** of the ***function definition*** that at the end of the header statement there is no '**;**' mentioned (to end the statement). This so because, it is the start of the function definition and it will be marked end by **}**.

Consider the definition of function **sum()** in ***Program-1***, which is mentioned after the **main()**:

```
int sum ( int x, int y )
{
        int s;       }
        s=x+y;       }  Body of the function sum()
        return (s);  }
}
```

In the above function **sum()**,

- Apart from return-type as int and function-name as sum, the function heading also contains **x** and **y** as two parameters of type **int**.
- { starts the body of the function.
- 1st statement, within the body of the function, is the declaration statement of the variable **s**.
- in the 2nd statement, within the body of the function, the value of **x** and **y** are added and stored into variable **s**. x and y will contain the values that will be passed when the function will be called.

- The last statement, within the body of the function, will return the value of the variable **s** to the function from where the function **sum()** will be called upon.

Space for learners notes

5.6.1 Formal Parameters/Arguments:

Consider the definition of function **sum()** mentioned above. Here, **x** and **y** are used as parameters/arguments for the data/values to be passed when the function **sum()** will be called and they are called as **Formal Parameters** or **Formal Arguments.** Thus **Formal Parameters/Arguments** can be defined as the parameters/arguments that are mentioned in the definition of a function.

In the function **sum()** there is a variable **s** which is local to the function. But at the same time the parameters **x** and **y** also can be treated as local variables of the function **sum()**.

Now, when the **sum()** is called with the parameter-values, the function **sum()** starts executing and the two parameter-values are stored in **x** and **y** respectively.

STOP TO CONSIDER

The names of the formal and the actual arguments may be same or different but the data-types have to be same.

5.6.2 The return Statement:

In the definition of the **sum()** function of ***Program-1***, the **return** is used at the end of the function body along-with the variable **s** within **( )**. This means that the value of **s** is returned to the function from which the **sum()** is called upon, i.e. from within the **main()** function.

Basically, the **return** statement is used for two purposes:

- ✓ To return a value from a function(known as **called function**) to the function(known as **calling function**) from which the 1st function is called. For this, the ***value/value of the variable*** to be returned is mentioned at the end of **return** statement within **( )**. The **( )** are optional i.e. we may not use the **( )** while mentioning the ***value/value of the variable***.
- ✓ To end the execution of the called function and transferring the control back to the calling function. So, in this situation along-with the **return** statement no value/value of a variable is mentioned.

STOP TO CONSIDER

The main limitation in the use of the return statement is that you can use it to return only one value.

Space for learners notes

CHECK YOUR PROGRESS-1

1. What is function?
2. What do you understand by User Defined Function?
3. What is Function Prototype?
4. What is Formal Arguments or Formal Parameters?

State TRUE or FALSE:

5. A function always returns a value.
6. A function may or may not have parameters.
7. **return** statement is used to end a function.
8. Return type of a function can be **void**.
8. ASCII stands for ______________________.
9. Division by zero (0) is a ______________ error.

5.7 FUNCTION CALL: ACTUAL PARAMETER

Basically, the **Function Call** means the use of a function in a program where the function may be a library function or a user defined function.

Consider the **main()** function in ***Program-1***.

```
void main()
{
        clrscr();
        int a, b, result;
        printf("Enter a number:");
        scanf("%d", &a);
        printf("Enter another number:");
        scanf("%d", &b);
        result=sum(a, b);
        printf("The Summation=%d", result);
        getch();
}
```

Here in the **main()** function,

- ➢ two ***integer inputs*** are stored into the variable **a** and **b** using two **scanf()** functions,
- ➢ in the statement,

  result=sum(a, b);

  the function **sum()** is used(or called) with the variable **a** and **b** mentioned within ( ).

Actual Parameters/Arguments:

Consider the definition of function **main()** mentioned above. As mentioned earlier, **a** and **b** are the two variables passed to the function **sum()** when it is called. Here, **a** and **b** are known as **Actual Parameters** or **Actual Arguments**. Thus **Actual Parameters/Arguments** can be defined as the data/value those are passed to a function [mentioned within ( )] when it is called. These arguments are the actual arguments to be worked with.

Now, let's discuss about how many ways parameters can be passed to functions. There are two ways of passing parameters to a function (may be a library or user defined function) and they are:

- Call By Value
- Call By Address

In terms of parameter/argument passing, the above mentioned ways are also known as:

- Pass By Value
- Pass By Address

5.8 CALL BY VALUE

In this parameter passing technique only the values are passed as parameters. In the above mentioned **main()** function while calling the function **sum()**, the variables **a** and **b** are passed as arguments(actual). Thus, the values stored in **a** and **b** are passed to the function **sum()** when it is called. As in this method of function call, the values are passed as parameters to the called function, hence this method of function call is known as **Call by Value** or **Pass by Value**.

Consider the ***Program-2***. Here in this program **sum()** is a user defined function, same as mentioned in **Program-1**. But the **main()** function(in ***Program-2***) is different from the **main()** in ***Program-1***.

Program-2: Demonstration of Call By Value.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        clrscr();
        int result;
        result=sum(5, 2);
        printf("The Summation=%d", result);
        getch();
}
```

Space for learners notes

```
int sum ( int x, int y )
{
        int s;
        s=x+y;
        return (s);
}
```

Output:

```
The Summation=7
```

Explanation:

When the program runs, execution starts for the **main()** function. Statements in the **main()** start executing one-by-one. When, the following statement executes,

result=sum(5, 2);

- ✓ the **sum()** function is called with actual arguments **5, 2**.
- ✓ The execution control is now transferred to the function **sum()** with the values **5** and **2**, [passed from the **main()**] those eventually stored in the **x** and **y** respectively.
- ✓ Inside **sum()**, the values stored in **x** and **y**, i.e. **5** and **2** respectively, are added and then assigned to **s**.
- ✓ At the end of **sum()**, the **return** statement returns the value of **s**, i.e. **7**, and transfers the execution control back to the above statement in the **main()** function from where the function **sum()** was called.
- ✓ The returned value 7[value of **s** in **sum()**] is assigned to the variable **result** in the **main()**.
- ✓ Now, the **printf()** statement is executed and the output is produced onto the screen.

The above explanation is depicted in **Fig-2**.

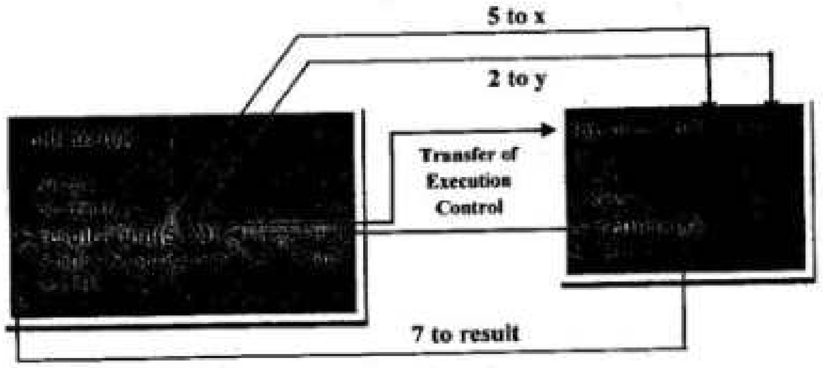


Fig-5.2: Illustration of function calling, parameter passing & returning a value (Program-2)

Now, let's try to explain the execution of ***Program-1***. Consider that **10** and **20** are the given input for the variable **a** and **b** respectively.

Output:

```
Enter a number:10
Enter another number:20
The Summation=7
```

Explanation:

In the **main()** function, when the following statement executes,

result=sum(a, b);

- ✓ the **sum()** function is called with actual arguments **10, 20** as they were the given inputs for **a** and **b** using the two **scanf()** functions.
- ✓ The execution control is now transferred to the function **sum()** with the values of **a** and **b**, i.e. **10** and **20**, [passed from the **main()**] those eventually stored in the **x** and **y** respectively.
- ✓ Inside **sum()**, the values stored in **x** and **y**, i.e. **10** and **20** respectively, are added and then assigned to **s**.
- ✓ At the end of **sum()**, the **return** statement returns the value of **s**, i.e. **30**, and transfers the execution control back to the above statement in the **main()** function from where the function **sum()** was called.
- ✓ The returned value **30** [value of **s** in **sum()**] is assigned to the variable **result** in the **main()**.
- ✓ Now, the **printf()** statement is executed and the output is produced onto the screen.

The above explanation is depicted in **Fig-3**.

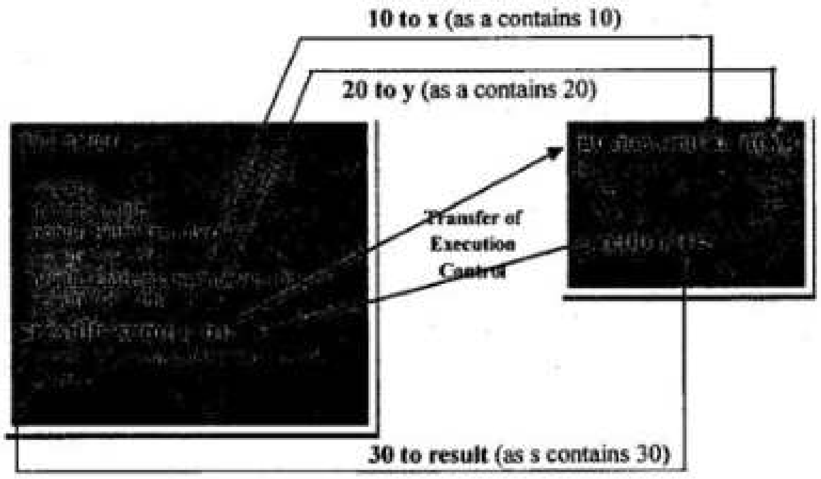

10 to x (as a contains 10)
20 to y (as a contains 20)
Transfer of Execution Control
30 to result (as s contains 30)

Fig-5.3: Illustration of function calling, parameter passing & returning a value (Program-1)

Space for learners notes

5.9 CALL BY ADDRESS

Before discussing the topic, let's first know about **Address**. The main memory is addressable which means that any object which resides in it has proper **Address**. The addresses of main memory are just some unsigned whole numbers. The variables can be accessed by their names as well their addresses. We already know that the **Address-of-Operator (&)** is used to get the Address of the memory location used by a variable. Now the term, **Call By Address,** means that while calling a function we need to pass the location address. And in the called function the data-item in that location can be accessed using the address passed. So, in the definition of the function the argument should be such that it can hold an address of a location. For this let's discuss about **Pointers.**

What is a Pointer?

Apart from address-of-operator we also have to know about another kind of variable that can store the address of a memory location. This kind of variable is known as **Pointer Variable.** Thus, a **Pointer Variable** can be defined as the variable which can store an address of a memory location. The declaration syntax of a pointer variable is the same as that of a simple variable with a little difference and this difference is the use of '*' symbol before the variable name. For example,

```
int *a;
```

The above statement is pointer variable declaration statement which means that '**a**' is a pointer variable which can store the address of an integer variable. Consider the following statements, which clearly describe the use of a pointer variable.

```
int x = 100;
int *p;
p = &x;
```

- ✓ The first statement declares an integer variable **x** as well as 100 is assigned to it.
- ✓ Second statement declares an integer pointer variable **p**.
- ✓ In the third statement, the address of the variable **x** is assigned to the pointer variable **p**.

Fig-4 depicts the scenario after execution of the third statement while considering the address of **x** is **100001** and address of **p** is **100010**.

Fig-5.4

Thus, **p** contains the address of **x** and this is like variable **p** is pointing to the variable **x**. Because of the above fact such kind of variable is termed as **Pointer** variable. Therefore the variable **x** can be accessed using **x** itself and **p**. These are illustrated in the following display statement.

```
printf("The value of x = %d", x);
printf("\nThe value of x = %d", *p);
```

Output:

```
The value of x = 100
The value of x = 100
```

Explanation:

- ✓ The first **printf()** will display the value of **x** using itself.
- ✓ The second **printf()** will display the value of **x** using **p**. So, here ***p** means the ***value at the memory location*** whose **address** is stored in **p**. In other words, ***p** means the **value at the location pointed by p.**

Now, you may think of that if ***p** and **x** means the same location then what does **p** mean??? The p contains the address of x, i.e. 1000001. So, the following statement will display the content of **p**, i.e. the address of **x**, and which is **1000001**.

```
printf("The address of x (using p) = %u", p);
```

Here, the **format specifier %u** is used, as the address of a memory location is an **unsigned integer.** The **address** value will be displayed as a **hexadecimal number** instead as a **decimal number**.

The following statement will also display the address of x.

```
printf("The address of x (using x itself) = %u", &x);
```

Hope!!! that you have a got a clear idea about addresses and pointers. Now, let's come to the topic of discussion i.e. **Call By Address** or **Pass By Address.**

But in this case, **Call By Address**, the items to be passed to the called function (while calling) are the addresses of the locations, i.e. addresses of the variables. And the items to be taken by the function (while defining) are pointers.

To understand this, let's consider the ***Program-3*** which is a modification of ***Program-1***.

Program-3: Demonstrating the Call By Address/Pass By Address.

```
#include<stdio.h>
#include<conio.h>
int sum(int, int);
void main()
{
        clrscr();
        int a, b, result;
        printf("Enter a number:")
        scanf("%d", &a);
        printf("Enter another number:")
```

Space for learners notes

Space for learners notes

```
        scanf("%d", &b);
        result=sum(&a, &b);
        printf("The Summation=%d", result);
        getch();
}
int sum(int *x, int *y)
{
        int s;
        s=*x + *y;
        return (s);
}
```

The statements which are modified are marked as bold in the above program. The output of this program will be the same as the output of the ***Program-1*** if same inputs are considered for **a** and **b.**

Let's discuss the execution of the program in brief. Execution starts form the **main()** function.

In the **main()** function:

✓ Using **scanf()** functions, two inputs are taken and stored into the variables **a** and **b**.

✓ In the statement, marked as **bold**, the **sum()** function is called and **&a** (i.e. address of **a**) and **&b** (address of **b**) are passed as arguments.

Now, execution control is transferred to the **sum()** function. As the arguments from the **main()** function are addresses of **a**, **b** therefore the formal arguments in the definition of the function **sum()** are declared as **integer pointers**.

In the **sum()** function:

✓ The addresses of **a** and **b** passed from **main()** are stored into the pointer variables **x** and **y** respectively, i.e. **x** and **y** are pointing to variables **a** and **b** in **main()** function.

✓ In the statement, **s=*x+*y; *x** and ***y** means the variables **a** and **b** of the **main()** function. So, the values of **a** and **b**, i.e. pointed by **x** and **y**, are added and stored into the variable **s.**

✓ The value of **s** is returned to **main()**.

Now, the execution control is transferred back to **main()** function. In the **main()** function:

✓ The returned value from **sum()** is then stored into the variable **result**.

✓ The value of the **result** variable is then displayed using **printf()**.

5.10 TYPES OF USER DEFINED FUCNTIONS

Space for learners notes

We have already discussed the basic concepts related to Function Prototype, Function Definition and Function Calling. Functions can be categorized in to different types depending on the return type and the arguments. These are:

5.10.1 Function With No Argument And No Return Value:

Function with no argument means no argument list within **( )** in a function definition and hence no argument in function calling and function prototype.

Function with no return value means **void** as return type of the function.

Functions of this type, defined by user are very rare as there is no communication made within this kind of function and the caller of the function. But there are built-in/ library functions of this type e.g, **clrscr()**, **getch()** etc.

Program-4 is an example of a user defined function of this kind.

Program-4: Write a function which will display the nos. from 1 to 10. Also write the **main()** function.

```
#include<stdio.h>
#include<conio.h>
void displaynums();

void main()
{
        clrscr();
        displaynums();
        getch();
}
void displaynums()
{
        int i;
        printf("The numbers from 1 to 10 are:n\");
        for (i=1; i<=100; i++)
        {
                printf("%d ", i)
        }
}
```

Space for learners notes

Output:

```
The numbers from 1 to 10 are:
1 2 3 4 5 6 7 8 9 10
```

5.10.2 Function With Argument(s) But No Return Value:

Function with arguments means there may be one or more than one arguments in a function definition and hence argument list in function calling and function prototype.

Function with no return value means **void** as return type of the function.

This is illustrated in ***Program-5*** mentioned below.

Program-5: Write a function which will take two numbers as arguments and displays the nos. between them. Also write the **main()** function.

```
#include<stdio.h>
#include<conio.h>
void displaynums(int start, int end);
void main()
{
        clrscr();
        displaynums(100, 500);
        getch();
}
void displaynums(int start, int end)
{
        int i;
        printf("The numbers from %d to %d are:n\", start, end);
        for (i=start; i<=end; i++)
        {
                printf("%d ", i)
        }
}
```

Output:

```
The numbers from 1 to 10 are:
1 2 3 4 5 6 7 8 9 10 ........................... 500
```

5.10.3 Function With Argument(s) and Return Value:

Function with arguments means there may be one or more than one arguments in a function definition and hence argument list in function calling and function prototype.

Function with return value means data types other than **void** as return type of the function.

This is illustrated in ***Program-6*** mentioned below.

Program-6: Write a function which will take two numbers as arguments and returns the summation of the nos. between them. Also write the **main()** function.

