

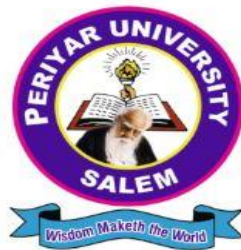
PERIYAR UNIVERSITY

NAAC 'A++' Grade - State University - NIRF Rank 56 – State Public University Rank 25

SALEM - 636 011, Tamil Nadu, India.

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

BACHELOR OF SCIENCE (COMPUTER SCIENCE) SEMESTER - I



PROBLEM SOLVING TECHNIQUES (Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

B.Sc.,(Computer Science) 2024 admission onwards

PROBLEM SOLVING TECHNIQUES

Prepared by:

Centre for Distance and Online Education (CDOE)

Periyar University, Salem – 11.

SYLLABUS

PROBLEM SOLVING TECHNIQUES

Unit I : Introduction: Introduction to problem solving Concept – Steps for problem solving - Problem solving techniques — Algorithms – flowcharts – Types of Algorithms – Use flowcharts to represent algorithms – Pseudo code.

Unit II : Features of C : Introduction-Character Set - Structure of a 'C' Program - Data Types in 'C' - Operations – Expressions - Assignment Statement - Conditional Statements - Structure for Looping Statements - Nested Looping Statements - Multi Branching Statement (Switch), Break and Continue - Differences between Break and Continue - Unconditional Branching (Go to Statement)

Unit III : Functions: Introduction – Functions- Differences between Function and Procedures - Advantages of Functions - Advanced features of Functions – Recursion

Unit – IV : Arrays : Introduction - Definition of Array - Types of Arrays - Two - Dimensional Array - Declare, initialize array of char type - String handling functions in C - File operations

Unit – V : Structures: Introduction - Definition of Structure - Structure Declaration - Structures and Arrays - Structure contains Pointers – Unions - Definition of Union - Differences between Structure and Union

LIST OF CONTENTS

UNIT	CONTENTS	PAGE
1	Introduction: Introduction to problem solving Concept – Steps for problem solving - Problem solving techniques — Algorithms – flowcharts – Types of Algorithms – Use flowcharts to represent algorithms – Pseudo code	8 - 39
2	Features of C : Introduction-Character Set - Structure of a 'C' Program - Data Types in 'C' - Operations – Expressions - Assignment Statement - Conditional Statements - Structure for Looping Statements - Nested Looping Statements - Multi Branching Statement (Switch), Break and Continue - Differences between Break and Continue - Unconditional Branching (Go to Statement)	41 - 84
3	Functions: Introduction – Functions- Differences between Function and Procedures - Advantages of Functions - Advanced features of Functions – Recursion	86 - 103
4	Arrays : Introduction - Definition of Array -	105 -

	Types of Arrays - Two - Dimensional Array - Declare, initialize array of char type - String handling functions in C - File operations	135
5	Structures: Introduction - Definition of Structure - Structure Declaration - Structures and Arrays - Structure contains Pointers – Unions - Definition of Union - Differences between Structure and Union	137 - 166

PROBLEM SOLVING TECHNIQUES

UNIT - I

UNIT 1 - Introduction

Introduction: Introduction to problem solving Concept – Steps for problem solving - Problem solving techniques — Algorithms – flowcharts – Types of Algorithms – Use flowcharts to represent algorithms – Pseudo code

TABLE OF CONTENTS		
UNIT	TOPICS	PAGE
1	1.1. Introduction	8
	1.1.1. Introduction to problem solving Concept	9
	1.2. Steps for problem solving.	10
	1.2.1. Analyzing the problem	11
	1.2.2. Developing an algorithm	11
	1.2.3. Coding	11
	1.2.4. Testing and Debuging	11
	1.3. Problem solving techniques	12
	1.4. Algorithms	14
	1.5. Flowcharts	17
	1.6. Types of Algorithms	21
	1.7. Use flowcharts to represent algorithms	22
	1.8. Pseudo code	25

UNIT OBJECTIVES:

- Introduction to problem solving Concept
- Steps for problem solving
- Problem solving techniques
- Algorithms, flowcharts, Types of Algorithms
- Use flowcharts to represent algorithms
- Pseudo code

1.1 INTRODUCTION

Problem solving is a critical skill in computer science and programming. It refers to the process of finding solutions to problems or challenges by applying logic and critical thinking.

Today, computers are all around us. We use them for doing various tasks in a faster and more accurate manner. For example, using a computer or smartphone, we can book train tickets online.

India is a big country and we have an enormous railway network. Thus, railway reservation is a complex task. Making reservation involves information about many aspects, such as details of trains (train type, types of berth and compartments in each train, their schedule, etc.), simultaneous booking of tickets by multiple users and many other related factors.

It is only due to the use of computers that today, the booking of the train tickets has become easy. Online booking of train tickets has added to our comfort by enabling us to book tickets from anywhere, anytime.

We usually use the term computerization to indicate the use of computer to develop software in order to automate any routine human task efficiently. Computers are used for solving various day-to-day problems and thus problem solving is an essential skill that a computer science student should know. It is pertinent to mention that computers themselves cannot

solve a problem. Precise step-by-step instructions should be given by us to solve the problem. Thus, the success of a computer in solving a problem depends on how correctly and precisely we define the problem, design a solution (algorithm) and implement the solution (program) using a programming language. Thus, problem solving is the process of identifying a problem, developing an algorithm for the identified problem and finally implementing the algorithm to develop a computer program.

Problem solving is a critical skill in computer science and programming. It refers to the process of finding solutions to problems or challenges by applying logic and critical thinking.

1.1.1 Introduction to problem solving concept:

Understanding the problem: This involves carefully reading and comprehending the problem statement and defining the problem in your own words.

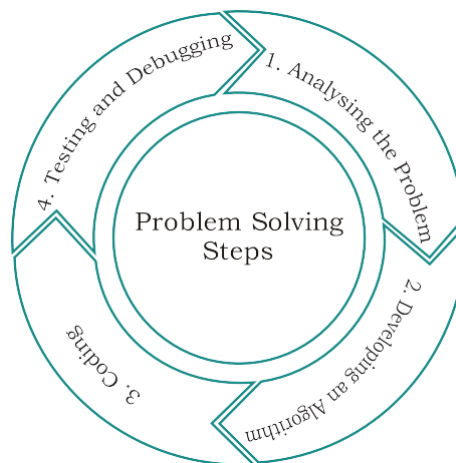
1. **Analyzing the problem:** This involves breaking down the problem into smaller, more manageable parts and identifying the information and data required to solve the problem.
2. **Formulating a plan:** This involves creating a step-by-step plan for solving the problem, including the methods and algorithms that will be used.
3. **Implementing the plan:** This involves coding the solution and testing it to ensure that it works as expected.
4. **Evaluating the solution:** This involves analyzing the solution to ensure that it's correct, efficient, and meets the requirements of the problem.

It's important to note that problem solving is an iterative process and may require multiple iterations of the above steps. The goal is to find a solution that works and meets the requirements of the problem. Effective problem solving skills require patience, persistence, and a willingness to try different approaches until the right solution is found

Problem solving in programming requires critical thinking, creativity, and a deep understanding of the programming concepts and algorithms. It's important to be able to identify patterns and use abstraction and decomposition to break down complex problems into simpler parts. Effective problem solving skills also require patience, persistence, and a willingness to try different approaches until the right solution is found.

1.2 STEPS FOR PROBLEM SOLVING

Suppose while driving, a vehicle starts making a strange noise. We might not know how to solve the problem right away. First, we need to identify from where the noise is coming? In case the problem cannot be solved by us, then we need to take the vehicle to a mechanic. The mechanic will analyse the problem to identify the source of the noise, make a plan about the work to be done and finally repair the vehicle in order to remove the noise. From the above example, it is explicit that, finding the solution to a problem might consist of multiple steps.



When problems are straightforward and easy, we can easily find the solution. But a complex problem requires a methodical approach to find the right solution. In other words, we have to apply problem solving techniques. Problem solving begins with the precise identification of the problem and ends with a complete working solution in terms of a program or software. Key steps required for solving a problem using a computer are shown in the above Figure and are discussed below.

1.2.1 Analyzing the problem

It is important to clearly understand a problem before we begin to find the solution for it. If we are not clear as to what is to be solved, we may end up developing a program which may not solve our purpose. Thus, we need to read and analyze the problem statement carefully in order to list the principal components of the problem and decide the core functionalities that our solution should have. By analyzing a problem, we would be able to figure out what are the inputs that our program should accept and the outputs that it should produce.

1.2.2 Developing an Algorithm

It is essential to devise a solution before writing a program code for a given problem. The solution is represented in natural language and is called an algorithm. We can imagine an algorithm like a very well-written recipe for

1.2.3 Coding

After finalising the algorithm, we need to convert the algorithm into the format which can be understood by the computer to generate the desired solution. Different high level programming languages can be used for writing a program.

It is equally important to record the details of the coding procedures followed and document the solution. This is helpful when revisiting the programs at a later stage.

1.2.4 Testing and Debugging

The program created should be tested on various parameters. The program should meet the requirements of the user. It must respond within the expected time. It should generate correct output for all possible inputs. In the presence of syntactical errors, no output will be obtained. In case the output generated is incorrect, then the program should be checked for logical errors, if any. Software industry follows standardised testing methods like unit or component testing, integration testing, system testing, and acceptance testing while developing complex applications. This is to ensure

that the software meets all the business and technical requirements and works as expected. The errors or defects found in the testing phases are debugged or rectified and the program is again tested. This continues till all the errors are removed from the program.

Once the software application has been developed, tested and delivered to the user, still problems in terms of functioning can come up and need to be resolved from time to time. The maintenance of the solution, thus, involves fixing the problems faced by the user, answering the queries of the user and even serving the request for addition or modification of features

1.3 Problem solving Techniques

There are several techniques that can be used to solve problems in programming. Here are some of the most common problem solving techniques along with examples:

1.Brute Force: This involves trying all possible combinations or solutions to find the correct answer. For example, you can use brute force to solve a problem by trying every possible password combination until the correct one is found.

2. Divide and Conquer: This involves breaking down a complex problem into smaller, more manageable sub-problems and solving each sub-problem individually. For example, you can use divide and conquer to solve a problem by breaking down a large data set into smaller chunks, sorting each chunk individually, and then merging the sorted chunks back together.

3.Greedy Algorithm: This involves making the best choice at each step, with the hope of finding an optimal solution. For example, you can use a greedy algorithm to solve a problem by selecting the highest-value item at each step until you have a complete solution.

4.Backtracking: This involves trying out different solutions and undoing the steps if they lead to an incorrect solution. For example, you can use backtracking to solve a problem by trying out different combinations of numbers and undoing the steps if they don't lead to the correct solution.

5.Dynamic Programming: This involves breaking down a problem into sub-problems and storing the solutions to those sub-problems in a table for later reuse. For example, you can use dynamic programming to solve a problem by breaking down a large data set into smaller chunks and storing the solutions to each chunk in a table for later reuse.

6.Recursion: This involves breaking down a problem into smaller sub-problems and solving each sub-problem recursively. For example, you can use recursion to solve a problem by breaking down a large data set into smaller chunks and solving each chunk recursively until you have the final solution.

It's important to note that different problems may require different problem solving techniques, and that a single problem may have multiple solutions using different techniques. Effective problem solving skills require being able to identify the right technique for the problem at hand and using it to find the optimal solution.






Steps in problem solving

1. **Define the problem:** Clearly identify and understand the problem that needs to be solved.
2. **Gather information:** Collect data and information related to the problem.
3. **Develop potential solutions:** Generate multiple possible solutions to the problem.
4. **Evaluate potential solutions:** Assess each solution based on its potential effectiveness, feasibility, and impact.
5. **Select a solution:** Choose the best solution based on the evaluation.
6. **Implement the solution:** Put the chosen solution into action.
7. **Monitor progress:** Continuously monitor and evaluate the solution to ensure it is solving the problem effectively.
8. **Refine the solution:** Make necessary adjustments to the solution if it is not working as intended.

Algorithms and Flowcharts

The **algorithm and flowchart** are two types of tools to explain the process of a program. In this page, we discuss the differences between an algorithm and a flowchart

and how to create a flowchart to illustrate the algorithm visually. **Algorithms and flowcharts** are two different tools that are helpful for creating new programs, especially in computer

Terminal Box – Start / End	
Input / Output	
Process / Instruction	
Decision	
Connector / Arrow	

programming. An algorithm is a step-by-step analysis of the process, while a flowchart explains the steps of a program in a graphical way.

1.4. ALGORITHM

In our day-to-day life we perform activities by following certain sequence of steps. Examples of activities include getting ready for school, making breakfast, riding a bicycle, wearing a tie, solving a puzzle and so on. To complete each activity, we follow a sequence of steps. Suppose following are the steps required for an activity 'riding a bicycle':

remove the bicycle from the stand,

1. sit on the seat of the bicycle,

2.start peddling,

3.use breaks whenever needed and

4.stop on reaching the destination.

5.Let us now find Greatest Common Divisor (GCD) of two numbers 45 and 56.

Note: GCD is the largest number that divides both the given numbers.

Step 1: Find the numbers (divisors) which can divide the given numbers

Divisors of 45 are: 1, 3, 5, 9, 15, and 45

Divisors of 54 are: 1, 2, 3, 6, 9, 18, 27, and 54

Step 2: Then find the largest common number from these two lists.

Therefore, GCD of 45 and 54 is 9

Hence, it is clear that we need to follow a sequence of steps to accomplish the task. Such a finite sequence of steps required to get the desired output is called an algorithm. It will lead to the desired result in a finite amount of time, if followed correctly. Algorithm has a definite beginning and a definite end, and consists of a finite number of steps.

Definition of Algorithm

Writing a logical step-by-step method to solve the problem is called the **algorithm**. In other words, an algorithm is a procedure for solving problems. In order to solve a mathematical or computer problem, this is the first step in the process.

An algorithm includes calculations, reasoning, and data processing. Algorithms can be presented by natural languages, pseudocode, and flowcharts, etc.

Why do we need Algorithm?

A programmer writes a program to instruct the computer to do certain tasks as desired. The computer then follows the steps written in the program code. Therefore, the programmer first prepares a roadmap of the program to be written, before actually

writing the code. Without a roadmap, the programmer may not be able to clearly visualize the instructions to be written and may end up developing a program which may not work as expected. Such a roadmap is nothing but the algorithm which is the building block of a computer program. For example, searching using a search engine, sending a message, finding a word in a document, booking a taxi through an app, performing online banking, playing computer games, all are based on algorithms.

Writing an algorithm is mostly considered as a first step to programming. Once we have an algorithm to solve a problem, we can write the computer program for giving instructions to the computer in high level language. If the algorithm is correct, computer will run the program correctly, every time. So, the purpose of using an algorithm is to increase the reliability, accuracy and efficiency of obtaining solutions.

A) Characteristics of a good algorithm

- Precision — the steps are precisely stated or defined.
- Uniqueness — results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- Finiteness — the algorithm always stops after a finite number of steps.
- Input — the algorithm receives some input.
- Output — the algorithm produces some output.



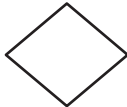
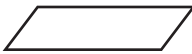

B) While writing an algorithm, it is required to clearly identify the following:

- The input to be taken from the user
- Processing or computation to be performed to get the desired result
- The output desired by the user

Using their algorithmic thinking skills, the software designers or programmers analyse the problem and identify the logical steps that need to be followed to reach a solution. Once the steps are identified, the need is to write down these steps along with the required input and desired output

1.5. Flowcharts

REPRESENTATION OF ALGORITHMS

Flowchart symbol	Function	Description
	Start/ End	Also called “Terminator” symbol. It indicates where the flow starts and ends.
	Process	Also called “Action Symbol,” it represents a process, action, or a single step.
	Decision	A decision or branching point, usually a yes/no or true/ false question is asked, and based on the answer, the path gets split into two branches.
	Input/ Output	Also called data symbol, this parallelogram shape is used to input or output data
	Arrow	Connector to show order of flow between shapes.

Definition of Flowchart

A flowchart is the graphical or pictorial representation of an algorithm with the help of different symbols, shapes, and arrows to demonstrate a process or a program. With algorithms, we can easily understand a program. The main purpose of using a

flowchart is to analyze different methods. Several standard symbols are applied in a flowchart:

There are two common methods of representing an algorithm —flowchart and pseudocode. Either of the methods can be used to represent an algorithm while keeping in mind the following:

1. it showcases the logic of the problem solution, excluding any implementational details
2. it clearly reveals the flow of control during execution of the program

Visual representation of Algorithms

A flowchart is a visual representation of an algorithm. A flowchart is a diagram made up of boxes, diamonds and other shapes, connected by arrows. Each shape represents a step of the solution process and the arrow represents the order or link among the steps.

Write an algorithm to find the square of a number.

Before developing the algorithm, let us first identify the input, process and output:

Input: Number whose square is required

Process: Multiply the number by itself to get its square

Output: Square of the number

Algorithm to find square of a number.

Step 1: Input a number and store it to num

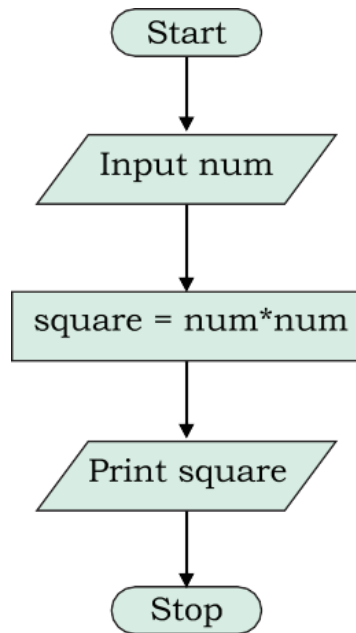
Step 2: Compute $\text{num} * \text{num}$ and store it in square

Step 3: Print square

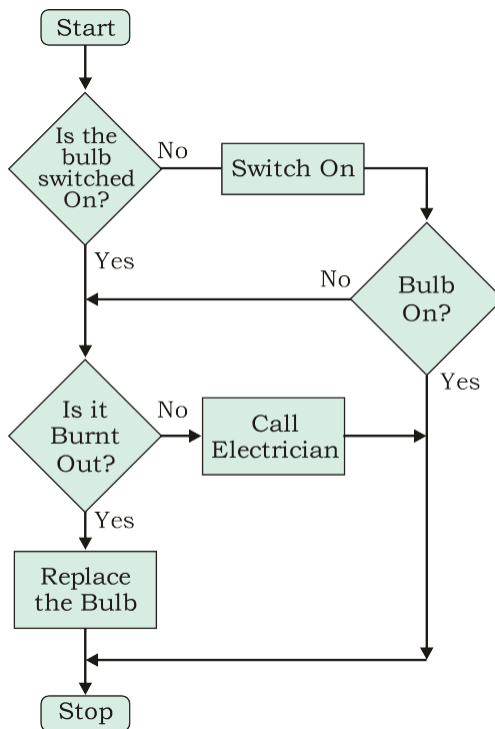
Draw a flowchart to solve the problem of a non-functioning light bulb

The algorithm to find square of a number can be represented pictorially using flowchart as shown in Figure

Flowchart to calculate square of a number



Flowchart to solve the problem of a non-functioning light bulb



The symbols above represent different parts of a flowchart. The process in a flowchart can be expressed through boxes and arrows with different sizes and colors. In a flowchart, we can easily highlight certain elements and the relationships between each part.

Difference between Algorithm and Flowchart

If you compare a flowchart to a movie, then an algorithm is the story of that movie. In other words, **an algorithm is the core of a flowchart**. Actually, in the field of computer programming, there are many differences between algorithm and flowchart regarding various aspects, such as the accuracy, the way they display, and the way people feel about them. Below is a table illustrating the differences between them in detail.

ALGORITHM

- It is a procedure for solving problems.
- The process is shown in step-by-step instruction.
- It is complex and difficult to understand.
- It is convenient to debug errors.
- The solution is showcased in natural language.
- It is somewhat easier to solve complex problem.
- It costs more time to create an algorithm.

Flowchart

- It is a graphic representation of a process.
- The process is shown in block-by-block information **Diagram**.
- It is intuitive and easy to understand.
- It is hard to debug errors.
- The solution is showcased in pictorial format.
- It is hard to solve complex problem.
- It costs less time to create a flowchart.

1.6. Types of algorithm

#1 Recursive Algorithm

It refers to a way to solve problems by repeatedly breaking down the problem into sub-problems of the same kind. The classic example of using a recursive algorithm to solve problems is the Tower of Hanoi.

#2 Divide and Conquer Algorithm

Traditionally, the divide and conquer algorithm consists of two parts:

1. breaking down a problem into some smaller independent sub-problems of the same type; 2. finding the final solution of the original issues after solving these more minor problems separately. The key points of the divide and conquer algorithm are:

- If you can find the repeated sub-problems and the loop substructure of the original problem, you may quickly turn the original problem into a small, simple issue.
- Try to break down the whole solution into various steps (different steps need different solutions) to make the process easier.
- Are sub-problems easy to solve? If not, the original problem may cost lots of time.

#3 Dynamic Programming Algorithm

Developed by Richard Bellman in the 1950s, the dynamic programming algorithm is generally used for optimization problems. In this type of algorithm, past results are collected for future use. Like the divide and conquer algorithm, a dynamic programming algorithm simplifies a complex problem by breaking it down into some simple sub-problems. However, the most significant difference between them is that the latter requires overlapping sub-problems, while the former doesn't need to.

#4 Greedy Algorithm

This is another way of solving optimization problems – greedy algorithm. It refers to always finding the best solution in every step instead of considering the overall optimality. That is to say, what he has done is just at a local optimum. Due to the

limitations of the greedy algorithm, it has to be noted that the key to choosing a greedy algorithm is whether to consider any consequences in the future.

#5 Brute Force Algorithm

The brute force algorithm is a simple and straightforward solution to the problem, generally based on the description of the problem and the definition of the concept involved. You can also use “just do it!” to describe the strategy of brute force. In short, a brute force algorithm is considered as one of the simplest algorithms, which iterates all possibilities and ends up with a satisfactory solution.

#6 Backtracking Algorithm

Based on a depth-first recursive search, the backtracking algorithm focusing on finding the solution to the problem during the enumeration-like searching process. When it cannot satisfy the condition, it will return “backtracking” and tries another path. It is suitable for solving large and complicated problems, which gains the reputation of the “general solution method”. One of the most famous backtracking algorithm example is the eight queens puzzle.

1.7 Use Flowcharts to Represent Algorithms

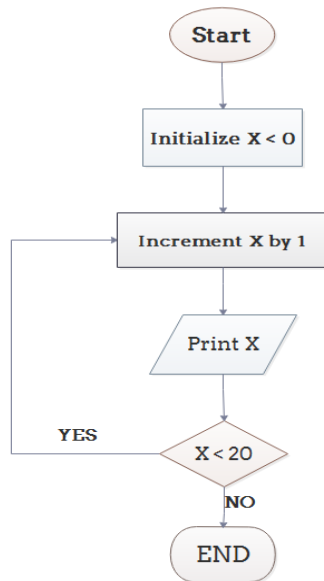
Use Flowcharts to Represent Algorithms

Example 1: Print 1 to 20:

Algorithm:

- Step 1: Initialize X as 0,
- Step 2: Increment X by 1,
- Step 3: Print X,
- Step 4: If X is less than 20 then go back to step 2.

Flowchart:

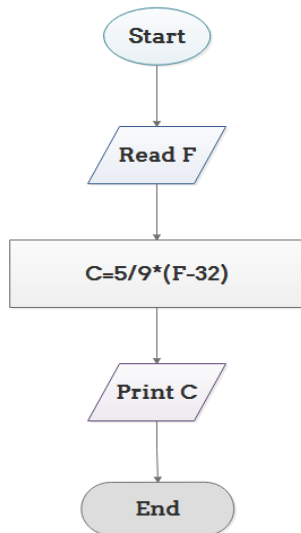


Example 2: Convert Temperature from Fahrenheit (°F) to Celsius (°C)

Algorithm:

- Step 1: Read temperature in Fahrenheit,
- Step 2: Calculate temperature with formula $C=5/9*(F-32)$,
- Step 3: Print C.

Flowchart:



Characteristics of an algorithm

1. **Input:** An algorithm must have zero or more inputs that define the problem to be solved or the data to be processed. The input can be in the form of values, variables, or any other data structures.
2. **Output:** An algorithm must have a well-defined output, which can be a single value, multiple values, or a set of values. The output must be related to the problem that the algorithm is trying to solve.
3. **Definiteness:** An algorithm must have a well-defined set of steps or instructions that are executed in a specific order. These steps must be clear and unambiguous.
4. **Finiteness:** The algorithm must terminate after a finite number of steps. It must not run indefinitely or get stuck in an infinite loop.
5. **Feasibility:** An algorithm must be implementable, meaning that it can be translated into a program that can be executed on a computer.
6. **Effectiveness:** The algorithm must be efficient and solve the problem in a reasonable amount of time and with a reasonable amount of resources, such as memory and computational power.
7. **Generality:** The algorithm must be able to solve a wide range of problems or process a wide range of data, not just specific cases.
8. **Optimality:** An algorithm can be optimal if it produces the best possible solution for a given problem, or if it produces the solution in the minimum amount of time or with the minimum amount of resources.

In summary, an algorithm is a set of well-defined, finite, and effective steps or instructions for solving a problem or processing data, which must have inputs and outputs, be feasible and implementable, and have a level of generality and optimality.

Algorithms and flowcharts (Real Life Examples)

Algorithms and flowcharts are tools that are commonly used in a variety of real-life situations to represent and solve problems in a structured and efficient manner. Here are a few examples of how algorithms and flowcharts are used in real life:

1. **Cooking recipes:** Cooking recipes are often written as algorithms, with each step represented in a clear and sequential manner. For example, a recipe for making cookies might include steps such as: preheat oven, mix ingredients, roll dough, cut into shapes, bake for a specified time, and cool on a wire rack.
2. **Banking transactions:** Banks use algorithms and flowcharts to process transactions and manage customer accounts. For example, a flowchart might represent the steps involved in processing a customer deposit, including verifying the customer's identity, verifying the deposit amount, updating the customer's account balance, and printing a receipt.
3. **GPS navigation:** GPS navigation systems use algorithms and flowcharts to determine the most efficient route to a destination. For example, a flowchart might represent the steps involved in calculating the shortest route, including determining the starting and ending points, considering factors such as traffic and road conditions, and providing turn-by-turn instructions.
4. **Sorting and searching algorithms:** Sorting and searching algorithms are commonly used in real life to organize and find information. For example, a search algorithm might be used to find a specific item in a large database, while a sorting algorithm might be used to sort a list of names alphabetically.
5. **Manufacturing processes:** Manufacturing processes often use algorithms and flowcharts to represent the steps involved in producing a product. For example, a flowchart might represent the steps involved in making a car, including assembling the engine, installing the transmission, adding the wheels and body, and painting the car.

1.8. Pseudocode

A pseudocode (pronounced Soo-doh-kohd) is another way of representing an algorithm. It is considered as a non-formal language that helps programmers to write algorithm. It is a detailed description of instructions that a computer must follow in a particular order. It is intended for human reading and cannot be

executed directly by the computer. No specific standard for writing a pseudocode exists. The word “pseudo” means “not real,” so “pseudocode” means “not real code”. Following are some of the frequently used keywords while writing pseudocode:

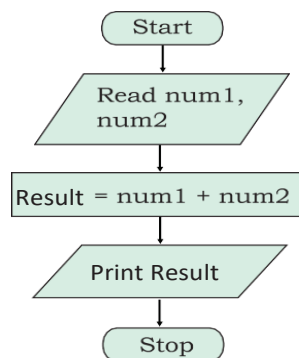
INPUT
COMPUTE
PRINT
INCREMENT
DECREMENT
IF/ELSE
WHILE
TRUE/FALSE

Example 1: Write an algorithm to display the sum of two numbers entered by user, using both pseudocode and flowchart.

Pseudocode for the sum of two numbers will be:

INPUT num1
INPUT num2
COMPUTE Result = num1 + num2
PRINT Result

The flowchart for this algorithms is given below



Example 2: Write an algorithm to calculate area and perimeter of a rectangle, using both pseudocode and flowchart.

Pseudocode for calculating area and perimeter of a rectangle.

INPUT length

INPUT breadth

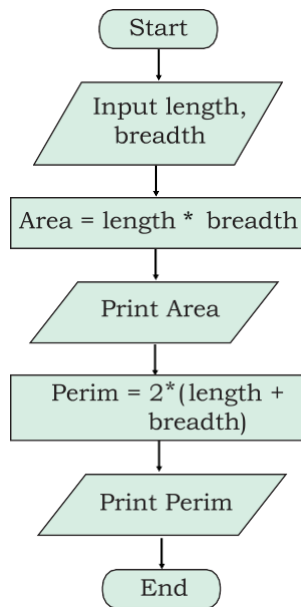
COMPUTE Area = length * breadth

PRINT Area

COMPUTE Perim = 2 * (length + breadth)

PRINT Perim

The flowchart for this algorithm is given below



(A) Benefits of Pseudocode

Before writing codes in a high level language, a pseudocode of a program helps in representing the basic functionality of the intended program. By writing the code first in a human readable language, the programmer safeguards against leaving out any important step. Besides, for non-programmers, actual programs are difficult to read and understand, but

pseudocode helps them to review the steps to confirm that the proposed implementation is going to achieve the desired output.

Conditionals in pseudo-code

Conditional statements allow the execution of code to be dependent on certain conditions being met. In pseudo-code, conditionals are typically expressed using the keywords “if” and “else”.

The basic syntax of an “if” statement in pseudo-code is as follows:

```
If (condition)
then (action to be taken if condition is true)
```

For example, consider a program that checks whether a number is positive or negative:

```
input x
if (x > 0)
then print “x is positive”
```

If the condition is true, the code inside the “if” statement will be executed. If the condition is false, the code inside the “if” statement will be skipped.

An “if-else” statement allows for two different actions to be taken, depending on whether the condition is true or false. The syntax of an “if-else” statement in pseudo-code is as follows:

```
if (condition)
then (action to be taken if condition is true)
else
then (action to be taken if condition is false)
```

For example, consider a program that checks whether a number is positive, negative, or zero:

```
input x
if (x > 0)
then print “x is positive”
else if (x < 0)
```

then print “x is negative”

else

then print “x is zero”

In this example, if the condition “ $x > 0$ ” is true, the code inside the first “if” statement will be executed. If the condition “ $x > 0$ ” is false, the program will move on to the next condition, “ $x < 0$ ”. If this condition is true, the code inside the second “if” statement will be executed. If both conditions are false, the code inside the “else” statement will be executed.

These are the basic concepts of conditionals in pseudo-code. The use of conditionals is essential in programming for controlling the flow of a program and making decisions based on the input data.

loops in pseudo code

Loops are a powerful programming construct that allow the repetition of a set of instructions multiple times, until a certain condition is met. In pseudo-code, loops are typically expressed using the keywords “for” or “while”.

A “for” loop is used to repeat a set of instructions a specific number of times. The basic syntax of a “for” loop in pseudo-code is as follows:

for i = 1 to n

do (action to be repeated n times)

For example, consider a program that prints the first 10 positive integers:

for i = 1 to 10

do print i

In this example, the “for” loop will repeat the instruction “print i” 10 times, and each time the value of “i” will be incremented by 1.

A “while” loop is used to repeat a set of instructions as long as a certain condition is true. The basic syntax of a “while” loop in pseudo-code is as follows:

while (condition)

do (action to be repeated while condition is true)

For example, consider a program that prints the positive integers until a certain number is reached:

```
input max
i = 1
while (i <= max)
do
print i
i = i + 1
```

In this example, the “while” loop will repeat the instructions “print i” and “i = i + 1” as long as the condition “i <= max” is true. The loop will terminate when “i” is no longer less than or equal to “max”.

These are the basic concepts of loops in pseudo-code. The use of loops is essential in programming for repeating actions, processing large amounts of data, and automating tasks.

Time complexity

Time complexity is a measure of the amount of time an algorithm takes to complete, as a function of the size of the input data. It provides a way to compare the performance of different algorithms and to evaluate the efficiency of a particular algorithm.

The time complexity of an algorithm is typically expressed using big O notation, which provides an upper bound on the number of operations performed by the algorithm as a function of the size of the input. For example, an algorithm with a time complexity of $O(n)$ is said to have a linear time complexity, meaning that the number of operations performed is proportional to the size of the input data.

A common example of a linear time complexity algorithm is a linear search. In a linear search, an algorithm checks each element in an array one by one until it finds the target element. The time complexity of this algorithm is $O(n)$, because the number of operations required to find the target element increases linearly with the size of the array.

Another example is a binary search, which is an algorithm for finding an element in a sorted array. The time complexity of binary search is $O(\log n)$, because the number of operations required to find the target element decreases logarithmically with the size of the array. This makes binary search a much faster algorithm than linear search for large arrays.

In summary, time complexity is a critical aspect of algorithm design and analysis. Understanding the time complexity of an algorithm is important for evaluating its performance, comparing it to other algorithms, and optimizing its efficiencies.

Big-Oh notation

Big-Oh notation, also known as big O notation, is a mathematical notation used to describe the upper bound on the growth rate of the time complexity of an algorithm. It provides a way to compare the performance of different algorithms and to evaluate the efficiency of a particular algorithm.

Big-Oh notation is expressed as $O(f(n))$, where “ $f(n)$ ” is a function of the size of the input data. For example, an algorithm with a time complexity of $O(n)$ is said to have a linear time complexity, meaning that the number of operations performed by the algorithm is proportional to the size of the input data.

Big-Oh notation only provides an upper bound on the growth rate of the time complexity, and does not provide an exact measure of the running time of an algorithm. For example, an algorithm with a time complexity of $O(n)$ may take 100 operations for an input of size 100, but only 10 operations for an input of size 10. The big O notation only indicates that the number of operations will not grow faster than linearly with the size of the input.

Big-Oh notation is a useful tool for comparing the performance of different algorithms and for evaluating the efficiency of a particular algorithm. Some common time complexities expressed using big O notation include $O(1)$ for constant time, $O(\log n)$ for logarithmic time, $O(n)$ for linear time, $O(n \log n)$ for log-linear time, and $O(n^2)$ for quadratic time.

In summary, big O notation is a mathematical notation used to describe the upper bound on the growth rate of the time complexity of an algorithm. It provides a way to compare the performance of different algorithms and to evaluate the efficiency of a particular algorithm

Self-Assessment Questions

1. Concept of Problem Solving:

- What are the key steps involved in problem solving?
- Why is it important to understand the problem before trying to solve it?

2. Steps for Problem Solving:

- Describe a scenario where you had to identify and analyze a problem. What steps did you follow to resolve it?
- How do you determine which solution is the best among several options?

3. Problem Solving Techniques:

- List and briefly explain at least three problem-solving techniques. How do they differ from each other?
- When might you use brainstorming versus trial and error in problem solving?

4. Algorithms:

- What is an algorithm, and why is it important in problem solving?
- Provide an example of a simple algorithm and explain its purpose.

5. Flowcharts:

- What are the main symbols used in flowcharts, and what do they represent?
- How does using a flowchart help in understanding and implementing an algorithm?