```
#include<stdio.h>
#include<conio.h>
int summation(int start, int end);
void main()
{
        clrscr();
        int x, y, result;
        printf("Enter the starting no: ");
        scanf("%d", &x);
        printf("Enter the ending no: ");
        scanf("%d", &y);
        result=summation(x, y);
        printf("The summation of the numbers from %d to %d= ",
                result);
        getch();
}
int summation(int start, int end)
{
        int i, sum=0;
        for (i=start; i<=end; i++)
        {
                sum = sum + i;
        }
        return (sum);
}
```

Consider the input given to x and y in the main() function are 1 and 10 respectively.

Output:

```
Enter the starting no: 1
Enter the ending no: 10
The summation of the numbers from 1 to 10= 55
```

Space for learners notes

Space for learners notes

5.10.4 Function With No Argument But Return Value:

In this type of functions, no arguments are declared in the definition of the functions but return type should be mentioned related with the type of the value to be returned from them.

This is illustrated in ***Program-7*** mentioned below.

Program-7: Write a function which will return the summation of the nos. from 1 to 10. Also write the **main()** function.

```
#include<stdio.h>
#include<conio.h>
int summation();
void main()
{
        clrscr();
        int result;
        result=summation();
        printf("The summation of the numbers from 1 to 10= ",
                result);
        getch();
}
int summation()
{
        int i, sum=0;
        for (i=1; i<=10; i++)
        {
                sum = sum + i;
        }
        return (sum);
}
```

Output:

```
The summation of the numbers from 1 to 10= 55
```

5.11 PASSING ARRAY TO FUNCTION

In a C, like variables/elements, we can also pass array to a function as parameter. We know that the syntax of declaring an one dimensional array is:

data-type array-name [size];

The same syntax is used while declaring an array as argument in a function definition but with a little modification and which is:

```
void process(int a[])
{
        Statements.......
}
```

But in this case, can the declaration of the array-argument be sufficient to carry the information about the size of the arrary passed to when the function is called. Definitely not!!! So, for the size information we also need to declare an integer argument along-with the declaration of the array-argument. Thus the definition of the above function should be

```
void process(int a[], int n)
{
        ...................
        ...................
}
```

where the argument **n** is for the size of the array-argument **a[]**.

Now, the prototype of the function **process()** may be written as:

```
        void process(int a[10]);
Or,     void process(int a[]);
Or,     void process(int []);
```

During the call to the function **sum()**, we have to pass two items as arguments: the **name** of the ***actual array*** as 1st argument and the ***no. of elements*** present in the array as the 2nd argument.

```
void main()
{
        ....................
        process(arr, 20);
        ....................
}
```

STOP TO CONSIDER

In a function, where an array is passed as parameter, if we modify the elements in the array this means that the modification is actually done in the original array.

Program-8: Write a function in C to calculate the summation of the nos. in an integer array passed as parameter to it. Also write the **main()** function.

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

Space for learners notes

Space for learners notes

```
int arraySUM(int [], int);
void main()
{
        clrscr();
        int a[100], n, i, result;
        printf("How many nos. you want to enter: ");
        scanf("%d", &n);
        printf("Enter the nos:\n");
        for (i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
        }
        result = arraySUM(a, n);
        printf("The Summation: %d", result);
        getch();
}
int arraySUM(int arr[], int size)
{
        int i, sum=0;
        for (i=0; i<size; i++)
        {
                sum = sum + arr[i];
        }
        return (sum);
}
```

In the ***output***, except the last line all the other lines contain the input(s).

Output:

```
How many nos. you want to enter: 5
Enter the nos:
2
4
8
45
1
The Summation: 60
```

Since, in Program-8, it was not mentioned clearly that after calculating the summation in the function what is to be done, i.e. whether the function should return the summation or display after calculation within it. So, the above program can also be written as:

```
#include<stdio.h>
#include<conio.h>
void arraySUM(int [], int);
void main()
{
        clrscr();
        int a[100], n, i, result;
        printf("How many nos. you want to enter: ");
        scanf("%d", &n);
        printf("Enter the nos:\n");
        for (i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
        }
        arraySUM(a, n);
        getch();
}
int arraySUM(int arr[], int size)
{
        int i, sum=0;
        for (i=0; i<size; i++)
        {
                sum = sum + arr[i];
        }
        printf("The Summation: %d", sum);
}
```

This program is written in such a way that if the inputs are the same as above then the output will be exactly the same.

Program-9: Write a function in C to find maximum of the nos. in the integer array passed as parameter to it. Also write the **main()** function.

```
#include<stdio.h>
#include<conio.h>
void findmax(int [], int);
void main()
{
        clrscr();
        int a[100], n, i, result;
        printf("How many nos. you want to enter: ");
```

Space for learners notes

Space for learners notes

```
        scanf("%d", &n);
        printf("Enter the nos:\n");
        for (i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
        }
        findmax(a, n);
        getch();
}
int arraySUM(int arr[], int size)
{
        int i, max=arr[0];
        for (i=1; i<size; i++)
        {
                if(arr[i]>max)
                {
                        max=arr[i];
                }
        }
        printf("The Maximum No: %d", max);
}
```

In the ***output***, except the last line all the other lines contain the input(s).

Output:

```
How many nos. you want to enter: 5
Enter the nos:
2
4
8
45
1
The Maximum No: 45
```

5.12 PASSING STRING TO FUNCTION

Like **array**, a **string** can also be passed as parameter to a function. The syntax of defining a function which takes a **string** as parameter is:

```
        return_type function_name(char string_array[])
        {
```

```
        Statements.........
    }
```

In the syntax, you can notice that unlike functions no argument related to size of the string is declared. This is because, since string is an array of characters but marked the end of data by the '\0' character. Also there is a library function, **strlen()**, declared in **string. h** which returns the length of the string passed to it. So, to get the size of the string, i.e., the length, a programmer has two options: either by counting the characters in the string one-by-one fill the character '\0' is seen or just by calling the **strlen()** library function.

It is not true that we cannot pass the length of the string along-with the string itself to function. We can pass the length of the string also as parameter but it is useless because of the reason mentioned above.

The ***Program-10*** will help you to understand the above facts.

Program-10: Write a function in C which returns the no. of vowels in the string passed as parameter to it. Also write the **main()** function where the string to be passed is to be taken as input.

```
#include<stdio.h>
#include<conio.h>
int vowelnos(char []);
void main()
{
        clrscr();
        char strname[100];
        int no;
        printf("Enter a String: ");
        scanf("%s", strname);
        no=vowelnos(strname);
        printf("The no. of Vowels= %d", no);
        getch();
}
int vowelnos(char n[])
{
        int i, count;
        for (i=0, count=0; n[i]!='\0'; i++)
        {
                if(n[i]=='a' || n[i]=='e' || n[i]=='i' || n[i]=='o' || n[i]=='u' || n[i]=='A'
|| n[i]=='E' || n[i]=='I' || n[i]=='O' || n[i]=='U')
                {
                        count++;
```

Space for learners notes

Space for learners notes

```
            }
        }
        return count;
}
```

Suppose in the main() the input string is WELCOME TO IDOL.

Output:

```
Enter a String: WELCOME TO IDOL
The no. of Vowels= 6
```

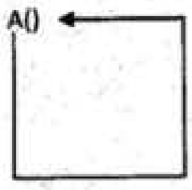
In the definition of the function **vowelnos()** there is only one parameter but it is sufficient because the parameter to be passed is a string.

5.13 RECURSIVE FUNCTION

Till now, all user defined functions discussed above are called by the **main()**, which is also a function. So, any function can be called from any other functions. Apart from this, C also enables a function to call itself. This technique is called **Recursion. Recursive Function** is a function which calls itself directly. Consider the following ***Fig-5.5***.

```
A()
void A()
{
        ...............
        A();
        ...............
}
```

(a)

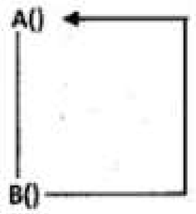


(b)

Fig-5.5

In the definition of A(), there is a statement which calls the function itself [***Fig-5.5(a)***]. The function **A()** is a **Resursive Function**. This is shown graphically in ***Fig-5.5(a)***. There is another term known as **Indirect Recursion. Indirection Recursion** occurs when one function calls another function that then calls the first function. The following is an example of Indirect Recursion. Consider the following ***Fig-5.6***.

```
void A()
A()
B()
{
      B();
}
void B()
{
      A();
}
```



(a)   (b)

Fig-5.6

In the definition of **A()**, there is a statement which calls the function **B()** and in the definition of **B()** a statement calls the function **A()** [***Fig-5.6(a)***]. This is called **Indirect Recursion**. This is shown graphically in ***Fig-5.6(a)***.

Program-11: Write a C program which calculates the summation of the no. from 1 to 3 using recursive function.

```
#include<stdio.h>
#include<conio.h>
int calc(int n);
void main()
{
      printf("\nThe Summation = %d", calc(3));
}
int calc(int n)
```

Output:

```
The Summation = 6
```

Explanation (Graphically):

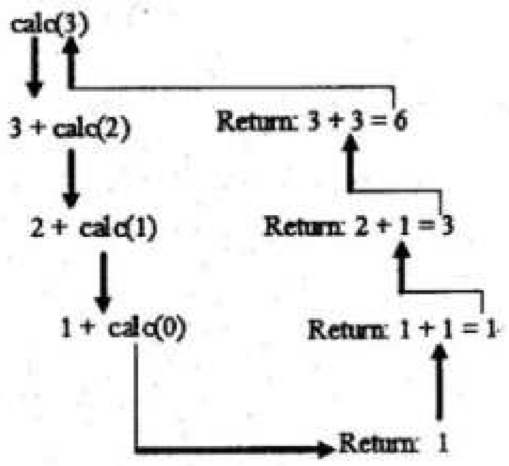


Fig-5.7

Space for learners notes

In the above ***Program-11***, when, in the **main()**, the **calc()** is called with argument value **3** and it calls itself as with argument value **2** then with argument value 1 then with argument value 0 and now the function **calc()** returns with a value of **1**. This return action makes the previous function calls to return with appropriate values. The function which is called last is the one to return first.

Program-12: Write a C program which calculates the factorial of a no. using recursive function.

```
#include<stdio.h>
#include<conio.h>
int fact(int n);
void main()
{
        printf("\nThe Factorial of 3 = %d", fact(3));
}
int fact(int n)
{
  if(n==0)
        return 1;
  else
        return n*fact(n-1);
}
```

Output:

```
The Factorial of 3 = 6
```

Explanation (Graphically):

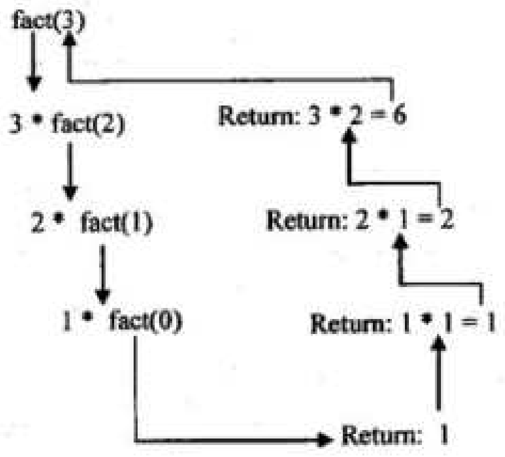


Fig-5.8

CHECK YOUR PROGRESS-2

9. What do you understand by Actual Parameters?

10. What are the two ways of calling a function?

11. What is Pointer? How is it related to Call By Address?

12. What do you understand by ***function with no argument and no return value***?

State True or False:

13. ***p** means the location whose address is stored in **p**.

14. In Call By Value the addresses of the storage locations are passed.

15. An array cannot be passed as parameters to a function.

Fill-in the Blanks:

16. ____________ function calls itself.

17. While passing an array to a function, the _______ of the array should also be passed.

18. The format specifier for address of a memory location is _______.

Space for learners notes

5.14 SUMMING UP

This Unit contains discussions about Functions; its different types and its advantages.

As discussed in this Unit, C library is consists of a no. of header files, e.g., **stdio.h**, **conio.h** (in Windows only), **string.h** etc. These header files contain different functions for different purposes.

The concept of User Defined Functions is also discussed in this Unit. This type of functions are very useful to cater out own needs while writing C programs. The structure of a user defined function is discussed very clearly in this Unit with its different parts.

The two ways of calling a function, Call By Value and Call By Address, are discussed very clearly with examples. Though the Pointer concept is to be discussed mainly in Unit-7, for Call By Address which requires to work with memory address, so a basic overview on pointers is given in this Unit.

The Unit is trying to give basic ideas related with function definition, function call, and function prototype. An important fact related to function prototype is that if a function is defined before another function [generally **main()**] form where the first function is called upon then declaration of the prototype for the first function is optional. But if the first function is defined after the function (from where the first function is called), then the prototype of first function is to be declared.

The types of user defined functions in relation with arguments and return type are tried to discuss in detail with the help of examples. Also functions with array as arguments are also discussed with examples.

Space for learners notes

5.15 ANSWERS TO CHECK YOUR PROGRESS

1. A function can be defined as a group of statements that perform a task. A function may be called (used) from anywhere in a program for any number of times.
2. User Defined Function is a function which is implemented by a user (mainly a programmer). main() is a special user-defined function from where the execution of a program starts.
3. Like variables, the declaration of a function is necessary before it is used. The function declaration is formally known as Function Prototype.
4. The Formal Parameters/Arguments can be defined as the parameters/arguments that are mentioned in the definition of a function.
5. False
6. True
7. False
8. True
9. Actual Parameters/Arguments can be defined as the items/values those are passed to a function when it is called.
10. There are two ways of calling a function and they are namely Call By Value/ Pass By Value and Call By Address/Pass By Address.
11. A Pointer or pointer Variable can be defined as the variable which can store an address of a memory location.
12. Function with no argument means no argument list within ( ) in a function definition and hence no argument in function calling and function prototype also. Function with no return value means void as return type of the function.
13. True
14. False
15. False
16. Recursive
17. name
18. %u

5.16 POSSIBLE QUESTIONS

Short answer type questions:

1. What are the main advantages of using functions?
2. Write down the categories of functions?
3. What is the syntax of defining a function?
4. Why does the **return** statement used in a function?

5. What is the difference between a Function Definition and Function Prototype?
6. What is the purpose of the return statement.

Long answer type questions:

7. Write down the syntaxes for Function Prototype, Function Definition and Function Call.
8. Differentiate between Call By Value and Call By Address.
9. How can you relate pointers with Call By Address? Discuss with the help of an example.
10. How can array be passed to a function? Discuss with the help of an example.
11. What is a Recursive Function? Discuss.
12. Write a C function to define a function which swaps the two arguments passed as parameters so that swapping reflects in the actual parameters.
13. Write a recursive function to evaluate

    S = 1 + 2 + ... + n
14. What is a Recursive Function? Discuss.

5.17 FURTHER READINGS

1. Kernighan, B. W., & Ritchie, D. M. (2006). *The C programming language*.
2. Balagurusamy, E. (2012). *programming in ANSI C*. Tata McGraw-Hill Education.
3. Kanetkar, Y. P. (2016). *Let us C*. BPB publications.
4. Schildt, H., & Turbo, C. (1992). C: The Complete Reference. *McGraw-Hill, Inc., New York, NY, USA*, *4*, 39.

Space for learners notes

UNIT 6 STRUCTURES AND UNIONS

Space for learners notes

CONTENTS

6.1 INTRODUCTION

We have seen in the earlier chapter that arrays are used to store a set of data items but the main disadvantage of using arrays is that all the items stored in the array are of the same data type. What if the programmer wants to store items of different data type? Fortunately C programming supports the concept of grouping items of different data types into a single logical unit called *structures*. The items within a structure are called members of the structure. The members can be of integer, floating point, character or any other data type. The members can also be an array, pointer or even structures. Structures are powerful concept that help us to manage complex data in a meaningful way which otherwise would not have been possible to represent with existing built in data type like integers, float, char etc.

Some examples of structures are:

• book:	title, author, publisher, edition, price
• vehicle:	make, model, year, price
• employee:	id, email, year_of_joining, date_of_birth, designation, salary
• time:	hour, minute, second, millisecond
• fruit:	weight, colour, taste

Space for learners notes

6.2 OBJECTIVES

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

- Understand the basic concept of structure and its advantage over arrays.
- Learn how to define a structure in a program.
- Learn how structure variables are declared and initialized in a program.
- Write programs to access individual members of a structure variable as well as perform operations on them.
- Learn how to pass structure values to function as arguments.
- Understand the way of declaring array of structures and use it in program.
- Understand the basic concept of union and its distinction with structures.
- Learn how to define a union in the program.
- Learn how union are declared and initialized in a program.
- Write programs to access and perform operations on individual members of a union.

6.3 DEFINING A STRUCTURE

A structure must be defined first before structure variables can be declared and initialized. The general syntax for defining a structure is shown in Fig. 6.1.