6. Using Flowcharts to Represent Algorithms:

- Create a flowchart for a simple algorithm, such as making a cup of tea. What are the key steps and decisions involved?

- How can flowcharts be used to debug an algorithm?

7. **Pseudo Code:**

- What is pseudo code, and how does it differ from actual programming code?
- Write a pseudo code example for an algorithm that calculates the factorial of a number.

8. **Types of Algorithms:**

- Compare and contrast two different types of algorithms, such as sorting algorithms versus search algorithms.
- Why is it important to understand different types of algorithms when solving problems?

9. **Implementation and Evaluation:**

- How would you go about implementing a solution once you have chosen the best one?
- What methods can you use to evaluate the effectiveness of a solution?

10. **Application:**

- Describe a real-life problem you've encountered and outline how you would apply the problem-solving steps, techniques, algorithms, and tools like flowcharts and pseudo code to address it.

Activities and Exercises

1. **Problem-Solving Steps Exercise:**

- **Activity:** Select a common problem you've encountered recently (e.g., scheduling a project, organizing a team event). Apply the problem-solving steps: identify, analyze, develop solutions, choose the best solution, implement it, and evaluate the results.
- **Objective:** Practice and understand each step in a real-world context.

2. **Flowchart Creation:**

- **Exercise:** Create a flowchart for a daily routine, such as preparing breakfast or getting ready for work. Use standard flowchart symbols to represent each step and decision.
 - **Objective:** Learn how to visually represent processes and decision-making.
- 3. Algorithm Design and Implementation:**
- **Exercise:** Write an algorithm in pseudo code for a task like calculating the average of a list of numbers. Then, create a flowchart for the same algorithm.
 - **Objective:** Understand how to convert between different forms of algorithm representation and practice writing pseudo code.
- 4. Problem-Solving Technique Application:**
- **Activity:** Form a small group and tackle a given problem (e.g., optimizing a classroom layout). Use different problem-solving techniques (e.g., brainstorming, root cause analysis) to generate and evaluate potential solutions.
 - **Objective:** Compare the effectiveness of various problem-solving techniques in a collaborative setting.
- 5. Algorithm Comparison:**
- **Exercise:** Research and compare two different sorting algorithms (e.g., Bubble Sort and Quick Sort). Create a simple flowchart for each algorithm and discuss their advantages and disadvantages.
 - **Objective:** Gain insight into different types of algorithms and their practical applications.

Case Studies

- 1. Case Study: Improving Customer Service**
- **Scenario:** A retail store is experiencing long wait times at checkout. The management wants to reduce the wait time and improve customer satisfaction.

- **Tasks:**
 - Identify the problem and analyze the potential causes.
 - Develop several solutions (e.g., adding more checkout counters, implementing self-checkout kiosks).
 - Choose the best solution and create a flowchart to represent the new checkout process.
 - Write pseudo code for the process changes and describe how the solution will be implemented.
- **Objective:** Apply problem-solving steps and techniques to a practical business scenario.

2. Case Study: Website Navigation Improvement

- **Scenario:** A website is experiencing high bounce rates and user complaints about navigation difficulty. The goal is to redesign the navigation to make it more user-friendly.
- **Tasks:**
 - Analyze the current navigation structure and user feedback.
 - Brainstorm potential improvements and solutions (e.g., redesigning the menu, adding a search feature).
 - Develop an algorithm for the proposed changes and create a flowchart to visualize the new navigation process.
 - Write pseudo code for any new functionality added to the website.
- **Objective:** Practice identifying and solving user experience problems using structured techniques.

3. Case Study: Inventory Management System

- **Scenario:** A company needs to develop a new inventory management system to track products, manage stock levels, and generate reports.
- **Tasks:**
 - Identify the key requirements and problems with the current system.

- Design an algorithm to handle inventory updates and stock management.
 - Create a flowchart to illustrate the inventory tracking process.
 - Develop pseudo code for critical functions such as updating stock levels and generating reports.
 - **Objective:** Understand how to develop and document a system using problem-solving tools and techniques.
4. **Case Study: Scheduling Optimization**
- **Scenario:** A manufacturing plant needs to optimize its production schedule to reduce downtime and improve efficiency.
 - **Tasks:**
 - Analyze the current scheduling process and identify inefficiencies.
 - Generate potential solutions (e.g., adjusting shift patterns, improving machine maintenance schedules).
 - Develop an algorithm for the optimized scheduling process and create a flowchart to represent the changes.
 - Write pseudo code for scheduling algorithms and describe how the new schedule will be implemented.
 - **Objective:** Apply problem-solving methods to improve operational efficiency in a manufacturing context.

Summary:

1. **Introduction to Problem Solving Concept:** Problem solving is a systematic approach to finding solutions for specific issues or challenges. It involves identifying the problem, understanding its nature, and developing strategies to resolve it. The goal is to achieve a desired outcome or improve a situation by applying logical reasoning and analysis.
2. **Steps for Problem Solving:**
 - **Identify the Problem:** Clearly define what the issue is.

- **Analyze the Problem:** Understand the root cause and the factors involved.
- **Generate Potential Solutions:** Brainstorm and consider various ways to address the problem.
- **Evaluate and Select Solutions:** Assess the feasibility and effectiveness of each solution and choose the best one.
- **Implement the Solution:** Put the chosen solution into action.
- **Review and Reflect:** Evaluate the effectiveness of the solution and make adjustments if necessary.

3. Problem Solving Techniques:

- **Trial and Error:** Testing different solutions until finding one that works.
- **Divide and Conquer:** Breaking the problem into smaller, more manageable parts.
- **Heuristics:** Using rules of thumb or educated guesses to simplify the problem-solving process.
- **Brainstorming:** Generating a wide range of ideas and solutions without immediate evaluation.
- **Root Cause Analysis:** Identifying the underlying cause of the problem to address it more effectively.

4. Algorithms, Flowcharts, Types of Algorithms:

- **Algorithms:** Step-by-step procedures or formulas for solving a problem or performing a task. They provide a clear sequence of actions to achieve a desired outcome.
- **Flowcharts:** Visual representations of algorithms, using symbols and arrows to illustrate the flow of steps and decisions in a process.
- **Types of Algorithms:** Include sorting algorithms (e.g., quicksort, mergesort), searching algorithms (e.g., binary search), and algorithms for specific tasks (e.g., Dijkstra's algorithm for shortest paths).

5. Use Flowcharts to Represent Algorithms:

Flowcharts help in visually organizing the steps of an algorithm. They use different symbols like ovals (for

start/end), rectangles (for processes), diamonds (for decisions), and arrows (for flow direction) to map out the sequence of operations and decisions involved in the algorithm.

6. **Pseudo Code:** Pseudo code is a high-level description of an algorithm that uses simple language and structure, resembling programming code but not bound by syntax rules of specific programming languages. It helps in planning and conceptualizing algorithms before coding, focusing on the logic and steps involved rather than specific coding syntax.

Glossary

- **Algorithm:** A step-by-step procedure or set of rules designed to perform a specific task or solve a particular problem. Algorithms are fundamental to computer science and programming.
- **Analyze the Problem:** The process of breaking down a problem into its components to understand its nature and underlying causes, which helps in devising effective solutions.
- **Choose the Best Solution:** The stage in problem-solving where different potential solutions are evaluated based on criteria such as effectiveness, feasibility, and resources before selecting the most suitable one.
- **Develop Solutions:** The phase of brainstorming or creating potential strategies and approaches to address the identified problem.
- **Evaluate the Results:** The process of assessing the outcome of the implemented solution to determine if the problem has been resolved and making any necessary adjustments.
- **Flowchart:** A graphical representation of a process or algorithm using various symbols (such as rectangles, diamonds, and ovals) connected by arrows to depict the flow of control or steps involved.

- **Identify the Problem:** The initial step in problem solving, which involves recognizing and defining the issue or challenge that needs to be addressed.
- **Implement the Solution:** The action phase where the chosen solution is put into effect in order to solve the problem.
- **Problem Solving:** A methodical approach to finding solutions to complex or challenging issues, involving identification, analysis, solution development, implementation, and evaluation.
- **Problem Solving Techniques:** Various methods used to address and solve problems, such as brainstorming, trial and error, and systematic approaches.
- **Pseudo Code:** A high-level description of an algorithm using a mix of natural language and programming constructs to outline the logic and sequence of steps without adhering to the syntax of any particular programming language.
- **Steps for Problem Solving:** The sequential phases in the problem-solving process: identifying, analyzing, developing, choosing, implementing, and evaluating.
- **Types of Algorithms:** Various kinds of algorithms, including simple ones (like those for sorting and searching) and more complex ones (such as those used in machine learning and artificial intelligence).

UNIT II

UNIT 2 - Features of C

Features of C : Introduction-Character Set - Structure of a 'C' Program - Data Types in 'C' - Operations – Expressions - Assignment Statement - Conditional Statements - Structure for Looping Statements - Nested Looping Statements - Multi Branching Statement (Switch), Break and Continue - Differences between Break and Continue - Unconditional Branching (Go to Statement)

TABLE OF CONTENTS		
UNIT	TOPICS	PAGE
2	2.1. Introduction	41
	2.1.1. Features of C	41
	2.2. Character Set	42
	2.3. Structure of a 'C' Program	43
	2.4. Data Types in 'C'	44
	2.5. Operations	52
	2.6. Expressions	59
	2.7. Assignment Statement	63
	2.8. Conditional Statements	64
	2.9. Structure for Looping Statements	68
	2.10. Nested Looping Statements	71
	2.11. Multi Branching Statement (Switch), Break and Continue	72
	2.12. Differences between Break and Continue	74
2.13. Unconditional Branching	75	

UNIT OBJECTIVES:

- To understand the importance of C Language
- To understand various data types
- To understand working function of input and output statements in C
- To understand working function of Branching statements in C
- To understand working function of Looping statements in C
- To Understand differences between Break and Continue

2.1 Introduction

'C' is high level language and is the upgraded version of another language (Basic Combined Program Language). C language was designed at Bell laboratories in the early 1970's by Dennis Ritchie. C being popular in the modern computer world can be used in Mathematical Scientific, Engineering and Commercial applications

The most popular Operating system UNIX is written in C language. This language also has the features of low level languages and hence called as "System Programming Language"

2.1.1.Features of C language

- Simple, versatile, general purpose language
- It has rich set of Operators
- Program execution are fast and efficient
- Can easily manipulates with bits, bytes and addresses
- Varieties of data types are available
- Separate compilation of functions is possible and such functions can be called by any C program
- Block- structured language
- Can be applied in System programming areas like operating systems, compilers & Interpreters, Assembles, Text Editors, Print Spoolers, Network Drivers, Modern Programs, Data Bases, Language Interpreters, Utilities etc.

2.2 Character Set

The character set is the fundamental raw-material for any language. Like natural languages, computer languages will also have well defined character-set, which is useful to build the programs.

The C language consists of two character sets namely – source character set execution character set. Source character set is useful to construct the statements in the source program. Execution character set is employed at the time of execution of h program.

1. Source character set : This type of character set includes three types of characters namely alphabets, Decimals and special symbols.

- i. Alphabets : A to Z, a to z and Underscore(_)
- ii. Decimal digits : 0 to 9
- iii. Special symbols: + - * / ^ % = & ! () { } [] “ etc

2. Execution character set :

This set of characters are also called as non-graphic characters because these are invisible and cannot be printed or displayed directly.

These characters will have effect only when the program being executed. These characters are represented by a back slash (\) followed by a character.

Execution character	Meaning	Result at the time of execution
\n	End of a line	Transfers the active position of cursor to the initial position of next line
\0 (zero)	End of string	Null
\t	Horizontal Tab	Transfers the active position of cursor to the next Horizontal Tab
\v	Vertical Tab	Transfers the active position of cursor to the next Vertical Tab

\f	Form feed	Transfers the active position of cursor to the next logical page
\r	Carriage return	Transfers the active position of cursor to the initial position of current line

2.3 Structure of a 'C' Program

The Complete structure of C program is

The basic components of a C program are:

- main()
- pair of braces { }
- declarations and statements
- user defined functions

Preprocessor Statements:

These statements begin with # symbol. They are called preprocessor directives. These statements direct the C preprocessor to include header files and also symbolic constants in to C program. Some of the preprocessor statements are

#include<stdio.h>: for the standard input/output functions #include<test.h>: for file inclusion of header file Test.

#define NULL 0: for defining symbolic constant NULL = 0 etc.

Global Declarations:

Variables or functions whose existence is known in the main() function and other user defined functions are called global variables (or functions) and their declarations is called global declaration. This declaration should be made before main().

main():

As the name itself indicates it is the main function of every C program. Execution of C program starts from main (). No C program is executed without main() function. It should be written in lowercase letters and should not be terminated by a semicolon. It calls other Library

functions user defined functions. There must be one and only one main() function in every C program.

Braces:

Every C program uses a pair of curly braces ({,}). The left brace indicates beginning of main() function. On the other hand, the right brace indicates end of the main() function. The braces can also be used to indicate the beginning and end of user-defined functions and compound statements.

Declarations:

It is part of C program where all the variables, arrays, functions etc., used in the C program are declared and may be initialized with their basic data types.

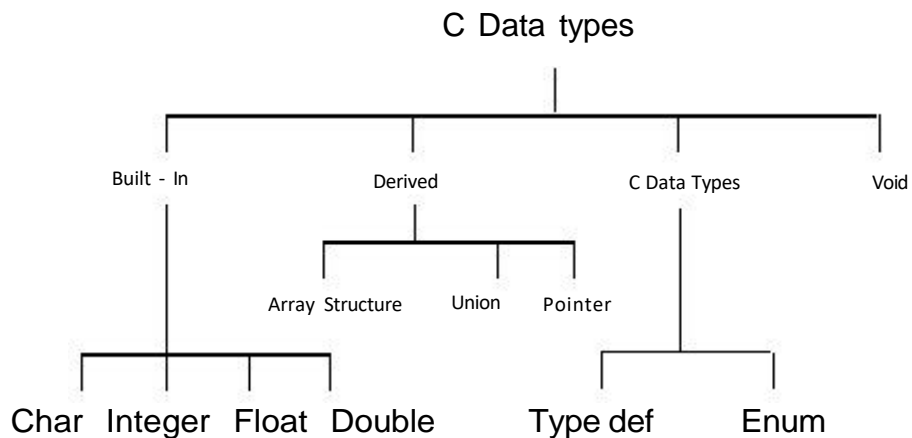
Statements:

These are instructions to the specific operations. They may be input-output statements, arithmetic statements, control statements and other statements. They are also including comments.

User-defined functions:

These are subprograms. Generally, a subprogram is a function, and they contain a set of statements to perform a specific task. These are written by the user; hence the name is user-defined functions. They may be written before or after the main () function.

2.4 Data Types in 'C'



The built-in data types and their extensions is the subject of this chapter. Derived data types such as arrays, structures, union and pointers and user defined data types such as typedef and enum.

Basic Data Types

There are four basic data types in C language. They are Integer data, character data, floating point data and double data types.

a. Character data:

Any character of the ASCII character set can be considered as a character data types and its maximum size can be 1 byte or 8 byte long. 'Char' is the keyword used to represent character data type in C.

Char - a single byte size, capable of holding one character.

a. Integer data:

b. The keyword 'int' stands for the integer data type in C and its size is either 16 or 32 bits.

The integer data type can again be classified as

1. Long int - long integer with more digits
2. Short int - short integer with fewer digits.
3. Unsigned int - Unsigned integer
4. Unsigned short int – Unsigned short integer
5. Unsigned long int – Unsigned long integer

As above, the qualifiers like short, long, signed or unsigned can be applied to basic data types to derive new data types.

int - an Integer with the natural size of the host machine.

c. **Floating point data:** - The numbers which are stored in floating point representation with mantissa and exponent are called floating point (real) numbers. These numbers can be declared as 'float' in C.

float –Single – precision floating point number value.

d. **Double data** : - Double is a keyword in C to represent double precision floating point numbers.

double - Double – precision floating point number value.

Data Kinds in C

Various data kinds that can be included in any C program can fall in to the following.

- a. Constants/Literals
 - b. Reserve Words Keywords
 - c. Delimiters
 - d. Variables/Identifiers
- a. **Constans/Literals:** Constants are those, which do not change, during the execution of the program. Constants may be categorized in to:
- Numeric Constants
 - Character Constants
 - String Constants

1. Numeric Constants

Numeric constants, as the name itself indicates, are those which consist of numerals, an optional sign and an optional period. They are further divided into two types:

- (a) Integer Constants (b) Real Constants

a. Integer Constants

A whole number is an integer constant Integer constants do not have a decimal point. These are further divided into three types depending on the number systems they belong to. They are:

- i. Decimal integer constants
- ii. Octal integer constants
- iii. Hexadecimal integer constants

i. **A decimal integer** constant is characterized by the following properties

- It is a sequence of one or more digits ([0...9], the symbols of decimal number system).
- It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it. Some examples of valid decimal integer constants: 456 -123

Some examples of invalid decimal integer constants:

4.56 - Decimal point is not permissible 1,23 - Commas are not permitted

ii. An octal integer constant is characterized by the following properties

- It is a sequence of one or more digits ([0...7], symbols of octal number system).
- It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the digit 0.
- Commas and blank spaces are not permitted.

• It should not have a period as part of it. Some examples of valid octal integer constants:

0456

-0123

+0123

Some examples of invalid octal integer constants:

04.56 - Decimal point is not permissible 04,56 - Commas are not permitted

x34 - x is not permissible symbol 568 - 8 is not a permissible symbol

iii. An hexadecimal integer constant is characterized by the following properties

- It is a sequence of one or more symbols ([0...9][A...Z], the symbols of Hexadecimal number system).
- It may have an optional + or - sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the symbols 0X or 0x.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid hexadecimal integer constants: 0x456

-0x123 0x56A 0XB78

Some examples of invalid hexadecimal integer constants: 0x4.56 - Decimal point is not permissible

0x4,56 - Commas are not permitted.

b. Real Constants

The real constants also known as *floating point constants* are written in two forms:

(i) Fractional form, (ii) Exponential form.

i. Fractional Form

The real constants in Fractional form are characterized by the following characteristics:

- Must have at least one digit.
- Must have a decimal point.
- May be positive or negative and in the absence of sign taken as positive.
- Must not contain blanks or commas in between digits.
- May be represented in exponential form, if the value is too higher or too low.

Some examples of valid real constants:

456.78

-123.56

Some examples of invalid real constants:

4.56 - Blank spaces are not permitted 4,56 -Commas are not permitted

456 - Decimal point missing

ii. Exponential Form

The exponential form offers a convenient way for writing very large and small real constant. For example, 56000000.00, which can be written as 0.56×10^8

, 10^8 is written as $0.56E8$ or $0.56e8$ in exponential form. 0.000000234 , which can be written as 0.234×10^{-6} is written as $0.234E-6$ or $0.234e-6$ in exponential form. The letter E or e stand for exponential form.

A real constant expressed in exponential form has two parts: (i) Mantissa part, (ii) Exponent part. Mantissa is the part of the real constant to the left of E or e, and the Exponent of a real constant is to the right of E or e. Mantissa and Exponent of the above two number are shown below.

Mantissa	Exponent	Mantissa	Exponent
----------	----------	----------	----------

0.56	E 8	0.234	E -6
------	-----	-------	------

In the above examples, 0.56 and 0.234 are the mantissa parts of the first and second numbers, respectively, and 8 and -6 are the exponent parts of the first and second number, respectively.

The real constants in exponential form are characterized by the following characteristics:

- The mantissa must have at least one digit.
- The mantissa is followed by the letter E or e and the exponent.
- The exponent must have at least one digit and must be an integer.
- A sign for the exponent is optional. Some examples of valid real constants:

3E4

23e-6

0.34E6

Some examples of invalid real constants:

23E - No digit specified for exponent

23e4.5 - Exponent should not be a fraction 23,4e5 - Commas are not allowed

256*e8- * not allowed

2. Character Constants

Any character enclosed with in single quotes (') is called character constant.

A character constant:

- May be a single alphabet, single digit or single special character placed with in single quotes.
- Has a maximum length of 1 character.

Here are some examples,

- 'C'
- 'c'
- '.'
- '*'

3. String Constants

A string constant is a sequence of alphanumeric characters enclosed in double quotes whose maximum length is 255 characters.

Following are the examples of valid string constants:

- "My name is Krishna"
- "Bible"
- "Salary is 18000.00"

Following are the examples of invalid string constants:

My name is Krishna - Character are not enclosed in double quotation marks.

"My name is Krishna - Closing double quotation mark is missing.

'My name is Krishna' - Characters are not enclosed in double quotation marks

b. Reserve Words/Keywords

In C language , some words are reserved to do specific tasks intended for them and are called Keywords or Reserve words. The list reserve words are

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

c. Delimiters

This is symbol that has syntactic meaning and has got significance. These will not specify any operation to result in a value. C language delimiters list is given below

Symbol	Name	Meaning
#	Hash	Pre-processor directive
,	comma	Variable delimiter to separate variable
:	colon	label delimiter
;	Semicolon	statement delimiter
()	parenthesis	used for expressions
{ }	curly braces	used for blocking of statements
[]	square braces	used along with arrays

d. Variables / Identifiers

These are the names of the objects, whose values can be changed during the program execution. Variables are named with description that transmits the value it holds.

[A quantity of an item, which can be change its value during the execution of program is called variable. It is also known as Identifier].

Rules for naming a variable:-

- It can be of letters, digits and underscore(_)
- First letter should be a letter or an underscore, but it should not be a digit.
- Reserve words cannot be used as variable names.

Example: basic, root, rate, roll-no etc are valid names.

Declaration of variables:

Syntax	type	Variable list
int	i, j	i, j are declared as integers
float	salary	salary is declared ad floating point variable
Char	sex	sex is declared as character variable

2.5 Operators

An Operator is a symbol that operates on a certain data type. The data items that operators act upon are called **operands**. Some operators require two operands, some operators act upon only one operand. In C, operators can be classified into various categories based on their utility and action.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operator
4. Assignment Operator
5. Increment & Decrement Operator
6. Conditional Operator
7. Bitwise Operator
8. Comma Operator

1. Arithmetic Operators

The Arithmetic operators performs arithmetic operations. The Arithmetic operators can operate on any built in data type. A list of arithmetic operators are

Operator Meaning

- + Addition
- Subtraction
- * Multiplication
- / Division
- % Modulo division

2. Relational Operators

Relational Operators are used to compare arithmetic, logical and character expressions. The Relational Operators compare their left hand side expression with their right hand side expression. They evaluate to an integer. If the Expression is false it evaluates to “zero”(0) if the expression is true it evaluates to “one”

Operator Meaning

- < Less than
- > Greater than
- <= Less than or Equal to
- >= Greater than or Equal to
- = = Equal to

!= Not Equal to

The Relational Operators are represented in the following manner:

Expression-1 Relational Operator
Expression-2

The Expression-1 will be compared with Expression -2 and depending on the relation the result will be either "TRUE" OR "FALSE".

Examples :

Expression Evaluate to

$(5 \leq 10)$ _____ 1

$(-35 > 10)$ _____ 0

$(X < 10)$ _____ 1 (if value of x is less than 10) 0 Other wise

$(a + b) == (c + d)$ 1 (if sum of a and b is equal to sum of c, d) 0 Other wise

2. Logical Operators

A logical operator is used to evaluate logical and relational expressions. The logical operators act upon operands that are themselves logical expressions. There are three logical operators.

Operators	Expression
&&	Logical AND
	Logical OR
!	Logical NOT

Logical And (&&): A compound Expression is true when two expression when two expressions are true. The && is used in the following manner.

Exp1 && Exp2.

The result of a logical AND operation will be true only if both operands are true.

The results of logical operators are: Exp1 Op. Exp2 Result

True && True True

True && False False False && False False False && True False

Example: a = 5; b = 10; c = 15;

Exp1	Exp2	Result
1. (a < b) && (b < c) =>	True	
2. (a > b) && (b < c) =>	False	
3. (a < b) && (b > c) =>	False	
4. (a > b) && (b > c) =>	False	

Logical OR: A compound expression is false when all expression are false otherwise the compound expression is true. The operator “||” is used as It evaluates to true if either exp-1 or exp-2 is true. The truth table of “OR” is Exp1

|| Exp2

Exp1 Operator Exp2 Result:

True		True True
True		False True
False		True True
False		False False

Example: a = 5; b = 10; c = 15;

Exp1	Exp2	Result
1. (a < b) (b < c) =>	True	
2. (a > b) (b < c) =>	True	
3. (a < b) (b > c) =>	True	
4. (a > b) (b > c) =>	False	

Logical NOT: The **NOT (!)** operator takes single expression and evaluates to true(1) if the expression is false (0) or it evaluates to false (0) if expression is true (1). The general form of the expression.

!(Relational Expression)

The truth table of

NOT :

Operator. Exp1 Result

! True False

! False True

Example: a = 5; b = 10; c = 15

1. $!(a < b)$ False
2. $!(a > b)$ True
3. Assignment Operator

An assignment operator is used to assign a value to a variable. The most commonly used assignment operator is $=$. The general format for assignment operator is :

$\langle \text{Identifier} \rangle = \langle \text{expression} \rangle$

Where identifier represent a variable and expression represents a constant, a variable or a Complex expression.

If the two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left.

Example: Suppose that **I** is an **Integer** type Variable then

1. $I = 3.33$ (Value of I)
2. $I = 3.93$ (Value of I)
3. $I = 5.745$ (Value of I)

Multiple assignment

$\langle \text{identifier-1} \rangle = \langle \text{identifier-2} \rangle = \dots = \langle \text{identifier-n} \rangle = \langle \text{exp} \rangle;$

Example: a,b,c are integers; j is float variable

1. $a = b = c = 3;$
2. $a = j = 5.6;$ then a = 5 and j value will be 5.6

C contains the following five additional assignment operators

1. $+=$ 2. $-=$ 3. $+=$ 4. $*=$ 5. $/=$

The assignment expression is: - $\text{Exp1} \langle \text{Operator} \rangle \text{Exp-2}$

Ex: $I = 10$ (assume that)

Expression Equivalent to Final Value of 'I'

1. $I += 5 \quad I = I + 5 \quad 15$
2. $I -= 5 \quad I = I - 5 \quad 10$

3. $l * = 5 \quad l = l * 5 \quad 50$

4. $l / = 5 \quad l = l / 5 \quad 10$

4. Increment & Decrement Operator

The increment/decrement operator act upon a Single operand and produce a new value is also called as “**unary operator**”. The increment operator ++ adds 1 to the operand and the Decrement operator – subtracts 1 from the operand.

Syntax: < operator >< variable name >;

The ++ or – operator can be used in the two ways.

Example : ++ a; Pre-increment (or) a++ Post increment —a; Pre- Decrement (or) a— Post decrement

1. **++ a** Immediately increments the value of a by 1.

2. **a ++** The value of the a will be increment by 1 after it is utilized.

Example 1: Suppose a = 5 ;

Statements Output

printf (“a value is %d”, a); a value is 5 printf (“a value is %d”, ++ a); a value is 6 printf (“a value is %d “, a); a value is 6

Statements Output

printf (“a value is %d “, a); a value is 5 printf (“a value is %d “, a++); a value is 5 printf (“a value is %d “,a); a value is 6

a and a- will be act on operand by decrement value like increment operator.

5. Conditional operator (or) Ternary operator (? :)

It is called ternary because it uses three expression. The ternary operator acts like If- Else construction.

Syn :(<Exp-1 > ? <Exp-2> : <Exp-3>);

Expression-1 is evaluated first. If Exp-1 is true then Exp-2 is evaluated other wise it evaluate Exp-3 will be evaluated.

Flow Chart :

Exp-1

Exp-2 Exp-3 Exit **Example:**

1. a = 5 ; b = 3;
(a> b ? printf (“a is larger”) : printf (“b is larger”));
Output is :a is larger
2. a = 3; b = 3;

(a> b ? printf (“a is larger”) : printf (“b is larger”));

Output is :b is larger

6. Bit wise Operator

A bitwise operator operates on each bit of data. These bitwiseoperator can be divided into three categories.

- i. The logical bitwise operators.
 - ii. The shift operators
 - iii. The one’s complement operator.
- i) **The logical Bitwise Operator :**There are three logical bitwise operators.

Meaning Operator:

- a) Bitwise **AND &**
- b) Bitwise **OR |**
- c) Bitwise exclusive **XOR ^**

Suppose b1 and b2 represent the corresponding bits with in the first and second operands, respectively.

B1 B2 B1 & B2 B1 | B2 B1 ^ B2

1 1 1 1 0

1 0 0 1 1

0 1 0 1 1

0 0 0 0 0

The operations are carried out independently on each pair of corresponding bits within the operand thus the least significant bits (ie the right most bits) within the two operands. Will be compared until all the bits have been compared. The results of these comparisons are

A **Bitwise AND** expression will return a 1 if both bits have a value of 1.

Other wise, it will return a value of 0.

A **Bitwise OR** expression will return a 1 if one or more of the bits have a value of 1. Otherwise, it will return a value of 0.

A **Bitwise EXCLUSIVE OR** expression will return a 1 if one of the bits has a value of 1 and the other has a value of 0. Otherwise, it will return a value of 0.

Example::Variable Value Binary Pattern

X 5 0101

Y 2 0010

X & Y 0 0000

X | Y 7 0111

X ^ Y 7 0111

ii) **The Bitwise shift Operations:** The two bitwise shift operators are **Shift left** (<<) and **Shift right** (>>). Each operator requires two operands. The first operand that represents the bit

pattern to be shifted. The second is an unsigned integer that indicates the number of displacements.

Example: $c = a \ll 3;$

The value in the integer a is shifted to the left by three bit position. The result is assigned to the c.

$A = 13; c = A \ll 3;$

Left shit << $c = 13 * 2^3 = 104;$

Binary no 0000 0000 0000 1101

After left bit shift by 3 places ie,. $a \ll 3$ 0000 0000 0110 1000

The right –bit – shift operator (>>) is also a binary operator.

Example: `c = a >> 2 ;`

The value of a is shifted to the right by 2 position insert 0's Right – shift >> drop off 0's

0000 0000 0000 1101

After right shift by 2 places is `a>>2` 0000 0000 0000 0011 `c=13>>2` `c= 13/4=3`

iii) **Bit wise complement:** The complement op. `~` switches all the bits in a binary pattern, that is all the 0's becomes 1's and all the 1's becomes 0's.

variable value Binary patter

`x 23` 0001 0111

`~x` 132 1110 1000

Comma Operator

A set of expressions separated by using commas is a valid construction in c language.

Example `:int i, j; i= (j = 3, j + 2) ;`

The first expression is `j = 3` and second is `j + 2`. These expressions are evaluated from left to right. From the above example `i = 5`.

Size of operator: The operator size operator gives the size of the data type or variable in terms of bytes occupied in the memory. This operator allows a determination of the no of bytes allocated to various Data items

Example `:int i; float x; double d; char c; OUTPUT`

2.6 Expressions

`Printf (“integer : %d\n”, sizeof(i)); Integer : 2` `Printf (“float : %d\n”, sizeof(i)); Float : 4` `Printf (“double : %d\n”, sizeof(i)); double : 8` `Printf (“char : %d\n”, sizeof(i)); character : 1`

An expression can be defined as collection of data object and operators that can be evaluated to lead a single new data object. A data object is a constant, variable or another data object.

Example : `a + b`

$x + y + 6.0$

$3.14 * r * r$

$(a + b) * (a - b)$

The above expressions are called as arithmetic expressions because the data objects (constants and variables) are connected using arithmetic operators.

Evaluation Procedure: The evaluation of arithmetic expressions is as per the hierarchy rules governed by the C compiler. The precedence or hierarchy rules for arithmetic expressions are

1. The expression is scanned from left to right.
2. While scanning the expression, the evaluation preference for the operators are

$*, /, \%$ - evaluated first

$+, -$ - evaluated next

3. To overcome the above precedence rules, user has to make use of

parenthesis. If parenthesis is used, the expression/ expressions with in parenthesis are evaluated first as per the above hierarchy.

Statements

Data Input & Output

An input/output function can be accessed from anywhere within a program simply by writing the function name followed by a list of arguments enclosed in parentheses. The arguments represent data items that are sent to the function.

Some input/output Functions do not require arguments though the empty parentheses must still appear. They are:

	<i>Input Statements</i>	<i>Output Statements</i>
Formatted Unformatted	scanf() getchar()gets()	printf() putchar() puts()

getchar()

Single characters can be entered into the computer using the C library Function **getchar()**. It returns a single character from a standard input device. The function does not require any arguments.

Syntax: <Character variable> = getchar();

Example: char c;

c = getchar();

putchar()

Single characters can be displayed using function **putchar()**. It returns a single character to a standard output device. It must be expressed as an argument to the function.

Syntax: putchar(<character variable>);

Example: char c;

gets()—————

putchar(c);

The function **gets()** receives the string from the standard input device.

Syntax: gets(<string type variable or array of char>); Where s is a string.

The function gets accepts the string as a parameter from the keyboard, till a newline character is encountered. At end the function appends a “null” terminator and returns.

puts()

The function **puts()** outputs the string to the standard output device.

Syntax: puts(s);

Where s is a string that was read with gets();

Example:

```
main()
{
char line[80]; gets(line); puts(line);
}
scanf()
```

scanf() function can be used to input the data into the memory from the standard input device. This function can be used to enter any combination of numerical values, single characters and strings. The function returns number of data items.

Syntax: scanf("control strings", &arg1, &arg2, ..., &argn);

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

Example:

```
#include <stdio.h>
main()
{
char item[20]; int partno; float cost;
scanf("%s %d %f", &item, &partno, &cost);
}
```

Where s, d, f with % are conversion characters. The conversion characters indicate the type of the corresponding data. Commonly used conversion characters from data input.

Conversion Characters	Characters	Meaning
%c		data item is a single character.
%d		data item is a decimal integer.
%f		data item is a floating point value.

%e	data item is a floating point value.
%g	data item is a floating point value.
%h	data item is a short integer.
%s	data item is a string.
%x	data item is a hexadecimal integer.
%o	data item is a octal interger.

printf()

The printf() function is used to print the data from the computer's memory onto a standard output device. This function can be used to output any combination of numerical values, single character and strings.

Syntax: printf("control string", arg-1, arg-2,———arg-n);

Where control string is a string that contains formatted information, and arg-1, arg-2 —— are arguments that represent the output data items.

Example:

```
#include<stdio.h> main()
{
char item[20]; intpartno; float cost;
_____
printf ("%s %d %f", item, partno, cost);
}(Where %s %d %f are conversion characters.)
```

2.7 Assignment Statement

Assignment statement can be defined as the statement through which the value obtained from an expression can be stored in a variable.

The general form of assignment statement is

< variable name> = < arithmetic expression> ; Example: sum = a + b + c;
tot = s1 + s2 + s3; area = ½ * b* h;

2.8 Conditional Statement (if, If-else, for, while, do-while)

Conditional Statements

The conditional expressions are mainly used for decision making. The following statements are used to perform the task of the conditional operations.

- a. if statement.
- b. If-else statement. Or 2 way if statement
- c. Nested else-if statement.
- d. Nested if –else statement.
- e. Switch statement.

a. if statement

The **if statement** is used to express conditional expressions. If the given condition is true then it will execute the statements otherwise skip the statements.