```
struct    structure_name
{
data_type            member 1;
data_type            member 2;
data_type            member 3;
.........................................
.........................................
data_type            member n;
};
```

Fig. 6.1: Syntax for defining a structure

Where

- *struct* is a keyword
- the definition must end in semicolon
- each member is declared separately with its data type and ends in semicolon
- *structure_name* is used to declare structure variables in the program

Let us consider an example as shown in Fig 6.2 to understand the process of defining a structure. We consider an employee database to store id, email, year_of_joining, date_of_birth, designation and salary for the all the employees of an organization. For this we define a structure with the name *employee* to hold the above mentioned information as follows:

```
struct   employee
{
int              id;
char             email[20];
char             year_of_joining[10];
char             date_of_birth[10];
char             designation[10];
float            salary;

};
```

Fig. 6.2: Example of an employee structure

Similarly we can define a structure *vehicle* to store make, model, year and price as in Fig. 6.3 and structure *date* to store day, month, year, week as in Fig. 6.4.

```
struct   vehicle
{
char             make[10];
char             model[10];
char             year[10];
float            price;
};
```

Fig. 6.3: Example of an vehicle structure

```
struct   date
{
char             day[2];
char             month[10];
char             year[4];
int              week;
};
```

Fig. 6.4: Example of an date structure

One more way of defining and declaring a structure is by using the keyword *typedef* as in Fig. 6.5.

Space for learners notes

Space for learners notes

```
typedef struct
{
data_type          member 1;
data_type          member 2;
data_type          member 3;
.......................................
.......................................
data_type          member n;
}type_name;
```

Fig. 6.5: Syntax for defining a structure

The *type_name* represent structure definition and it can be used to declare structure variables.

```
typedef struct
{
char        day[2];
char        month[10];
char        year[4];
int         week;
}date;
```

Fig. 6.6: Example of a structure date using typedef

STOP TO CONSIDER:

Use of structure_name is optional as in the example below where x,y,z are structure variables. However it is not recommended to use such a definition because without structure name it cannot be used in future declarations.

```
struct
{
// members
} x,y,z;
```

6.4 DECLARATION AND INITIALIZATION OF STRUCTURES

In C programming declaration of structure variable is same as declaration of any other variable of other data types. As for example after defining structure employee and structure vehicle as in the above section we can declare structure variables as:

- struct employee emp1, emp2, emp3;
- struct vehicle v1, v2, v3, v4;

where emp1, emp2, emp3 are variables of type *struct employee* and v1, v2, v3, v4 are variables of type *struct vehicle*.

It is also possible to combine structure definition and structure declaration in the same statement as given in Fig. 6.7.

```
struct   vehicle
{
char            make[10];
char            model[10];
char            year[10];
float           price;
}v1, v2, v3, v4;
```

Fig. 6.7: Example of a structure vehicle definition and declaration

Or *typedef* can be used to declare structure variables as in Fig. 6.8.

```
typedef struct
{
char            make[10];
char            model[10];
char            year[10];
float           price;

} vehicle;
```

Fig. 6.8: Example of a structure vehicle definition using typedef

The structure declaration can be

- vehicle v1, v2, v3;

During compilation memory is reserved for structure variables. No memory is allocated to members of a structure when defined but memory is allocated when structure variables are declared. In the example given in Fig. 6.9 three structure variables v1, v2 and v3 are created. The structure *vehicle* consists of four members *make, model, year* each of 10 bytes and member *price* of 4 bytes making it a total of 34 bytes. Thus the compiler allocates 34 bytes to each of the variable v1, v2 and v3. The output of the program is shown in Fig. 6.10.

```
#include<stdio.h>
struct vehicle{                                   //Definition
char make[10];
```

Space for learners notes

Space for learners notes

```
char model[10];
char year[10];
float price;
};

void main()
{
struct vehicle v1,v2,v3;                                  //Declaration
printf("Address of structure varibles:\n v1=%u \n v2=%u \n v3=%u
\n",&v1,&v2,&v3);
printf("size of structure in bytes = %d", sizeof(v1));            //Size of
structure
}
```

Fig. 6.9: Example of a structure vehicle definition using typedef

OUTPUT

```
Address of structure variable:
v1=65492
v2=65458
v3=65424
size of structure in bytes=34
```

Fig. 6.10: Output of the program in fig. 6.9

STOP TO CONSIDER

When a structure variable is declared it may contain slack/unoccupied bytes between the members. Therefore even if values of members of two structure variables are same their comparison may not be equal.

In C programming structure variables can be initialized during compile time. A structure can be initialized only during the structure variable declaration. It should be noted that a structure does not allow initialization of the individual members in a structure definition. For example initialization as shown in Fig. 6.11 is invalid.

```
struct vehicle
{
char a='W';
int b=123;
float c=12.4;
}
```

Fig. 6.11: Invalid way of initialization in structure definition

STOP TO CONSIDER

Following rules must be followed while initializing a structure variable:

- Members inside structure definition cannot be initialized
- The initialization values inside the braces must match the order of members in structure definition
- It is permitted to partially initialize i.e. first few members can be initialized remaining members only at the end can be left blank. By default members of type integer or float are assigned 0 and members of type character are initialized to '\0' if they are not initialized.

Space for learners notes

The example in Fig 6.12 shows three different ways of initializing structure variable v1, v2 and v3. It should be noted that the structure variable v1 is initialized outside main function with the values {"FORD","FIGO","2015", 6.8} representing respectively make, model, year and price. Similarly the structure variable v2 is initialized inside main with values {"HONDA","CITY","2018",11.5} whereas the structure variable v3 is partially initialized to {"MARUTI","BALENO"} representing respectively make and model. It should be noted that the values '\0' and 0 are assigned respectively to year and price of structure variable v3 by default.

```
#include<stdio.h>
struct vehicle{
char make[10];
char model[10];
char year[10];
float price;
}v1={"FORD","FIGO","2015",6.8};                        //initialization

void main()
{
struct vehicle v2={"HONDA","CITY","2018",11.5} ;  //initialization
struct vehicle v3={"MARUTI","BALENO"};              //initialization
}
```

Fig. 6.12: A sample example showing ways of initialization

6.5 ACCESSING THE MEMBERS OF A STRUCTURE

To access or assign values to any member of a structure variable, member operator '.' is used. This operator is also known as dot or period operator. For example the program given in Fig. 6.13 creates structure variable *p1 & p2,* where p1.name, p1.id and p1.height are used to store and display name, id and height respectively of a

Space for learners notes

person. Similarly strcpy(p2.name,"John") copies the name "John" to the variable p2.name and *p2.id*, *p2.height* is assigned the value 123,6.7 respectively.

```
#include<stdio.h>
struct person{
char name[15];
int id;
float height;
};

void main()
{
struct person p1;
struct person p2;

printf("\n Enter person's name, id & height \n");
scanf("%s %d %f",p1.name,&p1.id,&p1.height);
printf(" Name: %s \n id: %d \n height: %f",p1.name,p1.id,p1.height);

strcpy(p2.name, "John");
p2.id=123;
p2.height=6.7;
printf(" Name: %s \n id: %d \n height: %f",p2.name,p2.id,p2.height);
}
```

Fig. 6.13: Example to access members of a structure using period operator

There are two more ways of assigning and accessing members of a structure when using pointers. If *ptr* is a pointer to a structure variable then to access the member *id* of the structure variable following syntax need to be followed:

- (*ptr).id
- ptr->id

The Fig. 6.14 shows an example to access members of a structure using pointers.

```
#include<stdio.h>
struct person{
char name[15];
int id;
float height;
};

void main()
{
struct person p1;
struct person *ptr;
ptr=&p1;                    //initializing structure pointer

strcpy((*ptr).name,"John");              //access member using pointer
(*ptr).id=123;                           //access member using pointer
(*ptr).height=6.7;                       //access member using pointer
printf(" Name: %s \n id: %d \n height: %f",(*ptr).name,(*ptr).id,(*ptr).height);

strcpy(ptr->name,"Rama");                //access member using pointer
ptr->id=444;                             //access member using pointer
ptr->height=5.2;                         //access member using pointer
printf(" Name: %s \n id: %d \n height: %f",ptr->name,ptr->id,ptr->height);
}
```

Fig. 6.14: Example to access members of a structure using pointers

6.6 STRUCTURES AS FUNCTION ARGUMENTS

There are two ways in which structure can be used as function arguments. In the first method entire copy of the structure can be passed to the function. As the copy of the structure is passed to the function so any changes inside the called function is not visible inside the main function. Therefore it is necessary to return the entire structure to the calling function.

In the example given in Fig. 6.15 the function *update* accepts structure as argument and also returns a structure. The structure *p1* is initialized inside main and the copy of *p1* is passed to the function *update*. Inside the function *update* the values of structure *p1* is copied to structure *p*. The structure variable *p* is updated with new values. Finally the structure *p* is returned to the calling section where structure *p* is copied to *p2* inside main function. The output of the program in Fig. 6.15 is shown in Fig. 6.16.

```
#include<stdio.h>

struct person{
char name[15];
```

Space for learners notes

```
int id;
float height;
};
struct person update(struct person p)    // function with argument & return
                                          type as structure
{
strcpy(p.name,"John");
p.id=123;
p.height=6.7;
return p;
}
void main()
{
struct person p1={"Ram",357,5.5};
struct person p2;
printf("\n\n OLD VALUES \n");
printf("Name: %s \nid: %d \nheight: %f",p1.name,p1.id,p1.height);

p2=update(p1);          //Passing copy of the structure & storing the returned
                          result in p2
printf("\n\n NEW VALUES \n");
printf("Name: %s \nid: %d \nheight: %f",p2.name,p2.id,p2.height);
}
```

Fig. 6.15: Example structure used as function argument

OUTPUT

```
OLD VALUES
Name: Ram
id: 357
height: 5.5

NEW VALUES
Name: John
id: 123
height: 6.7
```

Fig. 6.16: Output of program in Fig. 6.15

The second way uses pointers to pass the structure. The Fig. 6.17 shows use of pointers to pass a structure. The address of the structure *p1* is passed to the called function *update*. Pointers in the called function are used to update the structure variable *p1*. The *update* function assigns values {"John", 123, 6.7} to the members {name, id, height} of the structure variable *p1*. The output of the program in Fig. 6.17 is shown in Fig. 6.18.

Space for learners notes

```
#include<stdio.h>
struct person{
char name[15];
int id;
float height;
};
void update(struct person *p)   // structure pointer accepts address of the
                                structure variable
{
strcpy((*p).name,"John");       //assigning values to members of structure
                                variable using
(*p).id=123;                    // pointers
(*p).height=6.7;
}
void main()
{
struct person p1;          //declaring structure variable p1

update(&p1);               // passing address of structure variable p1 to update
                           function

printf("Name: %s \nid: %d \nheight: %f",p1.name,p1.id,p1.height);
}
```

Fig. 6.17: Use of pointers to pass a structure

OUTPUT

```
Name: John
id: 123
height: 6.7
```

Fig. 6.18: Output of program in Fig. 6.17

Space for learners notes

6.7 STRUCTURES AND ARRAYS

Suppose we want to store details like name, id, and phone of all employees of an organization. In such a case we need to use array of structures to store details of each employee. The example given in Fig. 6.19 shows the way of using array of structures. Here 100 number of structure variables *struct employee emp[100]* are created using an array of structure. Each structure variable is recognized by the index number. emp[0] represents employee number one, emp[1] represents employee number two and so on. Each of the structure variables are initialized with values taken from the keyboard inside the *for* loop.

```
#include<stdio.h>

struct employee{
char name[15];
int id;
char phone[10];
};

void main()
{
struct employee emp[100];            //Array of structures
int i;

for(i=0;i<20;i++)
{
printf("Enter Details {name,id,phone} of Employee no. %d\n",i);
scanf("%s %d %s",emp[i].name,&emp[i].id,emp[i].phone);
printf("\n\n");
}

for(i=0;i<20;i++)
{
printf("Details of Employee no. %d is: ",i);
printf("%s %d %s", emp[i].name, emp[i].id, emp[i].phone);
printf("\n\n");
}
}
```

Fig. 6.19: Example array of structures

The array of structures is stored in memory like multidimensional array. The example below depicts how the employee array is stored inside memory.

Emp[0].name	Johny
emp[0].id	123
emp[0].phone	9876451212
emp[1].name	Rama
emp[1].id	345
emp[1].phone	9765432222
emp[2].name	Robin
emp[2].id	9887776543
emp[2].phone	231

6.8 UNIONS

Unions are similar to structures that groups items of different data types into a single logical unit. Therefore union follows the same syntax as structures. Fig. 6.20 shows the example of a union that uses the keyword union. The major difference between union and structure is in terms of memory allocation. While each member of a structure gets its separate memory location, whereas all members in a union share the same memory location of the biggest member. That means there can be many members of different data types in a union but it can handle only one member at a time.

```
union student
{
int roll;
char gender;
float height;
};
```

Fig. 6.20: Definition of a union

For the example given in Fig. 6.20 the compiler allocates the largest memory of 4bytes required by the member *height* among the three members (roll, gender, height) of the union. The example in Fig. 6.21 shows the memory sharing of each member of the union.

	100	101	102	103
float	■	■	■	■

Fig. 6.21: Sharing of memory by union members

Space for learners notes

Space for learners notes

6.9 INITIALIZINGAN UNION

Once a union variable is declared, it can be used to initialize one member at a time as shown in Fig. 6.22.

```
#include<stdio.h>
union student
{
int roll;
char gender;
float height;
}s2;                              //union variable s2 declaration

void main()
{
union student s1;                 // union variable s1 declaration
s1.roll=25;                       //union initialization
printf("%d",s1.roll);
}
```

Fig. 6.22: Initializing union members

However we can see in Fig. 6.23 that when a different member *s1.gender* is assigned a new value *s1.gender='M'*, the new value supersedes the previous member *s1.roll* value. Therefore *s1.roll* prints a garbage value while s1.gender prints the value 'M'.

```
#include<stdio.h>

union student
{
int roll;
char gender;
float height;
}s2;                              //union variable s2 declaration

void main()
{
union student s1;                 // union variable s1 declaration
```

```
        s1.roll=25;                         //union initialization
        s1.gender='M';                               //union initialization
        printf("%d",s1.roll);               //will print garbage value for roll
        printf("%c",s1.gender); //will print 'M'
        }
```

Fig. 6.23: Initializing union members

Unlike structure where all members can be initialized as in Fig. 6.24, union variable can be initialized only for the first member and rest of the members can be initialized by assigning values or by taking input from keyboard as in Fig. 6.25.

```
        struct student1
        {
        int roll;
        char gender;
        float height;
        };

        void main()
        {
        struct student s1={13,'F',5.5}; // structure variable s1 declaration &
        initialization
        printf("Roll=%d \n Gender=%c \n Height=%f",s1.roll,s1.gender,s1.height);

        }
```

Fig. 6.24: Initializing structure members

```
        union student2
        {
        int roll;
        char gender;
        float height;
        };

        void main()
        {
        //union student2 s3={13,'F',5.5};        ERROR INVALID initialization


        union student2 s4={6.2};                //INVALID initialization
```

Space for learners notes

Space for learners notes

```
union student2 s5={15};              //VALID initialization for the first
                                     member roll
union student2 s6;
printf("Roll=%d", s5.roll);          //prints 15

printf("\nEnter Gender\n");
scanf("%c",&s6.gender);              //initializing union s6, input from
                                     keyboard
printf("\n Gender=%c ",s6.gender);
}
```

Fig. 6.25: Initializing union members

6.10 ACCESSING THE MEMBERS OF AN UNION

To access members of a union same syntax can be followed as with structure i.e. use of dot or period operator along with the union variable as shown in Fig. 6.26.

```
union student2
{
 int roll;
char gender;
float height;
};
void main()
{
union student2 s2;                  // union variable s2 declaration
s2.roll=34;                         //accessing member roll & initializing
printf("\nRoll=%d ",s2.roll);       //Prints Roll=34

s2.gender='M';                             //accessing member gender &
                                           initializing
printf("\nGender=%c ",s2.gender);          //Prints Gender=M

s2.height=5.3;                      //accessing member height & initializing
                                    printf("\nHeight=%f",s2.height); //Prints
                                    Height=5.3
}
```

Fig. 6.26: Accessing the Members of an Union

6.11 CHECK YOUR PROGRESS

Space for learners notes

i. User-defined data type can be derived by__________.

a) typedef
b) enum
c) struct
d) All of the mentioned

ii. ______________cannot be a structure member.

a) Array
b) Another structure
c) Function
d) None of the mentioned

iii. Size of a union is determined by size of the____________________ in the union.

a) First member
b) largest member
c) Last member
d) Sum of the sizes of all members

iv. The size of the union declaration?

```
union sample
{
char a;
int b[10];
double c;
}s;
```

(Assume size of double = 8bytes, size of int = 4bytes, size of char = 1byte)

a) 13
b) 49
c) 40
d) 80

v. If A and B are structure and union respectively with same members then which of the following is statement is incorrect?

a) sizeof(A) is greater than sizeof(B)
b) sizeof(A) is less than to sizeof(B)
c) sizeof(A) is equal to sizeof(B)
d) None of the above

Space for learners notes

vi. What will be the output of following program? (Size of int is 4 bytes)

```
#include<stdio.h>
struct student
{
  int a;
  static int b;
};

void main()
{
 printf("%d", sizeof(struct student));
}
```

   a) 4
   b) 8
   c) Runtime Error
   d) Compiler Error

vii. The below C declaration define 'stud' to be

```
struct student
{
        int id;
         float height;
};
struct student *stud[5];
```

   a) A structure of 3 fields: an integer, a float, and an array of 5 elements
   b) A structure of 2 fields, each field being a pointer to an array of 5 elements
   c) An array, each element of which is a structure of type student
   d) An array, each element of which is a pointer to a structure of type student

viii. Consider the following C declaration

```
struct sample{
  short arr[5];
  union {
     float b;
     long c;
   }u;
 }s;
```

Assume that short, float and long occupy 2 bytes, 4 bytes and 8 bytes, respectively. The memory requirement for variable s, is

a) 22
b) 18
c) 14
d) 10

ix. The operators that can be applied on structure variables is ____________.

a) Assignment (=)
b) Equality comparison (==)
c) None of the above
d) Both of the above

x.
```
union sample
{
 int a,b;
 char arr[8];
};
void main()
{
 printf("%d", sizeof(union test));
}
```
Find the output of above program. Assume that integer is 4 bytes and character is 1 byte.

a) 8
b) 12
c) 16
d) None of the above

xi.
```
union sample{
int a,b;
char arr[4];
 };
void main()
{
  union sample s;
  s.a = 0;
  s.arr[1] = 'P';
  printf("%s", s.arr);
}
```

Space for learners notes

Space for learners notes

Find the output of above program. Assume integer as 4 bytes and character as 1 byte.

a) P
b) Nothing is printed
c) Garbage character followed by 'P'
d) Garbage character followed by 'P', followed by more garbage characters

xii. Find the output of following C program

```
#include<stdio.h>
struct position
{
 int a, b, c;
};
void main()
{
 struct position pos1 = {.b = 7, .c = 1, .a = 2};
 printf("%d %d %d", pos1.a, pos1.b, pos1.c);
}
```

a) 7 1 2
b) 2 7 1
c) Error
d) 2 1 7

xiii. If the structure variable is partially initialized then by default

a) Compilation Error
b) integer members will be 0
c) character will be '\0'
d) Both b and c

xiv. If x->y is syntactically correct then

a) x is a pointer to structure, y is a structure
b) x is a structure, y is a structure
c) y is a pointer to structure, x is a structure
d) x is a pointer to structure, y is a structure member

xv. Which of the following statement is true about usage of structure?

a) Individual members can be assigned Storage class
b) Individual members can be initialized within a structure type declaration
c) The scope of a member name is confined to the structure within which it is defined
d) None of the above

6.12 ANSWERS TO CHECK YOUR PROGRESS

i, d	ii, c	iii, b	iv, c
v, b			
vi, d	vii,d	viii,b	ix,a
x,a			
xi,b	xii,b	xiii,d	xiv, d
xv, c			

6.13 SUMMING UP

A structure is a concept that supports grouping of items of different data types into a single logical unit. A structure must be defined first using the keyword *struct* before any structure variables can be declared and initialized. Once a structure is defined variables can be declared in the definition of the structure by placing the structure variable between the closing brace and semicolon. During compilation memory is allocated to members of each structure variable. A structure does not allow initialization of the individual members in a structure definition. When initializing the structure variables the order of members in structure definition should match. It is also permitted to partially initialize the structure variable that is first few members can be initialized and remaining members can be left blank towards the end. The members which are not initialized are set to default values 0 and '\0' respectively for members of type integer/float and character. To access or assign values to any member of a structure variable dot or period operator is used with structure variable. Structures members can also be accessed using pointers like (*ptr).id and ptr->id. Structure can be used as function arguments either by passing the entire copy of the structure to the function or by using pointers.

Unions are similar to structures that groups items of different data types into a single logical unit. Union follows the same syntax as structures. The major difference between union and structure is in terms of memory allocation. While each member of a structure gets its separate memory location, whereas all members of a union share the same memory location. Also a union can handle only one member at a time. Unlike structures where all members can be initialized at the same time union can be initialized only for the first member. To access individual members of a union, dot or period operator is used with the union variable.

6.14 LET US SUM UP

- Balagurusamy, E., Computer fundamentals and C Programming, McGraw Hill Publishing Company Limited.
- Gottfried, Byron S., Programming with C, McGraw Hill Publishing Company Limited.

Space for learners notes

- Kanetkar, Y., Let us C, BPB Publication.
- Kernighan, B.W., and Ritchie, Dennis M., The C Programming Language, Prentice Hall Pvt Ltd.

6.15 MODEL QUESTION

Q1 Differentiate between Structure and Array.

Q2 Define a structure called book consisting of the following members title, author,publisher, edition, price. Declare a structure variable and initialize it with some values.

Q3 Discuss the rules that are followed while initializing a structure variable.

Q4 Explain using an example on how members of a structure can be accessed.

Q5 Discuss the different ways of passing a structure as function argument.

Q6 Explain the concept array of structures using an example.

Q7 Differentiate between Structure and Union.

Q8 Explain how the compiler allocates memory to a union variable?

Q9 Explain how the members of a union can be initialized?

Q10 Consider a structure called *test* with three members int, float, char in that order. Find out which of the following statements are incorrect and why?

i. struct test t1, t2, t3

ii. struct t1, t2, t3;

iii. test t1, t2, t3;

iv. struct test t1={ };

v. struct test t1[ ];

vi. struct test t1={1+5,8.9," 3" };

vii. struct test t1=5, 8.9, " 3";

viii. struct test t1={1, 2, 3 };

Q11 Consider the following structure declaration

struct test p,q,r;

Find out which of the following statements are legal?

i. p=q;

ii. p=q+r;

iii. if(p>q).........

iv. printf("%d",p);

v. scanf("%d",&q);

Q12 Consider a union called test with three members int, float, char in that order. Find out which of the following statements are incorrect and why?

i. union test t1, t2, t3;

ii. union t1, t2, t3;

iii. test t1, t2, t3;

iv. union test t1={ };

v. union test t1={5,8.9,"3"};

vi. union test t1=15, 3.9, "3";

Space for learners notes

PROGRAMMING EXERCISE

Q13 Define a structure called distance with the following members:

i. Kilometer

ii. Meter

Write a program to do the following:

i. Declare two structure variables d1 & d2.

ii. Initialize d1 & d2

iii. Add d1 & d2 and display the sum

Q14 Define a structure called *complex* containing two members real and imaginary. Write a C program that would initialize individual members and display complex number in the form: 2 + 4i. Also declare two complex variables and display their sum.

Q15 Define a structure named country with the following members:

- City name
- Population of the city
- Number of Schools
- Number of Colleges
- Number of Universities
- Literacy Rate

Write a C program to do the following:

i. To read the details for 5 cities randomly using an array variable

ii. To sort the list based on population

iii. To display the sorted list.

Q16 Define a structure called vector of the form (1, 2, 3, 4, 5, 6, 7, 8). Write a C program to do the following tasks:

i. To declare and initialize a structure variable vector.

ii. Multiply the vector by a scalar value

iii. Display the vector in the form (1, 2, 3, 4, 5, 6, 7, 8).

iv. Function that accepts two vector as input arguments and return their sum.

Q17 Define a structure called *ipl* with the following members:

- player_name

Space for learners notes

- team_name
- number_of_centuries
- number_of_half_centuries
- number_of_wickets
- batting_average
- bowling_average

Write a C program to do the following:

i. Store information of 15 players

ii. Display team wise players list along with their batting and bowling average

Q18 Define a structure called hostel. It should have members that include the name, address, branch, semester, room_number, day_of_entry, day_of_leaving. Write functions to perform the following operations:

i. To store entries of 10 hostel students.

ii. To print out room_number of all the students of a given semester.

iii. To print out name and room_number of all the students of the Computer Science branch.

UNIT 7 POINTER

Space for learners notes

CONTENTS

7.1 INTRODUCTION

In earlier units, we have learnt to declare different types of variables to store different types of data. But in some situations, we are required to store and access the addresses of declared variables. So in such cases, we can declare pointer variables. Pointers are variable that are used to hold memory addresses of another variable of identical type. In this unit, you will learn about pointers.

7.2 OBJECTIVES

After reading this unit, you are expected to be able to learn:

- What is Pointer?
- About Pointer arithmetic,

Space for learners notes

- Relationship of Pointer and Array,
- Use of Pointer in Function,
- Use of Pointer in Structure and
- About dynamic memory allocation.

7.3 DEFINITION OF POINTER

Pointer is a variable which can store the address of another variable of same type. Syntax of declaring a pointer variable is given as follows:

```
Data type * variable_Name;
For example: int *ptr;
```

Here, ptr is a pointer variable with data type int which means that ptr can store the memory address of any integer variable. Now ptr is a single pointer variable. We can declare a double pointer variable also which can store the memory address of any pointer variable of same data type i.e. pointer to pointer.

For example: int *ptr , **dptr;

Here dptr is a double pointer with data type int and it can store the memory address of any single pointer with data type int as shown below.

```
dptr = &ptr ;
```

Here dptr stores the address of ptr. The '&' operator used in the above statements is 'address of' operator in C programming. The expression &ptr will give the memory address of ptr. Now let us consider the following programming statements:

```
int p, *ptr, **ptr ;
p = 10;
ptr = &p;
dptr = &ptr;
```

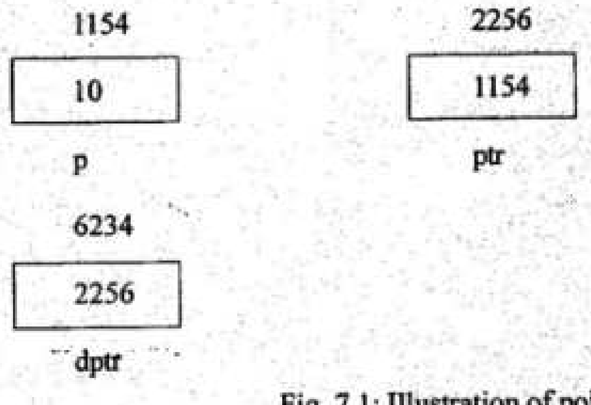


Fig. 7.1: Illustration of pointer

In fig. 7.1 three diagrammatic representations of the three variables p, ptr and dptr are presented where the memory addresses of p, ptr and dptr are assumed to be respectively 1154, 2256 and 6234. Here p is an integer variable which contains an integer value 10. The ptr is a single pointer variable with data type integer which contains the address of the integer variable p. The dptr is a double pointer variable with data type int which contains the address of the memory address of the single pointer variable ptr.

Now what will be the output of the following programming statements?

```
printf("\n The address of p = %u" , &p);
printf("\n Value in p = %u", p);
printf("\n The address of ptr = %u" , &ptr);
printf("\n Value in ptr = %u", ptr);
printf("\n The address of dptr = %u" , &dptr);
printf("\n Value in p = %u", dptr);
printf("\n Value in p = %u",*ptr);
printf("\n Value in ptr = %u", *dptr);
```

Outputs of the above statements are:

```
The address of p = 1154
Value in p = 10
The address of ptr = 2256
Value in ptr = 1154
The address of dptr = 6234
Value in dptr = 2256
Value in p = 10
Value in ptr = 1154
```

Here the ' * ' operator used in the above statements is 'value of' operator in C. So using this operator we can access the value stored in some memory address.

STOP TO CONSIDER

In memory, each byte of memory locations is identified by the CPU with a unique code which is called as the physical address of the particular memory location. A pointer variable is used to store the physical address of the first byte of memory locations allocated for a variable of same type.

Space for learners notes

Space for learners notes

7.4 POINTER TO ARRAY

From previous unit, we have learnt that the elements of an array are stored in contiguous memory locations as shown in fig. 7.2. In case of array, using the name of the array we will get the base address of the array. Now using pointer, we can use this base address to access the array elements.

	0	1	2	3	4	5	6
Arr	43	55	123	76	31	90	89
	6010	6012	6014	6016	6018	6020	6022

Fig. 7.2: Array of numbers with memory locations

7.4.1 Pointer to One Dimensional Array

In fig. 7.2, Arr is a one dimensional integer array of size 7 with base address 6010. Now consider the following programming statements.

```
int *ptr1 , *ptr2 ;
int Arr[7] ;
ptr1 = Arr ;
ptr2 = &Arr[0] ;
printf("\n Base address of Arr = %u", Arr);
printf("\n ptr1 = %u", ptr1);
printf("\n ptr2 = %u", ptr2);
```

Output of the above statements:

```
Base address of Arr = 6010
ptr1= 6010
ptr2= 6010
```

Here ptr1 and ptr2 are two integer pointer both storing address of the same memory location that is the base address of array Arr[ ]. So using '&' operator we can get the address of the element with subscript value 0 in an array which is the base address of the array. So *Ar or *ptr1 or *ptr2 will refer to the element with subscript value 0 in array Ar that is 43.

Now we have the base address of the array and to access all the elements of the array, we can use pointer arithmetic.

There are four arithmetic operators that can be used on pointers that are ++, —, +, and -

Now after the operation ptr1++, ptr1 will point to the location 6012 because each time ptr is incremented, it will point to the next integer location which is 2 bytes next to the current location. So now *ptr1 will refer the integer value 55 which is the element with subscript value 1 in the array Ar. If ptr contains the base address of the array Ar then ptr + i will point the element with subscript value i in the array Ar.

*(Ar + 0) and *Ar and Ar[0] refer the first element in the array Ar. So *(Ar + i) and Ar[i] refers to the (i+1)th element in the array Ar. Actually Ar[i] is internally converted to *(Ar + i) by the C compiler.

Pointers are also can be compared by using relational operators, such as ==, <, and >.

7.4.2 Pointer to Two Dimensional arrays

In case of two dimensional arrays, there are two subscripts values to refer an element. Consider the following programming statement:

```
int A[4][4];
int * ptr;
ptr = A;
```

Here A[ ][ ] is a two dimensional array which can store at most (4 × 4) = 16 integer elements. Let us consider the following diagrammatic representation of the two dimensional array A[ ][ ].

	0	1	2	3
0	20	25	8	12
1	24	32	67	54
2	43	65	71	59
3	89	76	21	41

Fig. 7.3: Two dimensional array

Let assume that the memory address of A[0][0] is 2014 which is the base address of A. In case of a two dimensional array like A, the 0th element of the array A is a one dimensional array. So *A or *(A+0) will give the memory address of first element of the first row of A that is the base address of the 0th one dimensional array in A. In this way, *(A+1) will give the base address of 1st one dimensional array. So *(A+i) will give the base address of ith one dimensional array.

Space for learners notes

Space for learners notes

We know that A[i][j] will refer the element from ith row and jth column. Using *(A+i), we can refer the ith row, so using *(*(A+i)+j) we can refer the particular element from ith row and jth column in A. Now with this concept we can have the following statements.

- A[0][0] and *(*(A)) will refer to the same element.
- A[i][j] and *(*(A+i)+j) will refer to the same element.
- A[i][j] and *(A[i] + j) will refer to the same element.
- A[i][j] and *((*A) + (i * col_no + j)) will refer to the same element, where col_no is the total number of columns in A.

Now from the above programming statements, ptr is a pointer variable and it stores the base address of A. So *ptr will give the element referred by A[0][0]. Now using *(ptr +2*4+3), we can access the element referred by A[2][3] where 4 is the total number of columns in A as shown in fig. 7.3. In this way we can access A[i][j] by using *(ptr + i*col_no + j), where col_no is the total number of columns that is 4 in case of A.

7.4.3 Pointer to Strings

A character pointer can be used to assign the address of a string stored in some memory location. For example: consider the following programming statements:

```
char *st1 = "Welcome to GUIDOL";
char *st2;
char str[ ] = "Welcome to Gauhati University";
st2 = str;
```

Here st1 is a character pointer which is used to assign the address of the string "Welcome to GUIDOL" stored in some memory location. Again str is a character array which is initialized with a string "Welcome to Gauhati University" and a character pointer st2 is used to assign the address of the string stored in str.

STOP TO CONSIDER

We know that a string is stored in a character array. Now address of a string means the physical address of the first character of the string that is stored in the character array with subscript value 0.

7.4.4 Array of Pointers

An array of pointers is an array which stores a collection of same type of pointers. For example; Let us consider the following statements:

```
int *Aptr[6];
int A[ ] = {9 , 5 , 8 , 11 , 90 , 32};
Aptr[0] = &A[0];
Aptr[1] = &A[1];
Aptr[2] = &A[2];
Aptr[3] = &A[3];
Aptr[4] = &A[4];
Aptr[5] = &A[5];
```

Here, Aptr[ ] is an array of integer pointers with size 6 that means it can store the memory addresses of 6 integer data. In the above statements, Aptr[ ] store the addresses of 6 integer data which are stored in the integer array A.

Now, let us consider the following programming statement:

```
char *strarr[ ]={
                    "Gauhati University",
                    "GUIDOL",
                    "ASSAM",
                    "INDIA"
              };
```

Here strarr[ ] is a array of character pointers and it is used to store the base addresses of four strings. So strarr[0] will store the base address of the string "Gauhati University". In this way, strarr[1], strarr[2] and strarr[3] will store the base addresses of the strings "GUIDOL" , "ASSAM" and "INDIA" respectively.

Space for learners notes

CHECK YOUR PROGRESS

1. **Multiple choices**

(a)
```
int a , *b;
a = 10;
b = &a;
printf("%d", a+*b);
```
The output of the above statements is_____.

(i) 10

(ii) 20

(iii) 21

(iv) Error message

Space for learners notes

(b) If Arr is a two dimensional array then Arr[i] gives______.

(i) Data stored in Arr[i][0]

(ii) Address of Arr[i][0]

(iii) Address of Arr[0][i]

(iv) Garbage value

(c) Which of the following statement is equivalent to &Arr[0][0] where Arr is a two dimensional character array?

(i) Arr

(ii) Arr[0]

(iii) Arr[ ][0]

(iv) Both (i) and (ii)

(d) Which of the following statement is equivalent to *(*(Arr+i)+j) where Arr is a two dimensional array?

(i) Arr[i][j]

(ii) *(A[i] + j)

(iii) *((*A) + (i * C + j)), where C is the total number of columns in Arr.

(iv) All of the above

(e) Let us consider the following programming statements.