The simple structure of 'if' statement is

- i. If (< conditional expression >) statement-1;
(or)
- ii. If (< conditional expression >)
{
statement-1; statement-2; statement-3;
.....
..... STATEMENT-N
}

The expression is evaluated and if the expression is true the statements will be executed. If the expression is false the statements are skipped and execution continues with the next statements.

Example: a=20; b=10;

if (a > b)

printf ("big number is %d" a);

b. if-else statements

The **if-else** statements is used to execute the either of the two statements depending upon the value of the exp. The general form is

```
if(<exp>
{
Statement-1; Statement -2;
..... "SET-I"
..... Statement- n;
}
else
{
Statement1; Statement 2;
..... "SET-II"
..... Statement n;
}
```

SET - I Statements will be executed if the exp is true. SET – II Statements will be executed if the exp is false. **Example:**

```
if ( a > b )
printf ("a is greater than b"); else
printf ("a is not greater than b");
```

c. Nested else-if statements

If some situations if may be desired to nest multiple **if-else** statements. In this situation one of several different course of action will be selected.

Syntax

```
if ( <exp1> )
Statement-1;
else if ( <exp2> )
```

```

Statement-2;
else if ( <exp3> )
Statement-3;
else
Statement-4;

```

When a logical expression is encountered whose value is true the corresponding statements will be executed and the remainder of the nested else if statement will be bypassed. Thus control will be transferred out of the entire nest once a true condition is encountered.

The final **else** clause will be apply if none of the exp is true.

d. nestedif-else statement

It is possible to nest if-else statements, one within another. There are several different form that nested if-else statements can take.

The most general form of two-layer nesting is

```

if(exp1)
else
if(exp3) Statement-3;
    else Statement-4;
if(exp2) Statement-1;
    else Statement-2;

```

One complete **if-else** statement will be executed if **expression1** is true and another complete **if-else** statement will be executed if **expression1** is false.

e. Switch statement

A switch statement is used to choose a statement (for a group of statement) among several alternatives. The switch statements is useful when a variable is to be compared with different constants and in case it is equal to a constant a set of statements are to be executed.

Syntax:

Switch (exp)

```
{  
case  
case constant-1: statements1;
```

```
constant-2:
```

```
statements2;
```

```
_____  
_____
```

```
default:
```

```
statement n;
```

```
}
```

Where constant1, constant2 — — — are either integer constants or character constants. When the switch statement is executed the exp is evaluated and control is transferred directly to the group of statement whose case label value matches the value of the exp. If none of the case label values matches to the value of the exp then the default part statements will be executed.

If none of the case labels matches to the value of the exp and the default group is not present then no action will be taken by the switch statement and control will be transferred out of the switch statement.

A simple switch statement is illustrated below.

Example 1:

```
main()
```

```
{
```

```
char choice;
```

```
printf("Enter Your Color (Red - R/r, White - W/w)"); choice=getchar();
```


The statements will be executed repeatedly as long as the exp is true. If the exp is false then the control is transferred out of the while loop.

Example:

```
int digit = 1;
While (digit <=5) FALSE
{
printf ("%d", digit); TRUE Cond Exp
Statements; ++digit;
}
```

The while loop is top tested i.e., it evaluates the condition before executing statements in the body. Then it is called entry control loop.

b. do-while statement

The **do-while** loop evaluates the condition after the execution of the statements in the body.

Syntax:do Statement; **While**<exp>;

Here also the statements will be executed as long as the exp value is true. If the expression is false the control come out of the loop.

Example:

```
-int d=1; do
{
printf ("%d", d); FALSE
++d;
} while (d<=5); TRUE Cond Exp
statements exit
```

The statement with in the do-while loop will be executed at least once. So the **do-while** loop is called a bottom tested loop.

c. for statement

The **for** loop is used to executing the structure number of times. The **for** loop includes three expressions. First expression specifies an initial value for an index (initial value), second expression that determines whether or not the loop is continued

(conditional statement) and a third expression used to modify the index (increment or decrement) of each pass.

Note: Generally, for loop used when the number of passes is known in advance.

Syntax: **for** (exp1;exp2;exp3)

{

}

exp2

exp3 Statement –1;

Statement – 2;

—————; FALSE

—————;

Statement - n; TRUE

Statements;

Exit loop exp1 start

Where **expression-1** is used to initialize the controlvariable. This expression is executed this expression is executed is only once at the time of beginning of loop.

Where **expression-2** is a logical expression. If **expression-2** is true, the statements willbe executed, other wise the loop will be terminated. This expression is evaluated before every execution of the statement.

Where **expression-3** is an increment or decrement expression after executing the statements, the control is transferred back to the **expression-3** and updated. There are different formats available in **for loop**. Some of the expression of loop can be omit.

Format - I

for(; exp2; exp3) Statements;

In this format the initialization expression (i.e., **exp1**) is omitted. The initial value of the variable can be assigned outside of the **for loop**.

Example 1

int i = 1;

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

Format - II

```
for( ; exp2 ; ) Statements;
```

In this format the initialization and increment or decrement expression (i.e **expression-1** and **expression-3**) are omitted. The exp-3 can be given at the statement part.

Example 2

```
int i = 1;
```

```
for( ; i<=10; )
```

```
{
```

```
printf ("%d\n",i); i++;
```

```
}
```

Formate - III

```
for( ; ; ) Statements;
```

In this format the **three expressions** are omitted. The loop itself assumes the **expression-2** is true. So **Statements** will be executed infinitely.

Example 3

```
int i = 1;
```

```
for ( ; i<=10; )
```

```
{
```

```
printf ("%d\n",i); i++;
```

```
}
```

2.10 Nested Looping Statements

Many applications require nesting of the loop statements, allowing on loop statement to be embedded with in another loop statement.

Definition

Nesting can be defined as the method of embedding one control structure with in another control structure.

While making control structures to be reside one with in another ,the inner and outer control structures may be of the same type or may not be of same type. But ,it is essential for us to ensure that one control structure is completely embedded within another.

```
/*program to implement nesting*/ #include <stdio.h>
main()
{

int a,b,c,
for (a=1,a< 2, a++)
{
printf ("%d",a)
for (b=1,b<=2,b++)
{
printf ("%d",b)
for (c=1,c<=2,c++)
{
printf (" My Name is Sunny\n");
}
}
}
}
```

2.11 Multi Branching Statement (switch), Break, and Continue

For effective handling of the loop structures, C allows the following types of control break statements.

a. Break Statement b. Continue Statement

a. Break Statement

The break statement is used to terminate the control form the loops or to exit from a switch. It can be used within a for, **while**, **do-while**, **for**.

The general format is : break;

If **break** statement is included in a **while**, **do-while** or **for** then control will immediately be transferred out of the loop when the break statement is encountered.

Example

for (; ;) normal loop

```
{  
break Condition within loop  
scanf ("%d",&n); if ( n < -1)  
break;  
sum = sum + n;  
}
```

b.The Continue Statement

The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. Rather, the remaining loop statements are skipped and the proceeds directly to the next pass through the loop. The “**continue**” that can be included with in a **while a do-while and a for loop** statement.

General form : continue;

The **continue** statement is used for the inverse operation of the **break** statement .

Condition with in loop

Remaining part of loop continue

Example

while (x<=100)

```
{  
if (x <= 0)  
{
```

```

printf("zero or negative value found\n"); continue;
}
}

```

The above program segment will process only the positive whenever a zero or negative value is encountered, the message will be displayed and it continue the same loop as long as the given condition is satisfied.

2.12 Differences between Break and Continue

Break	Continue
<p>1. Break is a key word used to terminate the loop or exit from the block. The control jumps to next statement after the loop or block</p> <p>2. Break statements can be used with for, while, do-while, and switch statement. When break is used in nested loops, then only the innermost loop is terminated.</p> <p>3. Syntax: { statement1; statement2; statement3; break; }</p> <p>4. Example :Switch (choice){ Case 'y': printf("yes"); break; Case 'n': printf("NO"); break; }</p> <p>5. When the case matches with the choice entered, the corresponding case block gets executed. When 'break' statement is executed, the control jumps out of the switch statement.</p>	<p>1. Continue is a keyword used for containing the next iteration of the loop</p> <p>2. This statement when occurs in a loop does not terminate it rather skips the statements after this continue statement and the control goes for next iteration. 'Continue' can be used with for, while and do- while loop.</p> <p>3. Syntax: { statement1; continue; statement 2; statement3; break; }</p> <p>4. Example:- l = 1, j=0; While(i<= 7){ l = l + 1; If((l == 6) Continue; j = j + 1; }</p> <p>5. In the above loop, when value of ' i becomes 6' continue statement is executed. So, j= j+1 is skipped and control is transferred to beginning of while loop.</p>

2.13 Unconditional Branching (Go To Statement)

goto statement

The **go to statement** is used to alter the program execution sequence by transferring the control to some other part of the program.

Syntax

Where label is an identifier used to label the target statement to which the control would be transferred the target statement will appear as:

Syntax

```
goto<label>; label : statements;
```

Example 1

```
#include <stdio.h> main();
{
inta,b;
printf ("Enter the two numbers"); scanf ("%d %d",&a,&b);
if (a>b) gotobig; else gotosmall;
big :printf ("big value is %d",a);
gotostop;
small :printf ("small value is %d",b);
gotostop; stop;
}
```

Simple Programs Covering Above Topics

Practice Programs

1. Write a C program to find out smallest value among A, B,C. Ans:

```
include <stdio.h> int a,b,c;
clrscr();
scanf(%d %d %d, &a, &b, &c); if (a<b)
```

```

{
if(a<c)
printf("a is small/n") else
}

```

02. Write a 'C' programe for 5th multiplication table with the help of goto statement.

Ans. #include<stdio.h> main()

```

{
int t, n = 1, P;
Printf("Enter table number:"); Scanf("%d,&t);
A:

```

```

if (n<=10)
{
}
else
}
P=t * n;
Printf("%d * %d = %d \n", t,n,p); n++;
goto A;
printf("Out of range");

```

03. Write a 'C' program to find greatest among three numbers. Ans. #include<stdio.h>

```

void main( )
{
int a,b,c;
printf("enter the values of a,b,c,"); scanf("%d%d%d", &a,&b,&c);
if((a>b)&&(c>b))
{
if(a>c)

```

```

printf("a is the max no"); else
printf("C is the max no");
}
else if ((b>c)&&(a>c))
{
if(b>a)
printf("b is the max no"); else
printf("a is the max no");
}
else if ((b>a)&&(c>a))
{
if(b>c)
printf("b is the max no"); else
printf("C is the max no");
}
}
}

```

Self Assessment Questions:

Understanding the C Language and Its Importance

1. **What are the main advantages of using the C programming language?**
2. **In what types of applications or environments is C particularly useful?**

Various Data Types in C

3. **What are the fundamental data types in C, and how do they differ?**
4. **How would you declare an integer and a floating-point variable in C?**
5. **Explain the difference between float and double data types. When would you use each?**

Input and Output Statements in C

6. **How do you use the printf function to display a string and an integer in C?**
7. **Write a scanf statement to read a floating-point number from user input.**
8. **What are the common format specifiers used with printf and scanf?**

Branching Statements in C

9. **Write an if-else statement that checks if a number is positive or negative.**
10. **How does the switch statement work, and what are its typical use cases?**
11. **Explain how the else if statement differs from a simple if-else structure.**

Looping Statements in C

12. **Write a for loop to print the numbers 1 to 10.**
13. **How does the while loop differ from the do-while loop? Provide an example where do-while is preferred over while.**
14. **What happens if the condition in a while loop is never met?**

Differences Between break and continue Statements

15. **Describe the effect of the break statement in a loop.**
16. **What does the continue statement do in a loop, and how does it affect the loop's execution?**
17. **Write a for loop example that uses both break and continue statements to demonstrate their functionality.**

Activities/ Excercises:

Understanding the C Language and Its Importance

Activity 1: Research Project

- **Objective:** Research the history and evolution of the C language.
- **Instructions:** Write a brief report on how C has influenced modern programming languages and its role in system-level programming. Include examples of major projects or systems developed using C.

Exercise 1: C Language Comparison

- **Objective:** Compare C with another high-level programming language (e.g., Python, Java).
- **Instructions:** Create a table or document highlighting key differences in syntax, performance, and typical use cases between C and the chosen language.

Various Data Types in C

Activity 2: Data Type Exploration

- **Objective:** Experiment with different data types and their storage requirements.
- **Instructions:** Write a C program that declares variables of various data types (int, float, double, char, and long). Print the size of each type using the sizeof operator and discuss the results.

Exercise 2: Type Conversion

- **Objective:** Practice type casting and conversion.
- **Instructions:** Create a program that reads an integer value from the user, converts it to a float, and prints the result. Ensure to handle any potential issues with type conversion.

Input and Output Statements in C

Activity 3: User Interaction Program

- **Objective:** Develop a program that interacts with the user.

- **Instructions:** Write a C program that prompts the user to enter their name, age, and favorite number. Use printf and scanf to display and collect the information, then provide a summary message to the user.

Exercise 3: Formatted Output

- **Objective:** Practice using format specifiers.
- **Instructions:** Create a program that takes multiple pieces of information (e.g., name, height, and weight) and displays them in a formatted table using printf. Experiment with different format specifiers for alignment and precision.

Branching Statements in C

Activity 4: Conditional Logic

- **Objective:** Implement and test conditional logic.
- **Instructions:** Write a program that asks the user for their exam score and provides a grade based on the score (e.g., A, B, C, D, F) using if-else statements. Test the program with various scores to ensure correct grading.

Exercise 4: Switch Case Application

- **Objective:** Use the switch statement for multiple choices.
- **Instructions:** Develop a simple calculator that can perform addition, subtraction, multiplication, or division based on user input. Use the switch statement to handle the operation selection.

Looping Statements in C

Activity 5: Looping Constructs

- **Objective:** Practice using different looping constructs.

- **Instructions:** Write three separate programs that achieve the same task (e.g., printing numbers from 1 to 10) using for, while, and do-while loops. Compare their implementations and discuss their suitability for different scenarios.

Exercise 5: Loop Control

- **Objective:** Implement loops with control statements.
- **Instructions:** Create a program that prints all even numbers between 1 and 50 using a for loop. Modify the program to skip numbers divisible by 10 using the continue statement and to exit the loop early if a certain condition is met using the break statement.

Differences Between break and continue Statements

Activity 6: Loop Control Comparison

- **Objective:** Understand the impact of break and continue.
- **Instructions:** Write a program that demonstrates the use of both break and continue within a loop. For example, create a loop that processes a list of numbers and uses continue to skip over specific values and break to terminate the loop based on a condition.

Exercise 6: Control Flow Scenarios

- **Objective:** Apply break and continue in practical scenarios.
- **Instructions:** Develop a program that simulates a simple game where the user has to guess a number between 1 and 100. Use continue to prompt the user to re-enter guesses that are out of range and break to end the game when the correct number is guessed.

Case Studies

Case Study 1: System Monitoring Tool

- **Objective:** Design a tool using C for system monitoring.
- **Instructions:** Design a basic system monitoring tool that tracks and displays system metrics (e.g., CPU usage, memory usage) using C. Implement features such as user input for selecting metrics to display and periodic updates using loops. Utilize branching statements to handle user choices and control program flow.

Case Study 2: Student Grading System

- **Objective:** Develop a grading system for a classroom.
- **Instructions:** Create a grading system that allows teachers to input student scores, calculates final grades based on specified criteria, and generates a report. Use data types to handle various score ranges and branching statements to assign grades. Include input validation and formatted output to display the results clearly.

Summary :

Understanding the C Language and Its Importance:

C is a powerful, general-purpose programming language known for its efficiency and flexibility. It provides a foundation for learning other languages and is widely used in system programming, application development, and embedded systems due to its performance and close-to-hardware operations.

Various Data Types in C:

Data types in C define the type of data a variable can hold. The primary data types include integers, floating-point numbers, characters, and derived types such as arrays, pointers, and structures. Understanding these data types is crucial for managing and manipulating data effectively in C programs.

Input and Output Statements in C:

C provides functions for input and output operations through the `stdio.h` library. The `printf` function is used for output, while `scanf` is used for input. Mastery of these functions is essential for interacting with users and processing data.

Branching Statements in C:

Branching statements control the flow of execution in a program based on conditions. The primary branching statements in C are `if`, `else`, `else if`, and `switch`. These constructs enable decision-making and facilitate different execution paths based on variable values.

Looping Statements in C:

Looping statements allow repetitive execution of code blocks. The main looping constructs in C are `for`, `while`, and `do-while` loops. Understanding these loops helps in automating repetitive tasks and managing iterations efficiently.

Differences Between `break` and `continue` Statements:

Both `break` and `continue` are control statements used within loops. The `break` statement terminates the loop immediately and exits it, while `continue` skips the remaining code in the current iteration and proceeds to the next iteration of the loop. Knowing when to use each statement is crucial for controlling loop execution flow.

Glossary :

C Language: A general-purpose, high-level programming language known for its efficiency and control over system resources. It is widely used in system programming, application development, and embedded systems.

Data Types: Categories of data that specify the type of values a variable can hold.

Common data types in C include:

- **Integer (`int`):** Represents whole numbers.
- **Float (`float`):** Represents numbers with fractional parts.
- **Double (`double`):** Represents floating-point numbers with double precision.
- **Char (`char`):** Represents single characters.

Derived Data Types: Types built from fundamental data types. Examples include:

- **Array:** A collection of elements of the same type.

- **Pointer:** A variable that stores the address of another variable.
- **Structure (struct):** A user-defined type that groups related variables of different types.

Input Statements: Functions used to read data from the user or other input sources. In C, the primary input function is `scanf`.

Output Statements: Functions used to display data to the user or other output destinations. In C, the primary output function is `printf`.

Branching Statements: Constructs that alter the flow of control based on conditions.