```
char  *S[ ] ={
                    "Gauhati University",
                    "IDOL",
                    "Computer Science",
                    "M.Sc.IT"
               };
puts(S[2]);
```

The output of the above statements is_____.

(i) Computer Science

(ii) IDOL

(iii) M.Sc.IT

(iv) None of the above

2. Fill-in the blanks

(a) _____ operator is used to access the value stored in a variable pointed by a pointer.

(b) The base address of a one dimensional array Arr[50] can be accessed by ______.

Space for learners notes

(c) If ptr is a pointer variable which points to a character array where the base address of the array is 1133 then ptr++ will point to the memory location_______.

(d) An array of pointers can store_______.

7.5 POINTER TO STRUCTURE

In previous units, structure has already been discussed. Here, we are going to discuss how pointer variable can be used to access a structure variable.

Let us consider the following structure and programming statements:

```
        struct  student
        {
                int roll_no;
                char  name[40];
                float  percentage;
        };
struct  student s1;
struct  student  *stptr;
stptr = &s1;
```

Here s1 is a variable of the structure struct student and stptr is a pointer with the datatype struct student. So stptr can store an address of a structure variable of type struct student using the above programming statement "stptr = &s1" and it is called as pointer to a structure. Now using this pointer variable we can access the structure variable by using -> operator as given below.

```
scanf("%d",&stptr->roll_no);
gets(stptr->name);
scanf("%f", &stptr->percentage);
printf("\n Roll number is = %d", stptr->roll_no);
printf("\n Name of the student is = );

puts(stptr->name);
printf("\n Percentage is = %f", stptr->percentage);
```

7.6 POINTER AND FUNCTION

7.6.1 Passing Memory Address to Function

In unit 5, call by value or pass by value in function has already been discussed where data is directly passed to functions. In this section we are going to discuss how

Space for learners notes

pointers can be used to pass data to functions. It is also referred as call by address or call by pointer or pass by address or pass by pointer.

In case of pass by address, the memory addresses of the actual parameters are passed to functions in the function calling statements and the formal parameters are the pointer variables that can store these addresses.

A C program to swap two integer numbers that are stored in two variables by defining a user defined function is shown below. In this program pass by address is used for argument passing to the user defined function.

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

void swap( int *, int *);
void main()
{
        int num1 , num2;
        clrscr();
        printf("\n Enter the first number =");
        scanf("%d",&num1);
        printf("\n Enter the second number =");
        scanf("%d",&num2);
        printf("\n Before swapping the input numbers are =");
        printf("\n First number = %d",num1);
        printf("\n Second number = %d",num2);
        swap(&num1 , &num2);
        printf("\n After swapping the input numbers are =");
        printf("\n First number = %d",num1);
        printf("\n Second number = %d",num2);
        getch();
}

void swap(int *n1 , int *n2)
{
        int temp;
        temp = *n1;
        *n1 = *n2;
        *n2 = temp;
}
```

In the above program, swap() is the user defined function whose functionality is to swap two numbers that are read in the function main(). Here num1 and num2 are the actual parameters and n1 and n2 are the formal parameters. In main(), swap() is called by passing the memory addresses of num1 and num2 using &("address of") operator. These addresses are stored in the formal parameters n1 and n2 respectively. Finally, in swap(), swapping of the two numbers is performed by using *("value of") operator and a variable "temp".

Space for learners notes

7.6.2 Passing Array to Function through Pointers

In case of array, the base address of the array can be passed to a function. Now using pointer variable and pointer arithmetic we can access the array elements inside the function as shown below with the programming statements.

```
void main()
{
        int Aone[40], Atwo[40][40] ;
        int n,m ;
        clrscr();
        printf("\n Enter the number of elements in the array Aone =");
        scanf("%d",&n);
        input_one(Aone,n);
        printf("\n Display the elements in the array Aone\n");
        display_one(Aone,n);
        printf("\n Enter the number of rows in the array Atwo =");
        scanf("%d",&n);
        printf("\n Enter the number of columns in the array Atwo =");
        scanf("%d",&m);
        input_two(Atwo, n, m);
        printf("\n Display the elements in the array Atwo\n");
        display_two(Atwo,n,m);
        getch();
}
void input_one(int *ptrone, int n)
{
        int i;
    printf("\n Enter %d elements into the array", n);
        for(i = 0 ; i < n ; i++)
        {
```

Space for learners notes

```
                        printf("\nEnter %d th element =",i+1);
            scanf("%d", ptrone + i);
            }
}
void input_two(int *ptrtwo, int n, int  m)
{
            int  i , j ;
            for(i = 0 ; i < n ; i++)
            {
            for(j = 0 ; j < m ; j++)
            {
                        printf("\nEnter (%d,%d) th element =",i , j);
                  scanf("%d", ptrone + i * 40 + j);
                  }
            }
}
void input_display(int *ptrone , int n)
{
            int  i;
            for(i = 0 ; i < n ; i++)
            {
                  printf("\t %d", *(ptrone+i));

            }
}

void display_two(int *ptrtwo, int n, int  m)
{
            int  i , j ;
            for(i = 0 ; i < n ; i++)
            {
            for(j = 0 ; j < m ; j++)
            {
                        printf("\t %d", *(ptrtwo + i * 40 + j));
                  }

            }
}
```

7.6.3 Passing Structure to Function through Pointers

Space for learners notes

In case of structure also, the memory addresses of structure variables can be passed to a function and using pointers to structure variables as formal parameters can be used to store these addresses. Structure variables can be accessed in functions through these pointers.

A C program is shown in the following part where structure is passed to user defined functions using pointer to structure.

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

struct  student
 {
            int roll_no;
            char  name[40];
            float  percentage;
 };

void main()
{
        struct student s1 , s2 , s3 ;
        clrscr();
       printf("\n Enter data for s1 =");
        input(&s1);
         printf("\n Enter data for s2 =");
        input(&s2);
         printf("\n Enter data for s3 =");
        input(&s3);
       printf("\n Display data for s1 =");
        display(&s1);
         printf("\n Display data for s2 =");
        display(&s2);
         printf("\n display data for s3 =");
        display(&s3);
         getch();
 }
 void input( struct student *st)
```

Space for learners notes

```
{
        printf("\n Enter RollNumber =");
        scanf("%d",&st->roll_no);
        printf("\n Enter Name =");
        gets(st->name);
        printf("\n Enter Percentage =");
        scanf("%f",&st->percentage);
}

void display( struct student *st)
{
        printf("\n RollNumber = %d",st->roll_no);
        printf("\n Name = ");
        puts(st->name);
        printf("\n Percentage = %f",st->percentage);
}
```

In the above program, formal parameter, "st" is a pointer to structure, "struct student". It is used to store the address of the structure variable that is declared in main ().

7.7 DYNAMIC MEMORY ALLOCATION

Memory allocation during program execution is called dynamic memoryh allocation. When memory assignment takes place at the runtime of a program, the method is known as dynamic memory allocation. So when we need to allocate memory at runtime and also release it, dynamic memory allocation is performed.

7.7.1 Dynamic Memory Allocation Using malloc() and calloc()

In C programming language, dynamic memory allocation can be performed by using two library functions that are malloc() and calloc(). These functions malloc() and calloc() are called as dynamic memory allocation functions.

The syntax to allocate memory by using malloc() is:

```
malloc (N);        /* N is the number of bytes to be allocated
                   in memory */
```

Here malloc() will allocate memory of N bytes and return the address of the allocated memory. Now if memory allocation is unsuccessful then it will return NULL.

Let ust consider the following programming statements:

```
int *P;
P=(int *) malloc (2);
```

Here P is a integer pointer and it store the address of the memory allocated by malloc() where the size of the allocated memory is 2 bytes as we know the size of a int type variable in C is 2 bytes. The pointer returned by malloc() is typecasted into an integer pointer because P is an integer pointer and malloc() returns a void pointer.

A void pointer is a pointer that points to some memory location which is not associated with any data type. A void pointer can store address of any type of variable and can be typecasted into any data type. The declaration syntax of a void pointer is:

```
void * variable_name ;
```

Now the syntax to call calloc() to allocate memory is:

```
calloc(N,M)  /* N is the number of elements for which memory locations
                are to be allocated and M the memory size of each
                element */
```

Here calloc() will allocate memory for N elements with M bytes for each element that means total N * M bytes of memory will be allocated. Now calloc() also return a void pointer to the beginning of the allocated storage area in memory if memory allocation is successful otherwise it returns NULL.

Let us consider the following programming statements:

```
int *P ;
P=(int *)calloc(50,2) ;
```

Here, P is a integer pointer and it store the starting memory address of the memory block to store 50 integer data allocated by calloc(). In the above statement, the second argument in calloc() is 2 because the size of int variable in C is 2 bytes. Here also typecasting into integer pointer is used as P is a integer pointer and calloc() returns a void pointer.

Differences between malloc() and calloc() are given as follows:

Space for learners notes

Space for learners notes

- malloc() allocates one block of memory locations to store one element. It requires only one argument which indicates the size of the memory block to be allocated. On the other hand, calloc() allocates multiple blocks of memory locations. It requires two arguments where the first argument indicates number of memory blocks to be allocated and the second one indicates the memory size of each blocks.
- The memory allocated by malloc() contains garbage values but in case of calloc(), it contains zeros.

7.7.2 Use of Library Function free()

Now free() is a library function in C which can be used to release a reserved memory space.

The syntax to call free() is:

```
free(P); // P is the address of the memory area to be released
```

Here free() will release the allocated memory space pointed by the pointer P. The return type of free() is void that means free() does not return any value.

STOP TO CONSIDER

It is necessary to include the header file "alloc.h" or "stdlib.h" to use malloc(), calloc() and free() in a C program.

Example 7.1 Write a C program to create a linear linked list to store a list of student names. Write a function to display data available in the linked list. Use malloc() for dynamic memory allocation. The linked list creation function should return the address of the first node in the linked list.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct Lnode            // Structure to create nodes of the linked list
{
        char st_name[200];
        struct Lnode *next;
};
typedef struct Lnode Lnode;
Lnode * List_create(Lnode **);
void List_display(Lnode *);
void main()
{
        Lnode *start = NULL;
        clrscr();
        start = List_create(&start);
        if(start = = NULL)
        {
                printf("\n Creation of linked list is not possible at this
moment");
        }
          List_display(start);
          getch();
}

Lnode * List_create(Lnode **start)      // Function to create a linked list
{

         Lnode *newnode;
         char cont;

         do
         {
                newnode = (Lnode *) malloc (sizeof (Lnode)); /*
 Dynamic memory
                              allocation to create

       a node  */

                if(newnode = = NULL)
                {
                        printf("\n Memory allocation is not possible");
                        return (*start);
                        }
                else
                {
                        printf("\n Enter a name to the new node =");
                        gets(newnode->st_name);
                        newnode->next = NULL;
```

Space for learners notes

Space for learners notes

```
                                if(*start==NULL)
                                        *start=newnode;
                                else
                                {
                                        newnode->next=*start;
                                        *start=newnode;

                                }

                        }
                        printf("\n Input 'y' or 'Y' to add more nodes to the
linked list=");
                        cont=getch();

                }while(cont=='y' || cont=='Y');
                return(*start);
}

void List_display(Lnode *start)  /* Function to display data available
                                                in the linked list */
{
                Lnode *ptr;
                if(start==NULL)
                        printf("\n The linked list is empty");
                else
                {
                        ptr=start;
                        printf("\n Data in the linked list are::\n");
                        while(ptr!=NULL)
                        {
                                puts(ptr->st_name);
                                printf("\n");
                                ptr=ptr->next;
                        }
                }
}
```

STOP TO CONSIDER

Linear linked list is a set of linked nodes where each node consists of two parts that are data part and address part. Data part contains data and address part contains address of the next node. The address part of the last node contains NULL. In this type of data structure, a pointer is also used to contain the address of the first node.

Space for learners notes

CHECK YOUR PROGRESS

3. **Multiple choices**

(a) Which of the following operator is used to access a structure variable through a pointer to structure?

(i) *

(ii) &

(iii) ->

(iv) Both (i) and (ii)

(b) Which of the following is a correct way to pass a string to a function where the string is stored in the character array str[40]?

(i) fun(str)

(ii) fun(&str[0])

(iii) fun(*str)

(iv) Both (i) and (ii)

(c) ______ is used to allocate memory at run time.

(i) malloc()

(ii) alloc()

(iii) free()

(iv) Both (i) and (ii)

(d) malloc() returns a _____.

(i) Integer pointer

(ii) Void pointer

(iii) Character pointer

(iv) None of the above

(e) The memory allocated by calloc() contains____.

(i) Garbage values

(ii) Zeros

(iii) Ones

(iv) None of the above

Space for learners notes

4. State whether true or false

(a) The memory allocated by malloc() contains Zeros.

(b) free() is used to release the allocated memory pointed by a pointer.

(c) "alloc.h" is included in a C program when dynamic memory allocation is required to perform by using malloc().

(d) A two dimensional array cannot be passed to a function by using a pointer.

7.8 SUMMING UP

In this unit, we have learnt about pointers in C programming. Pointer is a variable which can store the address of another variable of same type.

Pointer to pointer is a pointer variable which stores the memory address of any pointer variable of same data type.

Using pointer, we can use the base address of an array to access the array elements. A[i] can be referred by *(A+i) and A[i][j] can be referred by using *(*(A+i)+j). An array of pointers is an array which stores a collection of same type of pointers.

Using pointer variable we can access the structure variable by using -> operator.

In case of array, the base address of the array can be passed to a function and in the function definition, using pointer variable and pointer arithmetic we can access the array elements.

Dynamic memory allocation is the allocation of memory during program execution that is at the runtime of a program. In C programming, dynamic memory allocation can be performed by two library functions that are malloc() and calloc(). free() is a library function in C that can be used to release a reserved memory space. To use malloc() , calloc() and free() in a C program, it is necessary to include header file "alloc.h" or "stdlib.h".

A void pointer is a pointer that points to some memory location which is not associated with any data type and it can store address of any type of variable.

ANSWERS TO CHECK YOUR PROGRESS

1. (a). (ii) , (b). (ii) , (c). (iv) , (d). (iv) , (e). (i)
2. (a). * , (b). Arr , (c). 1134 , (d). Collection of similar types of pointers
3. (a). (iii) , (b). (iv) , (c). (i) , (d). (ii) , (e). (ii)
4. (a). False , (b). True , (c). True , (d). False

7.9 POSSIBLE QUESTIONS

Space for learners notes

1. Define a pointer with suitable example.
2. What is pointer to pointer? Give example.
3. Explain call by address with suitable example.
4. Explain how a two dimensional array can be passed to a function using pointer. Give a suitable example.
5. What do you mean by dynamic memory allocation?
6. Write down the differences between malloc() and calloc().
7. Explain how pointer arithmetic can be used to access one dimensional and two dimensional arrays. Give examples.

7.10 REFERENCES AND SUGGESTED READINGS

- Kanetkar, Yashavant P. *Let us C*. BPB publications, 2016
- Kanetkar, Yashavant P. *Understanding Pointers In C*. Bpb Publications, 2003.
- Byron Gottfried, Jitender Kumar Chhabra, *Programming with C*, Schaum's Outlines Series, Tata McGraw Hill Publications, 2011
- Balagurusamy Elappa, *Programming in ANSI C* , Tata McGraw Hill Publications, 2006
- Venugopal, K. R., Prasad S.R, *Mastering C*. Tata McGraw-Hill Education, 2007

UNIT 8 C PREPROCESSORS AND COMMAND LINE ARGUMENTS AND FILES

Space for learners notes

CONTENTS

8.1 INTRODUCTION

In this unit you will understand the concept of preprocessor directives and its functions. In C language, whenever user writing a program, it is termed as source program or source code. After compilation the source code is converted to object code. There is a special program in C language called preprocessor which is executed before compilation of a source program. The C-preprocessor is a collection of statements called directives. These directives usually appear before the main function of a source program. It begins with the symbol # in column one and do not require a semicolon at the end. In this unit you will learn about the different ways of defining macro substitution directives, File inclusion directives and conditional compilation directives. In this unit you will also learn about the uses of two command line arguments

Space for learners notes

argc and argv. The argc is an integer type argument and contains the number of arguments passed and argv is a pointer to an array of strings. In C-language, file is required for storing data permanently. It is not possible to preserve the output of a program without file. Further, in this unit you will learn about the data files and how the file pointer is used to accessing data from secondary storage device like hard-disk or a floppy drive. A file can be open in various file accessing modes and according to the file access mode, different library functions are used for accessing the data from a data file. The concept of opening and closing a data file, reading and writing a data file and the input output operations in binary files shall also be covered in this unit.

8.2 OBJECTIVES

After going through this unit you will be able to:

- learn about C preprocessor
- learn about different ways of macro substitution
- understand the different form of file inclusion directives
- learn about the different form of conditional compilation directives
- understand the use of command line arguments
- understand the concept of files
- understand the use of the file pointer
- learn about opening and closing a data file
- learn about input output operations with files
- learn about writing and reading a data file
- understand the concept of unformatted data files
- understand the concept of binary files

8.2 C-PREPROCESSOR

The C-preprocessor is a collection of statements called directives. These directives usually appear before the main function of a source program. Preprocessor is a program that processes the source code before it passes through the compiler.

The special syntax rules followed by a C preprocessor are

- Preprocessor directives usually appear at the beginning of a program.
- All preprocessor directives are starting with # symbol in column one.
- Preprocessor directive do not require any semicolon (;) at the end.
- There should be only one preprocessor directive on one line.

The preprocessor directive can categorized as

- Macro Substitution Directives,
- File Inclusion Directives and

- Conditional Compilation Directives.

Space for learners notes

8.4 MACRO SUBSTITUTION DIRECTIVES

In macro substitution an identifier in a program is replaced by a predefined string or values.

Syntax

The syntax of a macro substitution directive in C programming language is "

#define identifier value or string

Examples:

#define No 100

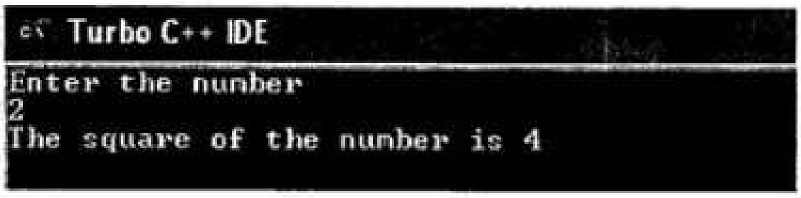
#define Institute "idol"

The constant value 100 is identified by No and "idol" is identified by Institute which is actually considered as string. At the time of preprocessing, the value 100 and "idol" are substituted in the place of No and Institute. The identifiers No and Institute are considered as macros.

Example 8.4.1 Find the square of any given number using macro substitutions.

```
#include<stdio.h>
#include<conio.h>
#define sqr(x) x*x
int main()
        {
        int n;
        clrscr();
        printf("Enter the number\n");
        scanf("%d",&n);
        printf("The square of the number is %d",sqr(n));
        getch();
        return 0;
        }
```

The output of the **Example 8.4.1** is as follows



```
Enter the number
2
The square of the number is 4
```

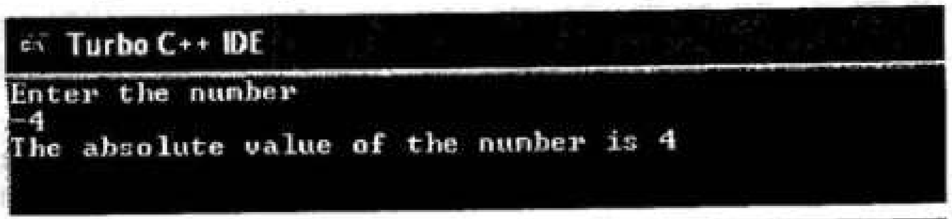
Explanation: Here sqr(2) is expanded as 2*2.

Space for learners notes

Example 8.4.2 Find the absolute value of a given number using macro substitutions

```
#include<stdio.h>
#include<conio.h>
#define abs(x) (((x)>0)?(x):(-(x)))
int main()
        {
        int n;
        printf("Enter the number\n");
        scanf("%d",&n);
        printf("The absolute value of the number is %d",abs(n));
        getch();
        return 0;
        }
```

The output of the **Example 8.4.2** is as follows



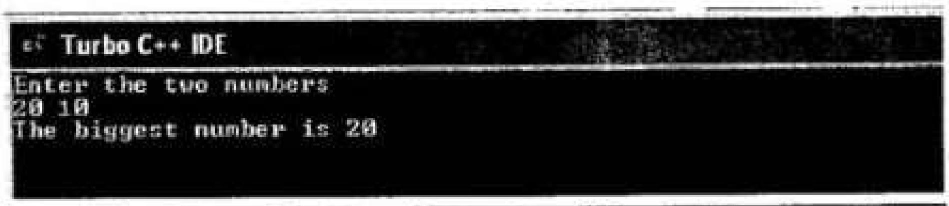
```
Enter the number
-4
The absolute value of the number is 4
```

Explanation: Since -4<0, from the conditional operator abs(-4) is expanded as –(-4) which is equal to 4.

Example 8.4.3: Find the biggest number from any two given numbers using macro substitutions.

```
#include<stdio.h>
#include<conio.h>
#define big(a,b) ((a>b)?a:b)
int main()
        {
        int a,b,result;
        clrscr();
        printf("Enter the two numbers\n");
        scanf("%d%d",&a,&b);
        result=big(a,b);
        printf("The biggest number is %d",result);
        getch();
        return 0;
        }
```

The output of **Example 8.4.3** is as follows



```
Enter the two numbers
20 10
The biggest number is 20
```

Explanation: Here the inputted numbers are 20 and 10, according to the condition from the conditional operator the biggest number is evaluated as 20.

Space for learners notes

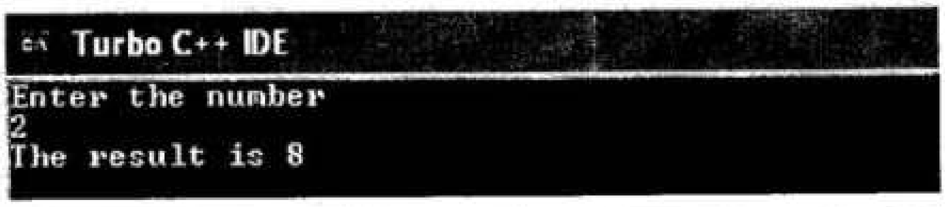
Nesting of Macro:

A macro can be defined within a macro which is called nesting of macro. Let us take the following example for implementing the nesting of macro.

Example 8.4.4 Calculate the cube of a given number using macro.

```
#include<stdio.h>
#include<conio.h>
#define sq(x) x*x
#define cube(x) (sq(x)*(x))
int main()
        {
        int n;
        printf("Enter the number\n");
        scanf("%d",&n);
        printf("The result is %d",cube(n));
        getch();
        return 0;
        }
```

The output of the **Example 8.4.4** is as follows



```
Enter the number
2
The result is 8
```

Explanation: Here cube (2) is first expanded into (Sq(2) * (2)) Since sq(2) is still a macro, it is further expanded into (2 * 2) *(2) which is finally evaluated as 2^3

Undefining a Macro:

A defined macro can be undefined, using the following statement

    #undef identifier

Space for learners notes

Let us take the following example for undefining a macro, which is already defined.

Example 8.4.5

```
#include<stdio.h>
#include<conio.h>
#define TEMP 10
#undef TEMP
#define TEMP 75
int main()
{
 printf("%d",TEMP);
 getch();
 return 0;
}
```

Here the variable TEMP is defined as 10 initially, but after that it is undefined and again defined as 75. So the output of the program is 75.

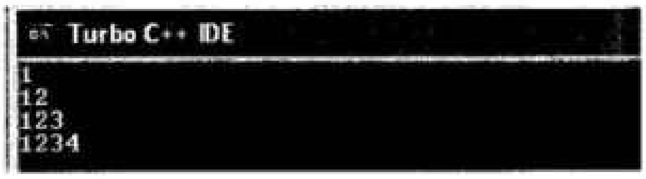
Defining multi line macro:

Multi line macros can be defined by placing a backslash (\) at the end of each line except the last one. This feature permits a single macro (i.e. a single identifier) to represent a compound statement.

Example 8.4.6

```
#include<stdio.h>
#include<conio.h>
#define loop for(i=1;i<=4;i++)            \
                    {                     \
                    for(j=1;j<=i;j++)             \
                      printf("%d",j);             \
                    printf("\n");         \
                    }
int main()
        {
                int i,j;
                clrscr();
                loop
                getch();
                return 0;
        }
```

The output of **Example 8.4.6** is as follows



```
1
12
123
1234
```

8.5 FILE INCLUSION DIRECTIVES

A file inclusion directive is commonly used to include the content of the header files in a program. A previously written program can be included in a new program by using file inclusion directive. External files containing functions or macro definition can also be included as a part of a program using file inclusion directives.

Syntax:

#include "filename"

#include <filename>

When the filename is included in double quotation mark, then computer searches for the file in the current directory and then in the standard directories.

When the filename is given inside angle brackets, then computer will search the file only in the standard directories.

Some commonly used header files with #include are as follows.

#include <stdio.h>

Here the header file stdio.h(short for standard input output) contains various input/ output functions like printf( ),scanf( ) etc..For using input/output functions, you need to include stdio.h at the beginning of a program.

#include<conio.h>

Here the header file conio.h contains all the console input/output functions like clrscr( ),getch( ) etc. For using console input/output functions, you need to include conio.h at the beginning of a program.

#include <math.h>

Here the header file math.h contains all the mathematical functions like acos( ),exp( ),sqrt( ),pow( ) etc. For using mathematical functions, you need to include math.h at the beginning of a program.

8.6 CONDITIONAL COMPILATION DIRECTIVES

The conditional directives are used for conditional compilation of the source program depending on the one or more true or false values.

The most frequently used conditional compilation directives are #if,#elif,#else and #endif.

Space for learners notes

Space for learners notes

Syntax:

```
#if <constant-expression>
 #else
 #endif

#if <constant-expression>
 #elif <constant-expression>
 #endif
```

The compiler only compiles the lines that follow the #if directive when <constant-expression> evaluates to true. Otherwise, the compiler skips the lines that follow until it encounters the matching #else or #endif.

If the expression evaluates to false and there is a matching #else, the lines between the #else and the #endif are compiled.

#if directives can be nested, but matching #else and #endif directives must be in the same file as the #if.

Example 8.6.1

```
#include<stdio.h>
#include<conio.h>
#define NUM 10
int main()
{
#if(NUM == 0)
    printf("\nNumber is Zero");
#elif(NUM > 0)
    printf("\nNumber is Positive");
#else
    printf("\nNumber is Negative");
#endif
getch();
return 0;
}
```

Explanation: In Example 8.6.1, the constant expression of the #if directive evaluates to false and the constant expression of the #elif directive which is evaluates as true. So, the output of the program is "Number is Positive".

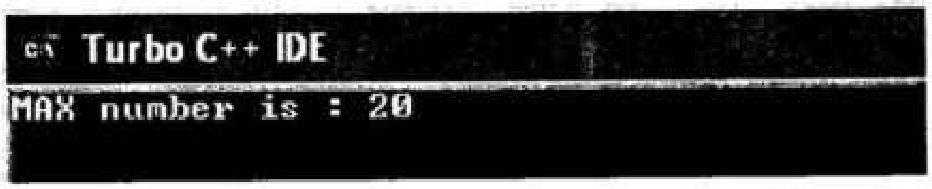
#ifdef:

If the macro name specified after #ifdef is defined previously in #define directive, then the statement_block is followed otherwise it is skipped. We can say that the conditional directive #ifdef is succeeded only for already defined macro.

Example 8.6.2:

```
#include<stdio.h>
#include<conio.h>
#define NUM 10

int main()
{
// Define another macro if MACRO NUM is defined
#ifdef NUM
    #define MAX 20
#endif
printf("MAX number is : %d",MAX);
getch();
return 0;
}
```

The output of **Example 8.6.2** is as follows



```
MAX number is : 20
```

Explanation: Since the macro "NUM" is already defined, so it can be defined another macro as "MAX" whose value is 20.

#ifndef:

If the macro name specified after #ifndef is not defined previously in #define then the statement_block is followed otherwise it is skipped. We can say that the conditional directive #ifndef is succeeded only for undefined macro.

Example 8.6.3

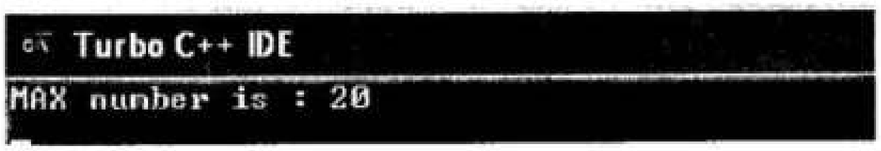
```
#include<stdio.h>
#include<conio.h>
int main()
{
#ifndef NUM
    #define MAX 20
#endif
printf("MAX number is : %d",MAX);
getch();
```

Space for learners notes

Space for learners notes

```
return 0;
}
```

The output of **Example 8.6.3** is as follows



```
MAX nunber is : 20
```

Explanation: Here MAX value will take as 20, since the macro "NUM" is not defined previously.

8.7 COMMAND LINE ARGUMENTS

The command line arguments represented by argc and argv passed to main function through command line. The first argument argc must be an integer variable, while the second one argv is an array of pointers to characters that is an array of strings. Each string in this array will represent an argument that is passed to main function. The value of argc indicates the number of arguments passed.

The following outline indicates how the arguments argc and argv are defined within the main( ) function.

```
main(int argc, char *argv[ ])
{
 .....................
}
```

For execution purpose the order of the arguments and the program name follow the following rule

Program-name argument1 argument 2 ....argument n

Here every single item must be separated from one another either by blank spaces or by tabs. The second argument argv stored the program name as the first item, followed by each of the arguments. If a program has n arguments, there will be (n+1) items in argv ranging from argv[0] to argv[n] and the first argument argc automatically assigned the (n+1) value. Consider the following example, which will execute from the command line.

Example 8.7.1

```
#include<stdio.h>
#include<conio.h>
int main(int argc, char *argv[ ])
        {
        int c;
printf("argc=%d\n",argc);
for(c=0;c<argc;++c)
 printf("argv[%d]=%s\n",c,argv[c]);
getch();
return 0;
}
```

Suppose for example, if the program name is given as hello, and the command line initiating the program execution as **hello welcome to idol**, then the output of the program is as follows.

```
argc  = 4
argv [0] = hello.exe
argv[1] = welcome
argv[2] = to
argv[3] = idol
```

The output indicates that four separate items have been entered from the command line. The first is the program name, hello.exe, followed by the three arguments welcome, to and idol. Each item is an element in the array argv. Here hello.exe is the object file, resulting after the compilation of the hello.c file. Now if the command line initiating the program execution is **hello welcome "to idol"**

The output will be

```
argc= 3
arg[0]=hello.exe
arg[1]=welcome
arg[2]=to idol
```

Here string "to idol" will be interpreted as a single argument.

CHECK YOUR PROGRESS

1. What is a preprocessor directive? What are the different types of preprocessor directives?
2. Describe the rules followed by a C-preprocessor.
3. Describe the rules for defining macros. How can you undefine a macro?
4. What is conditional compilation directive?
5. Give examples of some commonly used conditional compiler directives.
6. What is the use of #ifdef?
7. What is the use of #ifndef?
8. What are argc and argv?

Space for learners notes

8.8 WHY FILES

In computer, the information can be written to and read from secondary memory in many applications. This type of information stored in the secondary memory in the form of data file.

The data file allows you to store information permanently and access to information whenever required. When a programmer need to input huge amount of data for program execution purpose,it is required to type the data again and again for every execution of the program.In such situation using file, data is inputted once and it can be easily used as input file. Moreover transfer of input or output data from one computer to another can easily be-done by using files.

In C language there are two types of data files

(i) Stream oriented (or standard) data file.

(ii) System oriented (or low-level) data file

The stream oriented data file can be divided in two categories . The first category is referred as text files. The text file consists of consecutive characters which can be interpreted as individual data item or as components of strings or numbers.

The second category of stream oriented data file is referred as unformatted data files. In this unformatted data file, data items are arranged into blocks which represents more complex data structure, such as arrays and structures. Different sets of library functions are available for processing the stream oriented file of this type.

System oriented data file are closely related to operating system. This type of file is more efficient for certain kind of applications. Separate procedures and library functions are required for accessing the system oriented data files.

8.9 FILE POINTER

When you working with a data file ,the first step is to establish a buffer area.The buffer is used as temporary storage area, while data being transferred to computer's memory to a data file. The buffer area allows information to be write to and read from data file so quickly.

The buffer area is established in the following manner

```
FILE *fp;
```

Where

FILE → special structure type that establishes the buffer area.

fp → is a pointer variable that indicates the beginning of the buffer area

The structure type FILE is defined within the header file stdio.h.

8.10 OPENING AND CLOSING FILES

Space for learners notes

A file must be opened before it can be created or processed. The library function fopen( ) is used to open a file in different accessing modes.

The general format of fopen( ) function is as follows

fp = fopen(File_name, "File accessing mode")

Here fp is the file pointer variable

File_name represent the name of the file and

File accessing mode is the manner in which the data file will be utilized.

The fopen( ) function returns a pointer to the beginning of the buffer area connected with file. When a file cannot be opened, a NULL value is returned by the fopen( ) function.