Examples include:

- **if:** Executes a block of code if a specified condition is true.
- **else:** Executes a block of code if the preceding if condition is false.
- **else if:** Specifies a new condition to test if the previous if or else if conditions are false.
- **switch:** Allows the selection of one block of code from multiple options based on the value of an expression.

Looping Statements: Constructs that repeat a block of code multiple times. Examples include:

- **for:** Repeats a block of code a specific number of times.
- **while:** Repeats a block of code as long as a specified condition remains true.
- **do-while:** Repeats a block of code at least once, and then continues repeating as long as a specified condition remains true.

Break Statement: A control statement used to exit from a loop or switch statement prematurely, regardless of the loop's condition.

UNIT III

UNIT 3 - Functions

Functions: Introduction – Functions- Differences between Function and Procedures - Advantages of Functions - Advanced features of Functions – Recursion

TABLE OF CONTENTS		
UNIT	TOPICS	PAGE
3	3.1.Introduction	86
	3.2.Functions	86
	3.3.Differences between Function and Procedures	90
	3.4.Advantages of Functions	91
	3.5.Advanced features of Functions	91
	3.6.Recursion	98

UNIT OBJECTIVES:

- Define a Function
- Stress on Return statement
- Write programs using function call techniques.
- Function prototype
- Differentiate Local and Global variables
- Recursion.

3.1 Introduction

Experienced programmer used to divide large (lengthy) programs in to parts, and then manage those parts to be solved one by one. This method of programming approach is to organize the typical work in a systematic manner. This aspect is practically achieved in C language through the concept known as ‘Modular Programming’.

The entire program is divided into a series of modules and each module is intended to perform a particular task. The detailed work to be solved by the module is described in the module (sub program) only and the main program only contains a series of modules that are to be executed. Division of a main program in to set of modules and assigning various tasks to each module depends on the programmer’s efficiency.

Whereas there is a need for us repeatedly execute one block of statements in one place of the program, loop statements can be used. But, a block of statements need to be repeatedly executed in many parts of the program, then repeated coding as well as wastage of the vital computer resource memory will be wasted. . If we adopt modular

3.2. Functions

programming technique, these disadvantages can be eliminated. The modules incorporated in C are called as the FUNCTIONS, and each function in the program is meant for doing specific task. C functions are easy to use and very efficient also.

Definition

A function can be defined as a subprogram which is meant for doing a specific task.

In a C program, a function definition will have name, parentheses pair contain zero or more parameters and a body. The parameters used in the parenthesis need to be declared with type and if not declared, they will be considered as of integer type.

The general form of the function is :

```
function type name <arg1,arg2,arg3, _____,argn> data type arg1, arg2,;  
data type argn;  
{  
body of function;  
_____  
_____  
_____  
return (<something>);  
}
```

From the above form the main components of function are

- Return type
- Function name
- Function body
- Return statement

Return Type

Refers to the type of value it would return to the calling portion of the program. It can have any of the basic data types such as int, float, char, etc. When a function is not supposed to return any value, it may be declared as type void

Example

```
void function name(-----);  
int function name(-----);  
char function name (-----);
```

Function Name

The function name can be any name conforming to the syntax rules of the variable.

A function name is relevant to the function operation.

Example

```
output(); read data();
```

Formal arguments

The arguments are called **formal arguments (or) formal parameters**, because they represent the names of data items that are transferred into the function from the calling portion of the program.

Any variable declared in the body of a function is said to be local to that function, other variable which were not declared either arguments or in the function body, are considered **“global”** to the function and must be defined externally.

Example

```
int biggest (int a, int b)
```

```
{
```

```
_____  
_____  
_____
```

```
return();
```

```
}
```

a, b are the formal arguments.

Function Body

Function body is a compound statement defines the action to be taken by the function. It should include one or more **“return”** statement in order to return a value to the calling portion of the program.

Example

```
int biggest(int a, int b)
```

```
{
```

```
if ( a > b)
```

```
return(a); body of function. else
```

```
return(b); }
```


Every C program consists of one or more functions. One of these functions must be called as **main**. Execution of the program will always begin by carrying out the instructions in main. Additional functions will be subordinate to main. If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition can't be embedded within another.

Generally a function will process information that is passed to it from the calling portion of the program and return a single value. Information is passed to the function via arguments (**parameters**) and returned via the “**return**” statement.

Some functions accept information but do not return anything (**ex:** printf()) whereas other functions (**ex:** scanf()) return multiple values.

The Return Statement

Every function subprogram in C will have return statement. This statement is used in function subprograms to return a value to the calling program/function. This statement can appear anywhere within a function body and we may have more than one return statement inside a function.

The general format of return statement is return;

(or)

return (expression);

If no value is returned from function to the calling program, then there is no need of return statement to be present inside the function.

Programs using function Call Techniques

Example 1: Write a program to find factorial to the given positive integer

,using function technique. # include <stdio.h> main()

```
{
```

```
int n;
```

```
printf ( “ Enter any positive number\n”); scanf( “%d”, &n);
```

```
printf( “ The factorial of %d s %d \n”,fact (n)); }
```

```
fact(i) int l;
```

```

{ int j; f = 1 ;
for ( j = 1; j>0; j - -) f = f * j;
return ( f ) ; }

```

In the above program function with name 'fact' is called by the main program. The function fact is called with n as parameter. The value is returned through variable f to the main program.

Example 2: Write a program to find the value of f(x) as $f(x) = x^2 + 4$, for the given of x. Make use of function technique.

```

#include <stdio.h> main( )
{ f ( ) ; }
f ( )
{ int x,y;
printf( " Enter value of x \n"); scanf( " %d", & x );
y = (x * x + 4);

printf ( " The value of f (x) id %d \n", y ) ;}

```

3.3. Differences between Function and Procedures

Procedure	Function
1. Procedure is a sub program which is included with in main program.	1. Functions is sub program which is intended for specific task. Eg. sqrt()
2. Procedure donot return a value.	2. Functions may or may not return a value.
3. Procedure cannot be called again and again.	3. Function once defined can be called any where n number of times.
4. Global variables cannot be used in procedure.	4. In functions both local and global variables can be used.
5. Procedures can be writ- ten only in procedural pro- gramming such as Dbase, Foxpro.	5. Functions can be written in modular programming such as C, C++

3.4 Advantages of Functions

The main advantages of using a function are:

- Easy to write a correct small function
- Easy to read and debug a function.
- Easier to maintain or modify such a function
- Small functions tend to be self documenting and highly readable
- It can be called any number of times in any place with different parameters.

Storage class

A variable's storage class explains where the variable will be stored, its initial value and life of the variable.

Iteration

The block of statements is executed repeatedly using loops is called Iteration

Categories of Functions

A function, depending on, whether arguments are present or not and a value is returned or not.

A function may be belonging to one of the following types.

1. Function with no arguments and no return values.
2. Function with arguments and no return values.
3. Function with arguments and return values

3.5 Advanced Featured of Functions

- a. Function Prototypes
- b. Calling functions by value or by reference
- c. Recursion.

a. Function Prototypes

The user defined functions may be classified as three ways based on the formal arguments passed and the usage of the **return** statement.

Functions with no arguments and no return value

- a. Functions with arguments no return value
- b. Functions with arguments and return value.

b. Functions with no arguments and no return value

A function is invoked without passing any formal arguments from the calling portion of a program and also the function does not return back any value to the called function. There is no communication between the calling portion of a program and a called function block.

Example: #include <stdio.h> main()

```

{
void message( ); Function declaration
message( ); Function calling
}
void message( )
{
printf ("GOVT JUNIOR COLLEGE \n");
printf ("\t HYDERABAD");
}

```

c. Function with arguments and no return value

This type of functions passes some formal arguments to a function but the function does not return back any value to the caller. It is any one way data communication between a calling portion of the program and the function block.

Example

```

#include <stdio.h> main()
{
void square(int);
printf ("Enter a value for n \n"); scanf ("%d",&n);
square(n);
}
void square (int n)
{

```

```
int value; value = n * n;
printf ("square of %d is %d ",n,value);
}
```

d. Function with arguments and return value

The third type of function passes some formal arguments to a function from a calling portion of the program and the computer

value is transferred back to the caller. Data are communicated between the calling portion and the function block.

Example

```
#include <stdio.h> main()
{
int square (int); int value;
printf ("enter a value for n \n"); scanf("%d", &n);
value = square(n);
printf ("square of %d is %d ",n, value);
}
int square(int n)
{
int p;
p = n * n; return(p);
}
```

The keyword **VOID** can be used as a type specifier when defining a function that does not return anything or when the function definition does not include any arguments.

The presence of this keyword is not mandatory but it is good programming practice to make use of this feature.

Actual and Formal Parameters (or) Arguments

Function parameters are the means of communication between the calling and the called functions. The parameters may classify under two groups.

- a. Formal Parameters
- b. Actual Parameters

1. Formal Parameters

The formal parameters are the parameters given in function declaration and function definition. When the function is invoked, the formal parameters are replaced by the actual parameters.

2. Actual Parameters

The parameters appearing in the function call are referred to as actual parameters. The actual arguments may be expressed as constants, single variables or more complex expression. Each

actual parameter must be of the same data type as its corresponding formal parameters.

Example

```
#include <stdio.h> int sum (int a , int b )
{
}
main()
{
}
int c;
c = a + b; return(c);
int x,y,z;
printf ("enter value for x,y\n"); scanf ("%d %d" ,&x,&y);
z = x + y;
printf (" sum is = %d",z);
```

The variables **a and b** defined in function definition are known as **formal parameters**. The variables **x and y** are **actual parameters**.

Local and Global Variable:

The variables may be classified as local or global variables.

Local Variable

The variables defined can be accessed only within the block in which they are declared. These variables are called “Local” variables

Example

```
func (int ,int j)
{
intk,m;
_____ ;
_____ ;
}
```

The integer variables **k** and **m** are defined within a function block of the “**func()**”. All the variables to be used within a function block must be either defined at the beginning of the

block or before using in the statement. Local variables are referred only to the particular part of a block of a function.

Global Variable

Global variables defined outside the main function block. Global variables are not contained to a single function. Global variables that are recognized in two or more functions. Their scope extends from the point of definition through the remainder of the program.

e. Calling functions by value or by reference

The arguments are sent to the functions and their values are copied in the corresponding function. This is a sort of information interchange between the calling function and called function. This is known as Parameter passing. It is a mechanism through which arguments are passed to the called function for the required processing. There are two methods of parameter passing.

1. Call by Value
2. Call by reference.

1. Call by value: When the values of arguments are passed from calling function to a called function, these values are copied in to the called function. If any changes are made to these values in the called function, there are NOCHANGE the original values within the calling function.

Example

```
#include <stdio.h> main();
{
int n1,n2,x;
int cal_by_val(); N1 = 6;
N2 = 9;
printf( n1 = %d and n2= %d\n", n1,n2); X = cal_by_Val(n1,n2);
Printf( n1 = %d and n2= %d\n", n1,n2); Printf(" x= %d\n", x);
/* end of main*/
/*function to illustrate call by value*/ Cal_by_val(p1,p2)
int p1,p2;
{
int sum;
Sum = (p1 + p2); P1 += 2;
P2* = p1;
printf( p1 = %d and p2= %d\n", p1,p2); return( sum);
}
}
```

When the program is executed the output will be displayed N1 = 6 and n2 = 9
P1 = 8 and p2 = 72 N1 = 6 and n2 = 9 X = 15

There is NO CHANGE in the values of n1 and n2 before and after the function is executed.

2. Cal by Reference: In this method, the actual values are not passed, instead their addresses are passed. There is no copying of values since their memory locations are referenced. If any modification is made to the values in the called function, then the original values get changed with in the calling function. Passing of addresses requires the knowledge of pointers.

Example

This program accepts a one-dimensional array of integers and sorts them in ascending order. [This program involves passing the array to the function].

```
#include <stdio.h> main();
{
int num[20], l,max; void sort_nums();
printf( " enter the size of the array\n"); scanf("%d", &max);
for( i=0; i<max;l++)
sort_nums(num,max)          /* Function reference*/ printf("sorted numbers are as
follows\n");
for( i=0; i<max;l++) printf("%3d\n",num[i]);
/* end of the main*/
/* function to sort list of numbers*/
Void sort_nums(a,n) Int a[],n;
{
Int l,j,dummy; For(i=0;i<n;i++)
{
For(j=0; j<n; j++)
{
If ( a[i] >a[j])
{
Dummy= a[i]; a[i] = a[j];

a[j] = dummy;
```

```
}  
}  
}  
}
```

3.6 Recursion

One of the special features of C language is its support to recursion. Very few computer languages will support this feature.

Recursion can be defines as the process of a function by which it can call itself. The function which calls itself again and again either directly or indirectly is known as recursive function.

The normal function is usually called by the main () function, by mans of its name. But, the recursive function will be called by itself depending on the condition satisfaction.

For Example, main ()

```
{
```

```
f1( ) ;
```

———— Function called by main

```
_____
```

```
_____
```

```
_____
```

```
}
```

```
f1( ) ;
```

———— Function definition

```
{
```

```
_____
```

```
_____
```

```
_____
```

```
f1();  
}
```

———— Function called by itself

In the above, the main () function is calling a function named f1 () by invoking it with its name. But, inside the function definition f1 (), there is another invoking of function and it is the function f1 () again.

Example programs on Recursion

Example 1 : Write a program to find the factorial of given non-negative integer using recursive function.

```
#include<stdio.h> main ( )  
{  
  
int result, n;  
printf( " Enter any non-negative integer\n"); scanf ( " %d", & n);  
result = fact(n);  
printf ( " The factorial of %d is %d \n", n, result);  
}  
  
fact( n ) int n;  
{  
  
int i; i = 1;  
if ( i == 1) return ( i); else  
{  
  
i = i * fact ( n - 1); return ( i);  
}  
}
```

Summary:

Define a Function: A function in C is a self-contained block of code designed to perform a specific task. Functions allow you to break down complex problems into smaller, manageable pieces, making your code more modular and reusable. A function is defined with a return type, a name, and a list of parameters (if any), followed by a block of code enclosed in curly braces {} that performs the task.

Stress on Return Statement: The return statement is crucial in a function as it specifies the value that the function should return to its caller. This value must match the function's return type. If a function is defined to return a value (e.g., int), it must include a return statement with the appropriate type. For functions with a void return type, no return value is required, though a return; statement can be used to exit the function early.

Write Programs Using Function Call Techniques: To use a function, it must be called from another function, typically main() or another function. Function calls involve specifying the function name and passing any required arguments. Functions can be called from other functions, allowing for nested and complex function interactions. Techniques for calling functions include:

- **Direct Call:** Using the function's name and passing arguments, e.g., myFunction(arg1, arg2);.
- **Function Pointer:** Storing the address of a function in a pointer and calling it through the pointer.

Function Prototype: A function prototype is a declaration of a function that specifies its return type, name, and parameters. It provides the compiler with information about the function's signature before its actual definition is encountered. Prototypes are typically placed at the beginning of a file or in a header file, allowing functions to be used before their definitions.

Differentiate Local and Global Variables:

- **Local Variables:** These are declared inside a function or block and are only accessible within that function or block. They are created when the function is called and destroyed when the function exits.

- **Global Variables:** These are declared outside any function and are accessible from any function within the same file or across multiple files (if declared with `extern`). They persist for the duration of the program's execution.

Recursion: Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. A recursive function must have a base case to terminate the recursion and prevent infinite loops. Recursive solutions can simplify code for problems that have a natural recursive structure, such as factorial calculation, Fibonacci series, or tree traversal.

Glossary :

Function: A self-contained block of code designed to perform a specific task. It is defined with a return type, a name, and a list of parameters (if any), followed by a block of code that executes the task.

Return Statement: A statement used in a function to specify the value that the function should return to its caller. The type of the return value must match the function's return type. For functions with a void return type, no value is returned, though a `return;` statement can be used to exit the function early.

Function Call Techniques:

- **Direct Call:** Calling a function by its name and passing the required arguments, e.g., `myFunction(arg1, arg2);`.
- **Function Pointer:** A variable that stores the address of a function, allowing the function to be called indirectly through the pointer.

Function Prototype: A declaration of a function that provides the compiler with information about the function's return type, name, and parameters before its actual definition. Prototypes are usually placed at the beginning of a file or in a header file.

Local Variables: Variables declared within a function or block that are only accessible within that function or block. They are created when the function is called and destroyed when the function exits.

Global Variables: Variables declared outside any function that are accessible from any function within the same file or across multiple files (if declared with extern). They persist for the duration of the program's execution.

Recursion: A programming technique where a function calls itself to solve smaller instances of the same problem. Recursion must have a base case to terminate the recursion and avoid infinite loops. Recursive functions are useful for problems that can be broken down into similar sub-problems, such as factorials or tree traversal.

Self assessment questions :

1. What is the purpose of a function in C programming?
2. How do you define a function that takes two integers as input parameters and returns their sum? Provide the function definition and a sample call.
3. Explain the role of the function name in function calls.

Stress on Return Statement

4. Why is the return statement important in a function?
5. What happens if a function that is supposed to return a value does not include a return statement?
6. Write a function that returns the maximum of two floating-point numbers. Ensure you include the appropriate return statement.

Write Programs Using Function Call Techniques

7. Describe how you would call a function that calculates the area of a circle from within another function. Include the function prototype and call.
8. What is a function pointer, and how can it be used to call a function? Provide an example of defining and using a function pointer.
9. Write a program that includes a function for calculating the factorial of a number. Show how the function is defined and called.

Function Prototype

10. What is a function prototype, and why is it necessary?
11. Given the function `int multiply(int a, int b);`, what would be its prototype if defined in a header file?

12. How does the function prototype help in organizing and structuring code in C?

Differentiate Local and Global Variables

13. Define local and global variables and explain the key differences between them.

14. Write a program that demonstrates the use of both local and global variables.

Explain the output and behavior of the program.