A data file must be closed at the end of the program regarding to the file. This can be done with the library function fclose( ).

The general format of fclose( ) function is as follows

fclose(fp); where fp refers to the file pointer.

The accessing mode of file operation must be one of strings from the following tables.

File accessing mode	Meaning
"r"	Open an existing file for reading only.
"w"	Open a new file for writing only. If a file name already exists, it will destroyed and a new file is created in its place.
"a"	Open an existing file for appending (i.e, for adding new data at the end of file). If the file name of the specified file does not exists, it will create a new file.
"r+"	Open an existing file both reading and writing
"w+"	Open a new file name for both reading and writing. If a file name is already exists, it will be destroyed and a new file is created in its place.
"a+"	Open an existing file for reading and appending. If the file name does not exists, it will create new file.

Space for learners notes

8.11 INPUT AND OUTPUT OPERATIONS WITH FILES

In file, for handling characters the following functions are used:

putc( ) Function

putc ( ) is used to write a character to a file. The syntax for putc( ) function is as follows

putc(ch,fptr);

Here ch refers to the character variable and
fptr refers to the file pointer

getc( ) Function

getc ( ) is used to read a character in a file. The syntax for getc( ) function is as follows

ch=getc(fptr);

Here ch refers to the character variable and
fptr refers to the file pointer.

feof ( ) Function

While reading data, feof( ) function locate the end of file. The sysntax of feof( ) is as follows

feof(fptr)

Here fptr refers to the file pointer.

fgetc( ) Function

fgetc( ) is used to read a single character from a file. The getc( ) is a macro while fgetc( ) is a function which is defined on header file stdio.h. The general format of the fgetc( ) is

ch=fgetc(fp) ;

Here, fp is the file pointer of the file, and ch is the variable that receives the character.

fputc( ) Function

fputc( ) is used to write a single character on to a given file. The putc( ) is a macro while fputc( ) is function which is defined on header file stdio.h. The general format of the fputc( ) is

fputc(ch,fp);

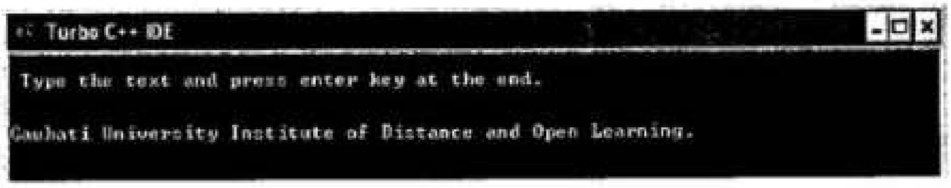
Here, ch is the character to be written and fp is the file pointer to the file to receive the character.

The following program will create a text file using the putc( ) function.

Example 8.11.1

```
#include<stdio.h>
#include<conio.h>
void main()
        {
         char ch;
         FILE *fp;
         fp=fopen("IDOL.txt","w");
         clrscr();
            printf("\n Type the text and press enter key at the end.\n\n\n");
         while((ch=getchar()) !='\n')
           {
           putc(ch,fp);
           }
         fclose(fp);
    }
```

The output of the **Example 8.11.1** is as follows



```
Type the text and press enter key at the end.

Gauhati University Institute of Distance and Open Learning.
```

If you want to read the texts file "IDOL.txt" and count the number of vowels. Then the program can be written as follows

Example 8.11.2

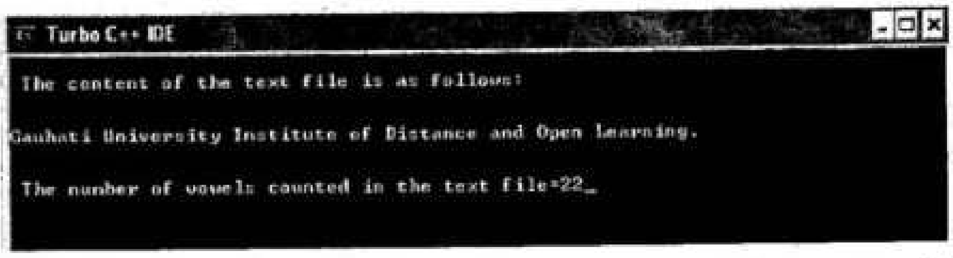
```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
        {
         char ch;
         int count=0;
         FILE *fp;
         clrscr();
         fp=fopen("IDOL.txt","r");
         printf("\n The content of the text file is as follows:\n\n\n");
```

Space for learners notes

Space for learners notes

```
    while(!feof(fp))
     {
      ch=getc(fp);
      printf("%c",ch);
      switch(toupper(ch))
       {
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U': count++;
                  break;
       }
     }
printf("\n\n\n The number of vowels counted in the text  file=%d",count);
     fclose(fp);
     getch();
}
```

The output of the **Example 8.11.2** is as follows



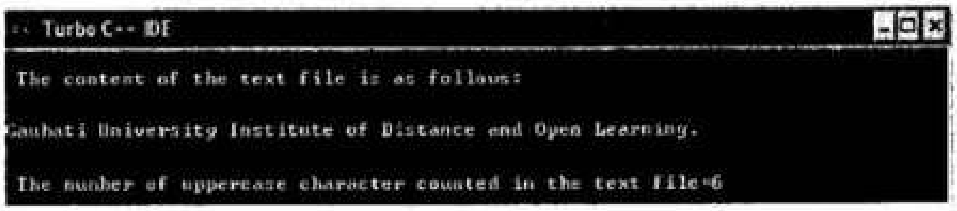
```
The content of the text file is as follows:

Gauhati University Institute of Distance and Open Learning.

The number of vowels counted in the text file=22_
```

In the above output, the content of the file is displayed first then the program will count the number of vowels until it reached the end of file.

Example 8.11.3: Write a C program to read the text file "IDOL.txt" and then count the uppercase characters present in the text file.

```
#include<stdio.h>
#include<conio.h>
void main()
        {
        char ch;
        int count=0;
        FILE *fp;
        clrscr();
        fp=fopen("IDOL.txt","r");
        printf("\n The content of the text file is as follows:\n\n\n");
        while(!feof(fp))
         {
          ch=getc(fp);
          printf("%c",ch);
          if(ch>=65 && ch<=90)
            count=count+1;

         }
         printf("\n\n\n The number of uppercase character counted in the text
file=%d",count);
        fclose(fp);
        getch();
       }
```

The output of **Example 8.11.3** is as follows



```
Turbo C++ IDE
 The content of the text file is as follows:

Gauhati University Institute of Distance and Open Learning.

 The number of uppercase character counted in the text file=6
```

In the above output, the content of the file is displayed first then the program will count the number of uppercase character present in the file using ASCII value until it reached the end of file.

In file, for writing and reading integer value the following functions are used

putw( ) Function

putw( ) function is used to write an integer value from a specified file. The general format of putw( ) function is

putw(num,fp);

Here, num is an integer value to be written and fp is the file pointer to a given file.

getw( ) Function

getw( ) function is used to read an integer value from a givel file. The general format of getw( ) function is

Space for learners notes

Space for learners notes

getw(fp);

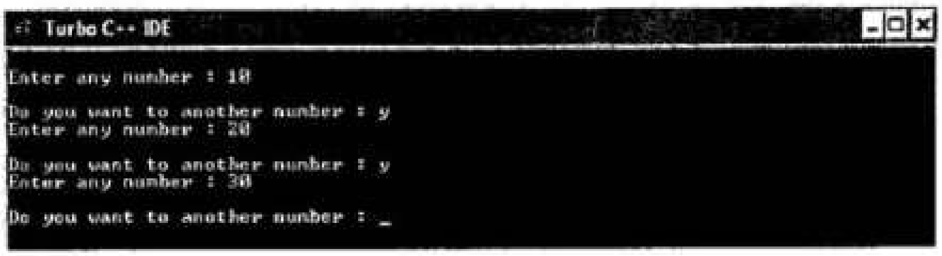
Here fp is a pointer to a file to receive an integer value.

The following programs demonstrates the uses of putw( ) and getw( ) functions.

Example 8.11.4

```
#include<stdio.h>

   void main()
    {
        FILE *fp;
        int num;
        char ch='n';

       fp = fopen("IDOL.txt","w");

        if(fp == NULL)
         {
          printf("\nCan't open file or file doesn't exist.");
              exit(0);
         }

         do
         {
           printf("\nEnter any number :");
             scanf("%d",&num);

              putw(num,fp);

           printf("\nDo you want to another number :");
              ch = getche();

        }while(ch=='y'||ch=='Y');

      printf("\nData written successfully...");

        fclose(fp);
    }
```

The output of **Example 8.11.4** is as follows



```
Enter any number : 10

Do you want to another number : y
Enter any number : 20

Do you want to another number : y
Enter any number : 30

Do you want to another number : _
```

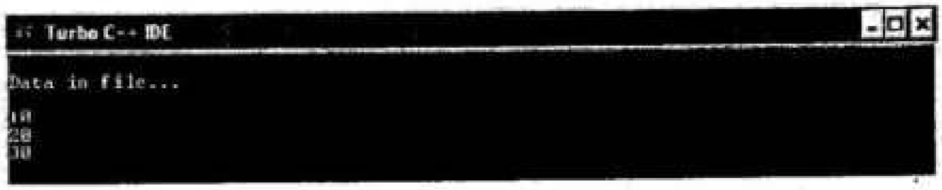
Example 8.11.5

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

   void main()
    {
      FILE *fp;
         int num;
         clrscr();

      fp = fopen("IDOL.txt","r");

       if(fp == NULL)
        {
         printf("\nCan't open file or file doesn't exist.");
            exit(0);
        }

      printf("\nData in file...\n");

      while((num = getw(fp))!=EOF)
        printf("\n%d",num);

         fclose(fp);
         getch();
    }
```

Space for learners notes

Space for learners notes

The output of the **Example 8.11.5** is as follows :



```
Data in file...

10
20
30
```

8.12 WRITING AND READING A DATA FILE

fprintf( ) and fscanf( ) are two commonly used library functions for accessing data in a data file. The details of these two library functions are as given below:

fprintf( ) Function

fprintf( ) is used to write data to a file. It has the following form

fprintf(fptr, "format string", a1,a2,.....an);

Where

fptr refers to file pointer

a1, a2,...................an → refers to variables whose values are written to file

"format string" → refers to the control string which represents the conversion specification

Note that the separator space is used in the format string to write values of the variables as they are stored as strings in the file.

fscanf( ) Function

fscanf( ) is used to read data from a file. It has the following form

fscanf(fptr, "format string", &a1,&a2,.....&an);

Where fptr refers to file pointer

a1,a2,...................an refers to variables whose values are read form file

"format string" refers to the control string which represents the conversion specification.

Note that the separator space must be given in the format string to read values of variables. Also the order of the format string must be same as the format string of fprintf() function used for writing data to the file.

Space for learners notes

rewind( ) Function

rewind( ) function is used to move the file pointer to the beginning of a file. It has the following form.

rewind(fp);

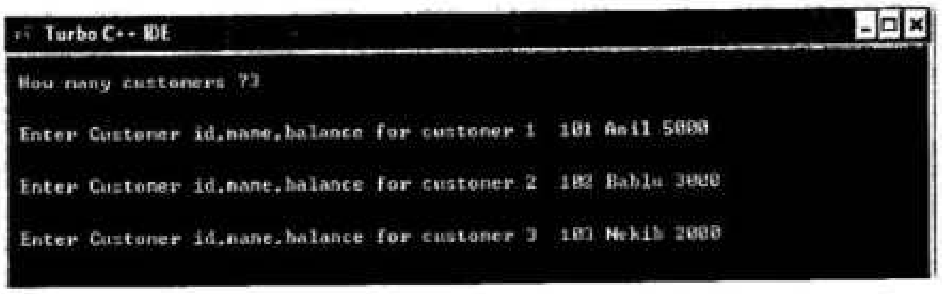
Here fp refers to the file pointer.

Example 8.12.1 A file called customer.dat contains the information of customers such as customerID, name and balance. Write a C-program to create a file to store the details of n customer.

```
#include<stdio.h>
#include<conio.h>
void main()
        {
        int cid,balance,n,i;
        char name[20];
        FILE *fp;
        fp=fopen("customer.dat","w");
        clrscr();
        printf("\n How many customers ?");
        scanf("%d",&n);
        for(i=0;i<n;i++)
         {
               printf("\n\n Enter Customer id,name,balance for customer  %d\t",i+1);
         scanf("%d %s %d",&cid,name,&balance);
         fprintf(fp,"%d %s %d\n",cid,name,balance);
         }
         fclose(fp);
        }
```

Space for learners notes

The output of the **Example 8.12.1** is as follows



Turbo C++ IDE

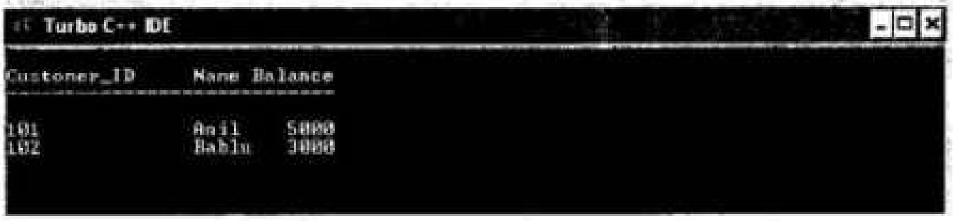
```
How many customers ?3

Enter Customer id,name,balance for customer 1  101 Anil 5000

Enter Customer id,name,balance for customer 2  102 Bablu 3000

Enter Customer id,name,balance for customer 3  103 Nekib 2000
```

Now if you want to read the data file "customer.dat" and display the customer whose balances is greater than 2000, the program can be written in the following manner

Example 8.12.2

```
#include<stdio.h>
#include<conio.h>
void main()
        {
        int cid,balance;
        char name[20];
        FILE *fp;
        fp=fopen("customer.dat","r");
        clrscr();
        printf("\nCustomer_ID           Name Balance");
        printf("\n———————————————————————\n");
        while(!feof(fp))
         {
              fscanf(fp,"%d %s %d\n",&cid,name,&balance);
         if(balance>2000)
          {
                          printf("\n%d\t\t%s\t%d",cid,name,balance);
          }
         }
         fclose(fp);
         getch();
        }
```

The output of **Example 8.12.2** is as follows



```
Customer_ID      Name  Balance
--------------------------------

101              Anil    5000
102              Bablu   3000
```

If you want to add new record to the file "customer.dat", then the program can be written as follows

Example 8.12.3

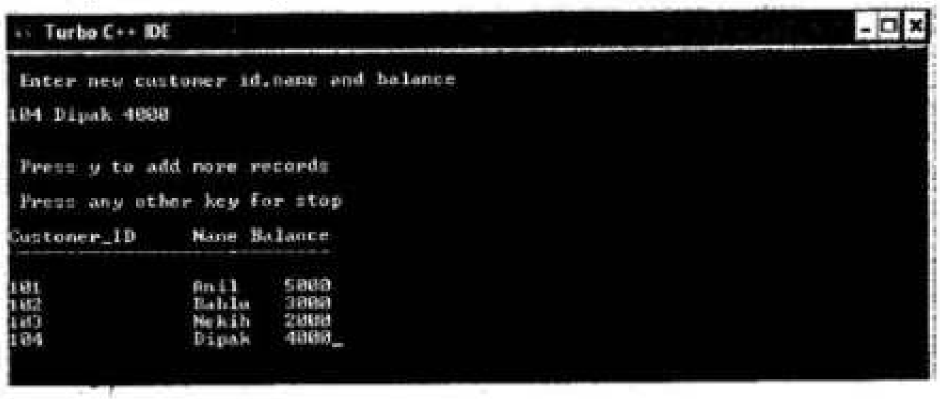
```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
        {
        int cid,balance;
        char name[20],ch='Y';
        FILE *fp;
        fp=fopen("customer.dat","a+");
        clrscr();

        while(toupper(ch)=='Y')
        {
        printf("\n Enter new customer id,name and balance\n\n");
        scanf("%d%s%d",&cid,name,&balance);
        fprintf(fp,"%d %s %d\n",cid,name,balance);
        printf("\n\n Press y to add more records");
        printf("\n\n Press any other key for stop");
        ch=getche();
        }
        rewind(fp);
        printf("\n\nCustomer_ID        Name Balance");
        printf("\n---------------------------------\n");
        while(!feof(fp))
        {
        fscanf(fp,"%d %s %d\n",&cid,name,&balance);
        printf("\n%d\t\t%s\t%d",cid,name,balance);
```

Space for learners notes

Space for learners notes

```
    }

    getch();
    fclose(fp);
   }
```

The output of the **Example 8.12.3** is as follows



Turbo C++ IDE

```
 Enter new customer id.name and balance
104 Dipak 4000

 Press y to add more records
 Press any other key for stop
Customer_ID      Name Balance
--------------------------------

101              Anil    5000
102              Bablu   3000
103              Nekib   2000
104              Dipak   4000_
```

8.13 Unformatted Data Files

In some applications information are stored in the form of blocks of data. These blocks of data generally represent the complex data structure such as an array or a structure. For such applications instead of the writing to or read from individual components of the blocks (such as members of an array or members of a structure), it is desirable to read the entire block from the data file or write entire block to data file. In this situation, two library functions fwrite( ) and fread( ) are used to write to or read from the data file. These two library functions often referred as the unformatted read write functions and the data file of this type referred as unformatted data files. The details of fread() and fwrite() functions are given below

fread( ) Function:

fread( ) function is used to read a structure from a file. The general format of fread( ) function is

```
fread(&st,sizeof(st),1,fptr);
```

Here , the first argument refers to the address of the structure read from the file.

The second argument refers to the size of the structure.

The third argument refers to number of structure read from a file. Usually it is assigned to 1.

The last argument refers to the file pointer.

fwrite( ) Function:

fwrite( ) function is used to write a structure to a file. The general format of fwrite( ) function is

```
fwrite(&st,sizeof(st),1,fptr);
```

Here, the first argument refers to the address of the structure written to the file.

The second argument refers to the size of the structure.

The third argument refers to number of structure written to file. Usually it is assigned to 1.

The last argument refers to the file pointer.

The following two programs demonstrate the uses of fwrite( ) and fread( ) functions.

Example 8.13.1

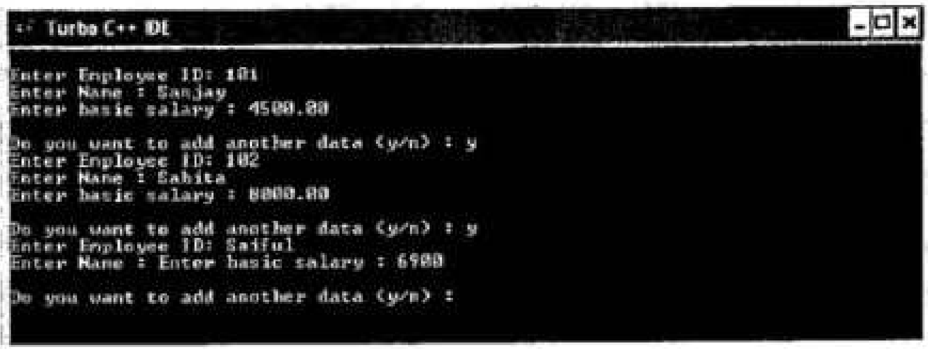
```
#include<stdio.h>
#include<conio.h>

   struct employee
    {
          int eid;
          char name[25];
          float basic;
    };

   void main()
    {
        FILE *fp;
        char ch;
          struct employee e;
        clrscr();

          fp = fopen("Employee.dat","w");

          if(fp == NULL)
        {
         printf("\nCan't open file or file doesn't exist.");
            exit(0);
        }
```

Space for learners notes

Space for learners notes

```
    do
    {
            printf("\nEnter Employee ID: ");
            scanf("%d",&e.eid);

           printf("Enter Name : ");
            scanf("%s",e.name);

            printf("Enter basic salary : ");
            scanf("%f",&e.basic);

            fwrite(&e,sizeof(e),1,fp);

       printf("\nDo you want to add another data (y/n) : ");
         ch = getche();

   }while(ch=='y' || ch=='Y');

  printf("\nData written successfully...");

    fclose(fp);
}
```

The output of Example **8.13.1** is as follows

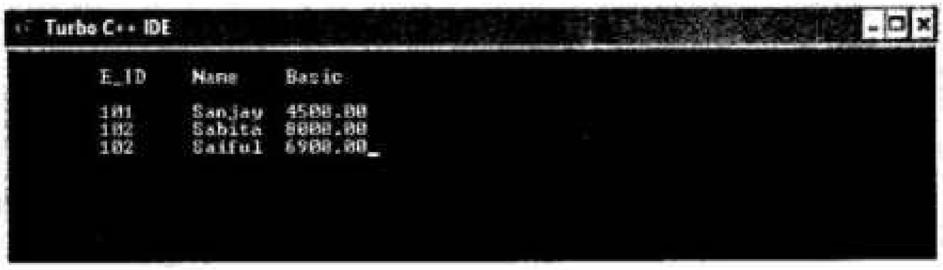


```
Enter Employee ID: 101
Enter Name : Sanjay
Enter basic salary : 4500.00

Do you want to add another data (y/n) : y
Enter Employee ID: 102
Enter Name : Sahita
Enter basic salary : 8000.00

Do you want to add another data (y/n) : y
Enter Employee ID: Saiful
Enter Name : Enter basic salary : 6900

Do you want to add another data (y/n) :
```

Example 8.13.2

```
#include<stdio.h>
#include<conio.h>
   struct Employee
   {
        int eid;
        char name[25];
        float basic;
  };

 void main()
  {
      FILE *fp;
      char ch;
        struct Employee e;
        clrscr();

        fp = fopen("Employee.dat", "r");

      if(fp == NULL)
       {
        printf("\nCan't open file or file doesn't exist.");
           exit(0);
       }

        printf("\n\tE_ID\tName\tBasic \n");

        while(fread(&e,sizeof(e),1,fp)>0)
              printf("\n\t%d\t%s\t%.2f",e.eid,e.name,e.basic);

        fclose(fp);
        getch();
  }
```

The output of **Example 8.13.2** is as follows



```
        E_ID    Name    Basic

        101     Sanjay  4500.00
        102     Sabita  8000.00
        102     Saiful  6900.00_
```

Space for learners notes

Space for learners notes

STOP TO CONSIDER:

A program that reads an unformatted data file can utilize a loop that continuous to read successive records, as long as the value returned by feof is false

8.14 BINARY FILES

Besides of text file, every computer uses another file that is known as binary file. In text file information is stored as text but in binary file information is stored in the form of 0 and 1. This means all the machine language file are actually binary files. For opening binary file the fopen() function is used and the file access modes are written as "wb" and "rb" for writing to and read from the file. Whenever you mention the file access mode as "w" or "r", these two notations actually indicate "wt" or "rt" where t denote the text file. Binary files are handled in different ways as compared to the text file.

The difference between text file and binary file are as in the following table.

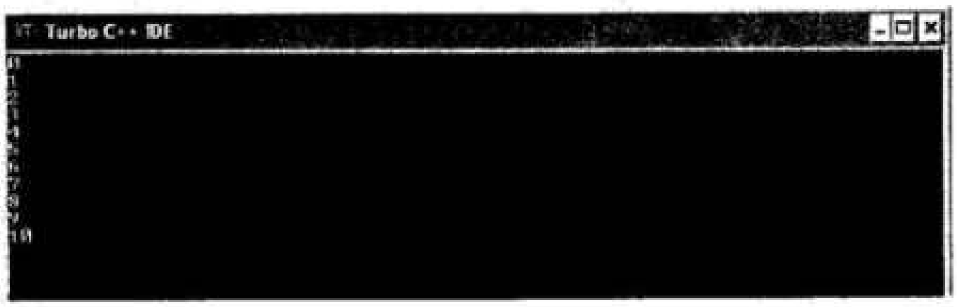
Text file	Binary file
New line character is converted into carriage return-linefeed combination before writing to file. Carriage return-linefeed converted back new line character when the file is read.	No conversion of new line
Special character whose ASCII value is 26 is inserted after the last character in the file to mark end of file.	No special character inserted
Numbers are stored as strings of characters, thus number with more digits would required more disk space. e.g. 4321 will occupy 4 bytes of disk space	Each number occupies same number of bytes on disk as it occupy in memory. e.g. 4321 will occupy 2 bytes only

The following program demonstrate the uses of fprintf( ) and fscanf( ) functions for writing digits to a binary file and then read the digits form that binary file.

Example 8.14.1

```
#include<stdio.h>
#include<conio.h>
void main()
        {
        int num,i;
FILE *fp;
clrscr();
fp=fopen("IDOL.doc","wb");
if(fp==NULL)
  printf("unable to open file\n");
else
 {
  for(i=0;i<=10;i++)
   {
   fprintf(fp," %d",i);
   }
  fclose(fp);
 fp=fopen("IDOL.doc","rb");
 for(i=0;i<=10;i++)
   {
   fscanf(fp,"%d",&num);
   printf("%d\n",num);
   }
 fclose(fp);
 }
getch();
}
```

The output of Example **8.14.1** is as follows



```
0
1
2
3
4
5
6
7
8
9
10
```

CHECK YOUR PROGRESS

9. Explain the data file and its type.
10. How can you establish a buffer area?
11. What are the uses of fopen() and fclose() functions?
12. Explain some character handling library functions.
13. Differentiate between getw() and putw() functions.

Space for learners notes

Space for learners notes

14. What are the uses of fscanf() and fprintf() functions?
15. Differentiate between fread() and fwrite() functions.
16. What is binary file?

8.15 SUMMING UP

In this unit you have learned the concept of preprocessors and its functions in C-programming language. You have learned here different macro substitution techniques and its uses in C-programming. You have also learned the concept of file inclusion directives and conditional compilation directives such as #if,#elif,#else and #endif.

In this unit you have learned the concept of the two command line arguments argc and argv.The argc count the number of arguments passed and argv represents an array of pointers to characters that is an array of strings.

Here in this unit you have learned the concept of file. This is only possible in file, whenever you required to storing the result of a program for future use. There are mainly two types of files one is stream oriented data file which is also referred to as standard data file and another one is the system oriented data file which is often referred as low level data file. Here in this unit you have learned basically about the stream oriented data files.

Before working with files, your first step is to establish the buffer area which is used as the temporary storage area of information. In this unit you have learned the concept of file pointer and its uses for accessing data from a file. There are different types of file opening modes of fopen() function like "w" for writing to file,"r" for reading from file and "a" for appending new data into the file. In this unit you have also learned the concept of various input output functions for handling characters like getc(), fgetc(), putc() and fputc() and handling numbers like getw() and putw(). The getc() and putc() are two macros commonly used to access a text file.The two library functions fprintf() and fscanf() are commonly used to access data in a data file. In this unit you have learned the concept of unformatted data file and fwrite() and fread() functions used to access block of data in unformatted data file. Here you have also learned the concept of binary file and different library functions for accessing the binary information. Here in this unit all the programs have written and executed in Turbo C++ IDE. The turbo C++ IDE offers everything you need to write, edit, compile, link, run, manage and debug your program. https://www.javatpoint.com/how-to-install-c is one of the example from various available links from which you can easily download and install Turbo C++ IDE.

8.16 KEY TERMS

- **C-Preprocessor:** The C-preprocessor is a collection of statements called directives.
- **argc:** Command line argument that indicates the number of parameters passed.

- **argv:** Command line argument that indicates an array of pointers to characters that is an array of strings
- **Stream oriented data file:** The stream oriented data file can be divided in two categories . The first one is referred as text files and second one is referred as unformatted data files
- **System oriented data file** : System oriented data file is closely related to operating system. This type of file is more efficient for certain kind of applications only.
- **Binary Files:** The binary file stored the information in the form of 0 and1.All machine language files are binary file.

Space for learners notes

8.17 ANSWER TO 'CHECK YOUR PROGRESS'

1. Preprocessor means processing is done before the compilation of the program. Macro substitution directive, File inclusion directive and Conditional compilation directive are the preprocessor directives.
2. The preprocessor directives always written in the beginning of the program in column one starting with the # symbol. Preprocessor directive do not require semicolon at the end.
3. Macro is usually written in capital letter for distinguish it from the general variables.

   No comma is allowed between the macro name and the identifiers. Macro can be undefined by using #undef.
4. Conditional compilation is used to define a macro depending on condition.
5. The most frequently used conditional compilation directives are #if,#elif,#else and #endif.
6. If the macro name specified after #ifdef is defined previously in #define directive then the statement_block is followed otherwise it is skipped. We can say that the conditional directive #ifdef is succeeded only for already defined macro.
7. If the macro name specified after #ifndef is not defined previously in #define then the statement_block is followed otherwise it is skipped. We can say that the conditional directive #ifndef is succeeded only for undefined macro.
8. There are two command line arguments argc and argv. argc is an integer type and refers to the number of strings in the command line. argv is an array of pointers to strings.
9. The data file allows you to store information permanently and acess to information whenever required. In C language there are two types of data files

   (i) Stream oriented (or standard) data file.

   (ii) System oriented (or low-level) data file

Space for learners notes

10. When you working with a data file ,the first step is to establish a buffer area. The buffer area is established as

    FILE *fp;

    where FILE is the special structure type that establishes the buffer area and defined on system file stdio.h.

    fp is a pointer variable that indicates the beginning of the buffer area.
11. A file must be opened before it can be created or processed. The library function fopen( ) is used to open a file in different accessing modes. The fopen( ) function returns a pointer to the beginning of the buffer area connected with file. When a file cannot be opened, a NULL value is returned by the fopen( ) function. A data file must be closed at the end of the program regarding to the file. The fclose( ) function is used for this purpose.
12. The library funtions getc( ),fgetc( ),putc( ) and fputc( ) are used for handling character in a text file. The getc( ) and putc( ) are macros while fgetc( ) and fputc( ) are two functions which are defined on header file stdio.h.
13. For writing and reading integer value from a file, two library functions putw( ) and getw() are used. The general format of putw( ) function is

    putw(num,fp);

    Here, num is an integer value to be written and fp is the file pointer to a given file.

    The general format of getw( ) function is

    getw(fp);

    Here fp is a pointer to a file to receive an integer value.
14. Two library functions fscanf( ) and fprintf( ) are commonly used to access a data file. fscanf( ) is used for read data from a data file and fprinf( ) is used for writing data into a data file.
15. fread( ) function is used to read a block of data from a file and fwrite( ) function is used to write a block of data to a file. The block represents the complex data structure such as an array or a structure.
16. In binary, file information is stored in the form of 0 and 1. All machine language files are binary file. The file access modes are written as "wb" and "rb" for writing to or read from a binary file.

8.18 QUESTIONS AND EXERCISES:

Multiple choice questions

1. Preprocessor means
   (a) Processing is done in between the execution of the program
   (b) Processing is done before the compilation of the program

(c) Processing is done after the compilation of the program

(d) Linking of object program.

2. A defined macro can be undefined by using
   - (a) #enddef
   - (b) #ifdef
   - (c) #undef
   - (d) #defined
3. The argument argc represents
   - (a) pointer to an array
   - (b) an integer type argument
   - (c) array of string
   - (d) only a string
4. Stream oriented data file is known as
   - (a) Standard data file
   - (b) Low level data file
   - (c) High level data file
   - (d) None of the above
5. The structure type FILE is defined on
   - (a) Include file
   - (b) ctype.h
   - (c) stdio.h
   - (d) math.h

Answer: 1.(b) 2.(c) 3.(b) 4.(a) 5.(c)

State whether True or False:

1. Preprocessor directive must require a semicolon at the end.
2. Multiline macro cannot be defined in C program.
3. The data file allows you to store information permanently.
4. fgetc() is used to read a single character from a file.
5. It is not possible to write digits into a binary file.

Answer: 1.False 2.False 3.True 4.True 5.False

Fill in the blanks:

1. In a macro substitution an identifier in a program replaced by a predefined ____________.
2. System oriented data file closely related to ________________.
3. The fopen() function returns a _____________ to the beginning of the buffer area connected with file.

Space for learners notes

Space for learners notes

4. _____________ function is used to read integer value from a file..
5. _________________ file store the information in the form of 0 and 1.

Answer: 1.value or string 2.operating system 3.pointer 4.getw ( ) 5.Binary

Match the following:

1.	No conversion of new line	(a)	text files
2.	getc ()	(b)	move the file pointer to the beginning a file
3.	rewind( )	(c)	array of strings
4.	Store sconsecutive characters.	(d)	Binary file
5.	argv	(e)	macro

Answer: 1.(d),2.(e),3.(b),4.(a),5.(c)

Short-Answer Questions

1. What is a macro?
2. How is a multiline macro defined?
3. What are the uses of file inclusion directives?
4. What is the requirement of file in C-language?
5. What is text file?
6. What is system oriented data file?
7. What is unformatted data file?
8. What is the use of rewind () function?
9. Explain the file pointer.
10. What does mode "wb" mean?

Long Answer Questions

1. Write a program with a macro to find the area of a triangle.
2. Write a program with a macro to find the roots of the quadratic equation.
3. Write program with a macro for display the following structure

```
      1
     1 2
    1 2 3
   1 2 3 4
```

4. Explain the two command line arguments with example.
5. Write a program with conditional compilation for finding the area of a rectangle
6. Write a program to write and read a data file.
7. Write a program that reads one character at a time till EOF is reached.
8. Write a program to input numbers in a data file and then read the even numbers from that file.

9. A file called "student.dat" contains the information of students such as Roll, name and total_marks. Write a C-program to create a file to store the details of n students.
10. Write a program to read the details of those students from the file "student.dat" whose total_marks are greater than 500.
11. Write a program to append new students into the file "student.dat" and display the updated content of the data file.
12. Write a program to copy the content of a data file from one to another.

Space for learners notes

8.19 FURTHER READING REFERENCES AND SUGGESTED READINGS

1. Kanetkar, Yashavant P. *Let us C*. BPB publications, 2016
2. Byron Gottfried, Jitender Kumar Chhabra, *Programming with C*, Schaum's Outlines Series, Tata McGraw Hill Publications, 2011
3. Balagurusamy Elappa, *Programming in ANSI C* , Tata McGraw Hill Publications, 2006
4. Jeyapoovan .T, *A first Course in Programmimng with C*, Vikas Publishing House, 2004