15. What potential issues could arise from using global variables extensively in a program?

Recursion

16. What is recursion, and when might it be preferable to use recursion over iteration?

17. Write a recursive function to compute the nth Fibonacci number. Describe the base case and recursive case in your function.

UNIT IV

UNIT 4 - Arrays

Arrays : Introduction - Definition of Array - Types of Arrays - Two - Dimensional Array - Declare, initialize array of char type - String handling functions in C - File operations

TABLE OF CONTENTS		
UNIT	TOPICS	PAGE
4	4.1.Arrays : Introduction	107
	4.2.Definition of Array	107
	4.2.1.Initialization of Array	
	Types of Arrays	
	Two - Dimensional Array	
	Declare, initialize array of char type	
	String handling functions in C	
	File operations	

UNIT OBJECTIVES:

- To understand the importance of Array
- Definition of array
- Declaration and Initializing of an Array
- Types of Arrays
- Examples of an Array
- String handling functions in c
- File operations

4.1 Arrays : Introduction

If we deal with similar type of variables in more number at a time, we may have to write lengthy programs along with long list of variables. There should be more number of assignment statements in order to manipulate on variables. When the number of variables increases, the length of program also increases.

In the above situations described above, where more number of same types of variables is used, the concept of ARRAYS is employed in C language. These are very much helpful to store as well as retrieve the data of similar type.

An Array describes a contiguously allocated non-empty set of objects with the same data type. The using arrays many number of same type of variables can be grouped together. All the elements stored in the array will referred by a common name

4.2 Definition of Array

Array can be defined as a collection of data objects which are stored in consecutive memory locations with a common variable name.

OR

Array can be defined as a group of values referred by the same variable name.

OR

An Array can be defined as a collection of data objects which are stored in consecutive memory locations with a common variable name.

The individual values present in the array are called elements of array. The array elements can be values or variables also.

Types of Arrays

Basically arrays can divide in to

1. One Dimensional Array

An array with only one subscript is called as *one-dimensional array or 1- d array*. It is used to store a list of values, all of which share a common name and are separable by subscript values

2. Two Dimensional Array

An array with two subscripts is termed as two-dimensional array.

A two-dimensional array, it has a list of given variable -name using two subscripts. We know that a one-dimensional array can store a row of elements, so, a two-dimensional array enables us to store multiple rows of elements.

4.2.1 Initialization of Array

Array can be made initialized at the time of declaration itself. The general form of array initialization is as below

```
type name[n] = [ element1, element2, .... element n];
```

The elements 1, element2... element n are the values of the elements in the array referenced by the same.

Example1:- `int codes[5] = [12,13,14,15,16];` Example2:- `float a[3] = [1.2, 1.3, 1.4];`

Example3:- `char name [5] = ['S', 'U', 'N', 'I', 'L'];`

In above examples, let us consider one, it a character array with 5 elements and all the five elements area initialized to 5 different consecutive memory locations.

```
name[0] = 'S'
```

```
name[1] = 'U'
```

```
name[2] = 'N'
```

```
name[3] = 'l'
```

```
name[4] = 'L'
```

Rules for Array Initialization

1. Arrays are initialized with constants only.
2. Arrays can be initialized without specifying the number of elements in square brackets and this number automatically obtained by the compiler.
3. The middle elements of an array cannot be initialized. If we want to initialize any middle element then the initialization of previous elements is compulsory.
4. If the array elements are not assigned explicitly, initial values will be set to zero automatically.
5. If all the elements in the array are to be initialized with one and same value, then repetition of data is needed.

4.3 Types of Array

The array must be declared as other variables, before its usage in a C program.

The array declaration included providing of the following information to C compiler.

- The type of the array (ex. int, float or char type)
- The name of the array (ex A[], B[], etc)
- Number of subscripts in the array (i.e whether one – dimensional or Two-dimensional)
- Total number of memory locations to be allocated.

The name of the array can be kept by the user (the rule similar to naming to variable).

There is no limit one on dimensions as well as number of memory locations and it depends on the capacity of computer main memory..

The general form for array declaration is Type name[n] ; { one dimensional array} Ex :

```
int marks[20];
```

```
char name[15]; float values [ 10];
```

An array with only one subscript is called as *one-dimensional array* or *1-d array*. It is used to store a list of values, all of which share a common name and are separable by subscript values.

Declaration of One-dimensional Arrays:

The general form of declaring a one-dimensional array is

data-type	array-name [size];
-----------	--------------------

Where data-type refers to any data type supported by C, array-name should be a valid C identifier; the size indicates the maximum number of storage locations (elements) that can be stored.

Each element in the array is referenced by the array name followed by a pair of square brackets enclosing a subscript value. The subscript value is indexed from 0 to size -1. When the subscript value is 0, first element in the array is selected, when the subscript value is 1, second element is selected and so on.

Example

```
int x [6];
```

1	2	3	4	5	6
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Here, x is declared to be an array of int type and of size six. Six contiguous memory locations get allocated as shown below to store six integer values.

Each data item in the array x is identified by the array name x followed by a pair of square brackets enclosing a subscript value. The subscript value is indexed from 0 to 5. i.e., x[0] denotes first data item, x[1] denotes second data item and x[5] denotes the last data item.

Initialization of One-Dimensional Arrays:

Just as we initialize ordinary variables, we can initialize one-dimensional arrays also, i.e., locations of the arrays can be given values while they are declared.

The general form of initializing an array of one-dimension is as follows: data - type array -

name [size] = {list of values};

The values in the list are separated by commas.

Example

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

as a result of this, memory locations of x get filled up as follows:

1	2	3	4	5	6
x	x[x[x[x	x
[1]	2	3]	[[
0]		4	5
]]]

Points to be considered during the declaration

1. If the number of values in initialization value - is less then the size of an array, only those many first locations of the array are assigned the values. The remaining locations are assigned zero.

Example: int x [6] = {7, 8, 6};

The size of the array x is six, initialization value - consists of only three locations get 0 assigned to them automatically, as follows:

7	8	6	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

2. If the number of values listed within initialization value - list for any array is greater than the six the array, compiler raises an error.

Example:

int x [6] = {1, 2, 3, 4, 5, 6, 7, 8};

The size of the array x is six. But the number of values listed within the initialization - list is eight. This is illegal.

3. If a static array is declared without initialization value - list then the all locations are set to zero.

Example:

```
static int x[6];
```

0	0	0	0	0	0
X(0)	X(1)	X(2)	X(3)	X(4)	X(5)

4. If size is omitted in a 1-d array declaration, which is initialized, the compiler will supply this value by examining the number of values in the initialization value - list.

Example:

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

0	0	0	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Since the number of values in the initialization value-list for the array x is six, the size of x is automatically supplied as six.

5. There is no array bound checking mechanism built into C-compiler. It is the responsibility of the programmer to see to it that the subscript value does not go beyond size-1. If it does, the system may crash.

Example:

```
int x [6];
```

```
x [7] = 20;
```

Here, x [7] does not belong to the array x, it may belong to some other program (for example, operating system) writing into the location may lead to unpredictable results or even to system crash.

6. Array elements can not be initialized selectively.

Example:

An attempt to initialize only 2nd location is illegal, i.e., `int x [6] = { , 10 }` is illegal.

Similar to arrays of int type, we can even declare arrays of other data types supported by C, also.

Example

```
char ch [6];
```

Ch	Ch	Ch	Ch	Ch	Ch
[0]	[1]	[2]	[3]	[4]	[5]

ch is declared to be an array of char type and size 6 and it can accommodate 6 characters.

Example:

```
float x [6];
```

x is declared to be an array of float type and size 6 and it can accommodate 6 values of float type. Following is the scheme of memory allocation for the array x:

x[1]	x[2]	x[3]	x[4]	x[5]
------	------	------	------	------

Note

A one array is used to store a group of values. A loop (using, for loop) is used to access each value in the group.

Example

```
Program to illustrate declaration, initialization of a 1-d array #include<stdio.h>
```

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

```
{
```

```
int i, x [6] = {1, 2, 3, 4, 5, 6 };
```

```
clrscr( );
```

```
printf("The elements of array x \n"); for(i=0; i<6; i++)
```

```
printf("%d", x[i]); getch( );
```

}Input – Output:

The elements of array x

1 2 3 4 5 6

4.4 Two - Dimensional Array

An array with two subscripts is termed as two-dimensional array.

A two-dimensional array, it has a list of given variable -name using two subscripts. We know that a one-dimensional array can store a row of elements, so, a two-dimensional array enables us to store multiple rows of elements.

Example: Table of elements or a Matrix.

Syntax of two-dimensional arrays:

The syntax of declaring a two-dimensional array is:

data - type array - name [rowsize] [colsize];
--

Where, data-type refers to any valid C data type, array -name refers to any valid C identifier, row size indicates the number of rows and column size indicates the number of elements in each column.

Row size and column size should be integer constants.

Total number of location allocated = (row size * column size).

Each element in a 2-d array is identified by the array name followed by a pair of square brackets enclosing its row-number, followed by a pair of square brackets enclosing its column-number.

Row-number range from 0 to row size-1 and column-number range from 0 to column size-1.

Example: `int y [3] [3];`

y is declared to be an array of two dimensions and of data type Integer (int), row size and column size of y are 3 and 3, respectively. Memory gets allocated to store the array y as follows. It is important to note that y is the common name shared by all the elements of the array.

0	1	2
x[0] [0]	x[0] [1]	x[0] [2]

Column numbers

x[1] [0]	x[1] [1]	x[1] [2]
x[2] [0]	x[2] [1]	x[2] [2]

Each data item in the array y is identifiable by specifying the array name y followed by a pair of square brackets enclosing row number, followed by a pair of square brackets enclosing column number.

Row-number ranges from 0 to 2. that is, first row is identified by row- number 0, second row is identified by row-number 1 and so on.

Similarly, column-number ranges from 0 to 2. First column is identified by column-number 0, second column is identified by column-number 1 and so on.

x [0] [0] refers to data item in the first row and first column x [0] [2] refers to data item in the first row and third column

x [2] [3] refers to data item in the third row and fourth column x [3] [4] refers to data item in the fourth row and fifth column

Initialization of Two-Dimensional Array

There are two forms of initializing a 2-d array. First form of initializing a 2-d array is as follows:

```
data - type array - name [rowsize][colsize] = [initializer - list];
```

Where, data-name refers to any data type supported by C. Array-name refers to any valid C identifier. Row size indicates the number of rows, column size indicates the number of columns of the array. initialier-list is a comma separated list of values.

If the number of values in initializer-list is equal to the product of row size and column size, the first row size values in the initializer-list will be assigned to the first row, the second row size values will be assigned to the second row of the array and so on

Example: int x [2] [4] = {1, 2, 3, 4, 5, 6, 7, 8 };

Since column size is 4, the first 4 values of the initializer-list are assigned to the first row of x and the next 4 values are assigned to the second row of x as shown hereinafter

3
4
7
8

Note: If the number of values in the initializer-list is less than the product of row size and col size, only the first few matching locations

of the array would get values from the initializer-list row-wise. The trailing unmatched locations would get zeros.

Example: `int x [2] [4] = { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9 } };`

col size is 4, but the number of values listed in the first row is 5. This results in compilation error.

1. Array elements can not be initialized selectively.
2. It is the responsibility of the programmer to ensure that the array bounds do not exceed the row size and col size of the array. If they exceed, unpredictable results may be produced and even the program can result in system crash sometimes.
3. A 2-d array is used to store a table of values (matrix).

Similar to 2-d arrays of int type, we can declare 2-arrays of any other data type supported by C, also.

Example: `float x [4] [5];`

x is declared to be 2-d array of float type with 4 rows and 5 columns double y [4] [5];

y is declared to be 2-d arrays of double type with 4 rows and 5 columns

Note

A two-dimensional array is used to store a table of values. Two loops (using for loops) are used to access each value in the table, first loop acts as a row selector and second loop acts as a column selector in the table.

4.5 Declare, Initialize Array of Char Type

Declaration of Array of char type: A string variable is any valid C variable name and is always declared as an array. The syntax of declaration of a string variable is:

```
char string-name[size];
```

The size determines the number of character in the string name.

Example: An array of char type to store the above string is to be declared as follows:

```
char str[8];
```

```
str[0] str[1] str[2] str[3]          str[4] str[5] str[6] str[7]
```

P	r	o	g	r	a	m	\0
---	---	---	---	---	---	---	----

An array of char is also called as a string variable, since it can store a string and it permits us to change its contents. In contrast, a sequence of characters enclosed within a pair of double quotes is called a string constant.

Example: “Program” is a string constant.

Initialization of Arrays of char Type

The syntax of initializing a string variable has two variations:

Variation 1

```
char str1 [6] = { 'H', 'e', 'l', 'l', 'o', '\0'};
```

Here, str1 is declared to be a string variable with size six. It can store maximum six characters. The initializer – list consists of comma separated character constants. Note that the null character '\0' is clearly listed. This is required in this variation.

Variation 2

```
char str2 [6] = { "Hello" };
```

Here, str2 is also declared to be a string variable of size six. It can store maximum six characters including null character. The initializer-list consists of a string constant. In this variation, null character '\0' will be automatically added to the end of string by the compiler.

In either of these variations, the size of the character array can be skipped, in which case, the size and the number of characters in the initializer-list would be automatically supplied by the compiler.

Example

```
char str1 [] = { "Hello" };
```

The size of str1 would be six, five characters plus one for the null character

'\0'.

'\0'.

```
char str2 [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The size of str2 would be six, five characters plus one for null character

Example 1

Program to sort a list of numbers.

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

void main()
{
int x [6], n, i, j, tmp; clrscr( );
printf ("Enter the no. of elements\n"); scanf ("%d", & n);
printf ("Enter %d numbers\n", n); for (i=0; i<n; i++)
scanf("%d", &x [i]);

/* sorting begins */ for (i=0; i<n; i++)
for (j=i + 1; j<n; j++) if (x[i]>x[j])

{
tmp = x [i]; x [i] = x [j]; x [j] = tmp;
}
/* sorting ends */ printf ("sorted list\n"); for (i=0; i<n; i++) printf ("%d", x [i]); getch( );
}
```

Input–Output

Enter the no. of elements 5

Enter 5 numbers

10 30	20	50	40
Sorted list			
10 20 30	40	50	

Example 2 :

C program for addition of two matrices.

```
#include<stdio.h> #include<conio.h> #include<process.h> void main( )
{
int x [5] [5], y [5] [5], z [5] [5], m, n, p, q, i, j;
clrscr ( );
printf (“Enter number of rows and columns of matrix x\n”); scanf (“%d %d”, &m, &n);
printf (“Enter number of rows and columns of matrix y\n”); scanf (“%d %d”, &p, &q);
if ((m !=p) || (n!=q))
{
printf (“Matrices not compatible for addition\n”); exit(1);
}
printf (“Enter the elements of x\n”); for (i=0; i<m; i++)
for (j=0; j<n; j++) scanf (“%d”, &x [i][j]);

printf (“Enter the elements of x\n”); for (i=0; i<p; i++)
for (j=0; j<n; j++) scanf (“%d”, &y [i] [j]);
/* Summation begins */ for (i=0; j<m; i++)
for (j=0; j<n; j++)
z[i] [j] = x [i] [j] + y [i] [j];
```

```

/* summation ends */ printf("Sum matrix z\n"); for (i=0; i<m; i++)
{
for (j=0; j<n; j++)
printf("%d", z[i][j]); printf("\n");
}
getch( );
}

```

Example3

Write a program for multiplication of two matrices.

```

#include<stdio.h> #include<conio.h> #include<process.h> void main( )
{
int x [5] [5], y [5] [5], z [5] [5], m, n, p, q, l, j, k;
clrscr ( );
printf ("Enter number of rows and columns of matrix x\n"); scanf ("%d %d", &m, &n);
printf ("Enter number of rows and columns of matrix y\n"); scanf ("%d %d", &p, &q);
if (n!=p)
{
printf ("Matrices not compatible for multiplication\n"); exit (1);
}
printf ("Enter the elements of x\n"); for (i=0; i<m; i++)
for (j=0; j<n; j++) scanf ("%d", &x [i] [j]);
printf ("Enter the elements of y\n"); for (i=0; i<p; i++)
for (j=0; j<q; j++) scanf ("%d", &x [i] [j]);
printf ("Enter the elements of y\n"); for (i=0; i<p; i++)
for (j=0; j<q; j++) scanf ("%d", &y [i] [j]);
/* Multiplication of matrices of x 7 y ends */ for (i=0; i<m; i++)

for (j=0; j<q; j++)

```

```

{ z [i] [j] = 0;
for (k=0; k<n; k++)
z [i] [j] += x [i] [k] * y [k] [j];
}
/* Multiplication of matrices of x & y ends */ printf ("Product Matrix z \n");
for (i=0; i<m; i++)
{
for (j=0; j<q; j++)
    printf ("%d", z[i] [j]); printf ("\n");
}
getch( );
}

```

Example 4: C program to print transpose of a matrix.

[The transpose of a matrix is obtained by switching the rows and columns of matrix].

```

#include<stdio.h> #include<conio.h> void main ( )
{
int a[3] [3], b[3] [3], i, j; clrscr ( );
printf ("Enter the elements of the matrix : \n"); for (i=0; i<3; i++)
for (j=0; j<3; j++) scanf ("%d", &a[i] [j]);
printf ("given matrix is : \n"); for (i=0; i<3; i++)
{
for (j=0; j<3; j++)
printf ("%d", a[i] [j]); printf ("\n");
}
printf ("transpose of given matrix is : \n"); for (i=0; i<3; i++)
{
for (j=0; j<3; j++)
{

```

```

b [i] [j] = a [j] [i];
printf ("%d", b[i] [j]);
printf ("\n");
}
}
getch( );
}

```

Example 5

Write a 'C' program to find the average marks of 'n' students of five subjects for each subject using arrays.

Ans.

```

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

```

```

Example 6 int Sno, S1,S2,S3; float tot, avg;
char sname[10]; printf("Enter student no,"); scanf("%d", & Sname);
printf("Enter subject - 1, sub - 2, sub - 3 marks;"); scanf("%d %d %d", &s1,&s2,&s3);
tot = S1+S2+S3;
avg = tot/3;
printf("total = %f", tot); printf("Average = %f", avg);

```

Write a C program to check the given word is 'Palindrome' or not. Ans.

```

#include<stdio.h>
#include<conio.h> void main ( )
{
char str[80], rev[80]; int k, i, j, flag = 0; clrscr ( );
printf ("Enter any string (max. 80 chars) : \n"); gets (str);
for (i=0; str[i]!='\0'; i++);

```

```

for (j=i-1; k=0; j>=0; j--, k++) rev[k] = str[j];
rev[k] = '\0';

```



```

for (i=0; str[i]!='\0'; i++)
{
if (str[i]!=rev[i])
{
flag=1; break;
}
}
Else
if (flag ==1);
printf ("Given string is not palindrome. \n");
printf ("Given string is palindrome. \n");
getch();
}

```

4.6 String Handling Functions in 'C'

To perform manipulations on string data, there are built-in-function (library function) Supported by 'c' compiler. The string functions are.

1. STRLEN (S1)
 2. STRCMP (S1, S2)
 3. STRCPY (S1, S2)
 4. STRCAT (S1, S2)
1. **STRLEN (S1):** This function is used to return the length of the string name S1,
Ex: S1 = 'MOID'
STRLEN (S1) = 4
 2. **STRCMP (S1, S2):** This is a library function used to perform comparison between two strings. This function returns a value <zero when string S1 is less than S2. The function return a value 0 when S1=S2. Finally the function return a value > 0 when S1>S2.
 3. **STRCPY (S1, S2):** This is a library function used to copy the string S2 to S1.
 4. **STRCAT (S1, S2):** This is a library function used to join two strings one after the other. This function concatenates string S2 at the end of string S1.

Example 5 : C program to concatenate the given two strings and print new string.

```
#include<stdio.h> #include<conio.h> main ( )
{
char s1[10], s2[10], s3[10],
int i,j,k;
printf ("Enter the first string : \n"); scanf("%s,S",s1);
printf ("Enter the second string : \n");
scanf("%s,S",s2);
i = strlen(s1);
j = strlen(s2);
for (k=0,k<i,k++) s3[k] = s1[k];
for (k=0,k<j,k++) s3[i+k] = s2[k];
s3[i+j] = \0';
printf(" The new string is \n".s3);
}
```

4.7 File Operations like fopen(), fclose(), fprintf(), fscanf()

The help of these functions the user can open a file with the data file specifications, create and write information in to the file and can close the file. The following are the file processing related functions.

- a. FILE OPEN function fopen()
 - b. FILE CLOSE function fclose()
 - c. FILE INPUT functions getc() and fscanf()
 - d. FILE OUTPUT functions putc() and fprintf()
- a. The function fopen()

This function is used to open a data file. Moreover, this function returns a pointer to a file. The use of the function is

```
file pointer = fopen(filename, mode);
```

Where file pointer is a pointer to a type FILE, the file name of the file name is name of the file in which data is stored or retrieved (should be enclosed within double quotes) and the mode denotes the type of operations to be performed on the file (this also to be enclosed within double quotes).

But, before making this assignment, the file pointer and fopen() should be declared as FILE pointer type variables as under :

```
FILE *file pointer, * fopen() ;
```

The mode can be one of the following types.

MODE	MEANING
“r“	read from the file
“w”	write to the file
“a”	append a file ie new data is added to the end of file
“r+”	open an existing file for the sake of updation.
“w +”	create a new file for reading and writing
“a +”	open a file for append, create a new one if the file does

not exist already

Examples

1. fptr = fopen(“rk. Dat”, “w”);
 2. file= fopen(“sample.dat”, “r +”);
- b.** The functions fclose ()

The files that are opened should be closed after all the desired operations are performed on it .This can be achieved through this function .The usage of this function is:

fclose (file pointer);

Where file pointer is returned value of the fopen () function. Examples:

1. fclose (input file); 2. fclose (output file);

c. The functions getc() & fscanf()

1. **getc () functions:** this function is used a single character from a given file ,whenever a file is referenced by a file pointer. The usage of this function is

getc (file pointer);

2. **fscanf ()function :** This function is used to read a formatted data from a specified file. The general usage of this function is

fscanf (f ptr, "Control String", & list); where

fptr → a file pointer to receive formatted data Control string → data specifications list

List → the list of variables to be read fscanf (infile , "%d %d ," & no, &marks);

d. The functions putc()& fprintf() Example

1. **putc () function:** This function is used write a single character into a file referenced by the file pointer. The usage of this function is

putc (ch, file pointer),

Where

ch - the character to be written

file pointer -a file pointer to the file that receives the character.

2. **fprintf () function:** this function is used t write a for matted data in to a given file. The specified information is written on the specified file.

The general form of usage for this function is: fprintf (fptr, "Control String", list):

Where

Fptr → file pointer to write a formatted data Control string → data specifications list

list → list of variables to be written.

Example

```
fprintf (out file, "%d %f", basic , gross);
```

Self Assessment Questions:

1. Definition of Array

1. What is an array in C, and how does it differ from a scalar variable?
2. Why are arrays useful in programming? Provide an example scenario where using an array is beneficial.

2. Declaration and Initializing of an Array

1. Write the code to declare and initialize an array of 10 integers with values from 1 to 10.
2. What will happen if you try to access an array element using an index that is out of bounds?

3. Types of Arrays

1. Describe the difference between a single-dimensional array and a two-dimensional array. Provide an example of each.
2. How would you declare a three-dimensional array in C? Give an example with dimensions 3x4x2.

4. Examples of an Array

1. Given the following declaration:

c

Copy code

```
int nums[4] = {10, 20, 30, 40};
```

Write a loop to print each element of the array.

2. How would you access and modify the value of the second element in a two-dimensional array declared as follows:

c

Copy code

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

5. String Handling Functions in C

1. Write a C function that takes two strings and concatenates them into a new string.
2. What is the purpose of the null character ('\0') in C strings, and how does it affect string manipulation functions?

6. File Operations

1. Write a C program snippet that opens a file named "data.txt" for writing, writes the string "Hello, File!" to it, and then closes the file.
2. How would you handle errors in file operations, such as if a file fails to open?

Answers for Self-Assessment Questions

1. Definition of Array

1. **Answer:** An array is a data structure that stores a fixed-size sequence of elements of the same type in contiguous memory locations. Unlike scalar variables that hold a single value, arrays allow you to store multiple values using indices.
2. **Answer:** Arrays are useful for handling collections of data efficiently. For example, if you need to store and process a list of student grades, using an array allows you to handle all grades using a single variable and access each grade using an index.

2. Declaration and Initializing of an Array

1. **Answer:**

c

Copy code

```
int numbers[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

2. **Answer:** Accessing an out-of-bounds index can lead to undefined behavior, which may include accessing garbage values, causing a crash, or corrupting data.

3. Types of Arrays

1. **Answer:**

- A single-dimensional array is a linear array. Example:

c

Copy code

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

- A two-dimensional array is a matrix or table. Example:

c

Copy code

```
int matrix[3][3] = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9}  
};
```

2. **Answer:**

c

Copy code

```
int array[3][4][2];
```

4. Examples of an Array

1. **Answer:**

c

Copy code

```
for (int i = 0; i < 4; i++) {  
    printf("%d\n", nums[i]);  
}
```

2. **Answer:**

c

Copy code

```
matrix[0][1] = 99; // Modifies the second element in the first row to 99
```

5. String Handling Functions in C

1. **Answer:**

c

Copy code

```
char* concatenate_strings(const char* str1, const char* str2) {  
    char* result = malloc(strlen(str1) + strlen(str2) + 1); // +1 for the null terminator  
    if (result == NULL) {  
        return NULL; // Memory allocation failed  
    }  
    strcpy(result, str1);  
    strcat(result, str2);  
    return result;  
}
```

2. **Answer:** The null character ('\0') signifies the end of a string in C. It allows functions like strlen, strcpy, and strcat to determine where the string ends.

6. File Operations

1. **Answer:**

c

Copy code

```
FILE *file = fopen("data.txt", "w");  
if (file != NULL) {  
    fprintf(file, "Hello, File!");  
    fclose(file);  
}
```

2. **Answer:** You can check for errors by verifying the return value of fopen. If fopen returns NULL, it indicates an error. Always ensure to check for errors and handle them appropriately, possibly by printing an error message or taking corrective action.

Activities and Exercises

1. Definition of Array

Activity:

- Create a program that prompts the user to enter the number of elements for an integer array. Allow the user to input values for each element and then print the array in reverse order.

Exercise:

- Write a function that takes an array of integers and returns the maximum and minimum values in the array.

2. Declaration and Initializing of an Array

Activity:

- Implement a program that initializes a 2D array representing a multiplication table (e.g., 10x10). Print the table in a formatted way.

Exercise:

- Write a C program that initializes a one-dimensional array with the first 20 Fibonacci numbers and prints them.

3. Types of Arrays

Activity:

- Create a program that uses a 3D array to store and display the scores of students across different subjects over multiple terms. Each dimension represents students, subjects, and terms respectively.

Exercise:

- Write a function that takes a 2D array representing a matrix and returns the transpose of the matrix.

4. Examples of an Array

Activity:

- Develop a program that sorts a given array of integers in ascending order using the bubble sort algorithm. Display the sorted array.

Exercise:

- Write a function to find the average value of all elements in a given 2D array and return it.

5. String Handling Functions in C**Activity:**

- Implement a program that reads a string from the user and then performs and displays the following operations: length of the string, converting the string to uppercase, and reversing the string.

Exercise:

- Create a function that counts and returns the number of occurrences of a specific character in a given string.

6. File Operations**Activity:**

- Write a program that reads data from a file named "input.txt", processes the data to calculate the sum of numbers found in the file, and writes the result to "output.txt".

Exercise:

- Develop a program that opens a file in read mode, reads its content line by line, and prints each line along with its line number.

Case Studies**Case Study 1: Student Grades Management System**

Scenario: You are tasked with developing a simple student grades management system. You need to use arrays to handle and process the data.

Tasks:

1. **Array Declaration and Initialization:** Use a 2D array to store grades for 5 students in 4 subjects.
2. **Data Input:** Write functions to input grades for each student and subject.
3. **Data Processing:** Calculate and display the average grade for each student and each subject.

4. **Display:** Implement functions to print out the grades in a tabular format.

Hints:

- Use nested loops to iterate through the 2D array.
- Ensure input validation for grades.

Case Study 2: Text File Analyzer

Scenario: You need to create a text file analyzer that performs various operations on text data stored in a file.

Tasks:

1. **File Reading:** Write a function to read the entire content of a file into a string.
2. **String Processing:** Use string handling functions to:
 - Count the number of words in the file.
 - Find and replace all occurrences of a specific word.
3. **File Writing:** Write the processed text into a new file.
4. **Error Handling:** Ensure that your program handles file I/O errors gracefully.

Hints:

- Use fopen, fread, fwrite, fclose, and fgets for file operations.
- Implement error checks after each file operation.

Case Study 3: Multi-Dimensional Array for Image Processing

Scenario: You are working on a basic image processing application where you use a 2D array to represent a grayscale image.

Tasks:

1. **Image Representation:** Create a 2D array to store pixel values of the image.
2. **Image Manipulation:** Implement functions to:
 - Apply a simple filter (e.g., increasing brightness by a fixed value).
 - Rotate the image 90 degrees clockwise.
3. **Display:** Write a function to print the 2D array in a grid format representing the image.

Hints:

- Use nested loops for manipulating the 2D array.
- Think about how to index and access pixel values efficiently.

Summary:

Importance of Array

Arrays are fundamental data structures in C that allow you to store and manage collections of data efficiently. They provide a way to handle multiple values of the same type under a single variable name, which simplifies code management and improves performance. Arrays are crucial for tasks that involve processing lists of items, such as sorting, searching, and handling data in algorithms.

Definition of Array

An array in C is a collection of elements of the same type stored in contiguous memory locations. Each element can be accessed using an index, with indexing typically starting from 0. Arrays enable efficient data storage and access patterns, making them a cornerstone of data manipulation in C.

Declaration and Initializing of an Array

- **Declaration:** To declare an array, you specify the type of its elements and the number of elements it will hold. For example, `int numbers[10];` declares an array of 10 integers.
- **Initialization:** Arrays can be initialized at the time of declaration using curly braces. For example, `int numbers[5] = {1, 2, 3, 4, 5};` initializes the array with specified values. If the array is not fully initialized, the remaining elements are set to zero.

Types of Arrays

- **Single-Dimensional Array:** A one-dimensional array is a simple list of elements. For example, `int arr[5];` creates a single-dimensional array of integers.

- **Multi-Dimensional Array:** Arrays with more than one dimension, such as two-dimensional arrays (matrices), can be declared. For example, `int matrix[3][4];` creates a 2D array with 3 rows and 4 columns.
- **Jagged Array:** An array of arrays where each sub-array can have different sizes. This is useful when dealing with non-uniform data structures.

Examples of an Array

- **Example 1:** A simple array of integers: `int scores[4] = {90, 85, 88, 92};`
- **Example 2:** A two-dimensional array representing a matrix:

```
c
Copy code
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

- **Example 3:** Accessing elements of an array: `int value = scores[2];` retrieves the third element from the scores array.

String Handling Functions in C

C provides a set of standard library functions for manipulating strings, which are arrays of characters terminated by a null character ('\0'). Common string functions include:

- **strlen():** Returns the length of a string.
- **strcpy():** Copies one string to another.
- **strcat():** Concatenates two strings.
- **strcmp():** Compares two strings.
- **strchr():** Finds the first occurrence of a character in a string.
- **strstr():** Finds the first occurrence of a substring in a string.

File Operations

File operations in C allow you to perform input and output operations on files. The key functions and concepts include:

- **Opening a File:** `fopen()` is used to open a file, specifying the file name and mode ("r" for read, "w" for write, "a" for append, etc.).
- **Reading from a File:** Functions such as `fgetc()`, `fgets()`, and `fread()` are used to read data from files.
- **Writing to a File:** Functions like `fputc()`, `fputs()`, and `fwrite()` are used to write data to files.
- **Closing a File:** `fclose()` is used to close a file and release the resources associated with it.
- **File Error Handling:** Functions like `ferror()` and `clearerr()` help manage and check for file errors.

Glossary :

Array: A data structure in C that stores a fixed-size sequence of elements of the same type in contiguous memory locations. Each element is accessed using an index, with indexing typically starting from 0.

Importance of Array: Arrays are essential for managing collections of data efficiently. They provide a way to store multiple values under a single variable name, simplifying code management and enhancing performance for data processing tasks.

Declaration of Array: The process of defining an array, specifying its type and the number of elements it will hold. For example, `int numbers[10];` declares an array of 10 integers.

Initializing of Array: Setting initial values for an array at the time of declaration. For instance, `int numbers[5] = {1, 2, 3, 4, 5};` initializes an integer array with specific values. Uninitialized elements are set to zero by default.

Single-Dimensional Array: A type of array with only one index, representing a list of elements. For example, `int arr[5];` creates a single-dimensional array of integers.

Multi-Dimensional Array: An array with more than one index, allowing for more complex data structures such as matrices. For example, `int matrix[3][4];` creates a two-dimensional array with 3 rows and 4 columns.

Jagged Array: An array of arrays where each sub-array can have different sizes. Useful for handling non-uniform data structures where sub-arrays need varying lengths.

String Handling Functions: Functions in C designed for manipulating strings, which are arrays of characters ending with a null character ('\0'). Common functions include:

- **strlen():** Returns the length of a string.
- **strcpy():** Copies one string to another.
- **strcat():** Concatenates two strings.
- **strcmp():** Compares two strings.
- **strchr():** Finds the first occurrence of a character in a string.
- **strstr():** Finds the first occurrence of a substring in a string.

File Operations: Functions and processes for handling files in C, allowing for input and output operations. Key functions include:

- **fopen():** Opens a file, specifying the file name and mode (e.g., "r" for read, "w" for write).
- **fgetc():** Reads a single character from a file.
- **fgets():** Reads a string from a file.
- **fread():** Reads a block of data from a file.
- **fputc():** Writes a single character to a file.
- **fputs():** Writes a string to a file.
- **fwrite():** Writes a block of data to a file.
- **fclose():** Closes an open file and releases associated resources.
- **ferror():** Checks for errors on a file stream.
- **clearerr():** Clears the error indicator for a file stream.

UNIT V

UNIT 5 - Structures

Structures: Introduction - Definition of Structure - Structure Declaration - Structures and Arrays - Structure contains Pointers – Unions - Definition of Union - Differences between Structure and Union

TABLE OF CONTENTS		
UNIT	TOPICS	PAGE
5	5.1.Introduction	137
	5.2.Definition of Structure	137
	5.3.Structure Declaration	138
	5.4.Structures and Arrays	142
	5.5.Structure contains Pointers	146
	5.6.Unions	148
	5.7.Definition of Union	148
	5.8.Differences between Structure and Union	149

UNIT OBJECTIVES:

- Importance of structures
- Definition of structure
- Implementation of arrays in structures
- Definition of Union
- Union declaration
- Differences between structures and Union

5.1 Introduction

Arrays are useful to refer separate variables which are the same type. i.e. Homogeneous referred by a single name. But, we may have situations where there is a need for us to refer to different types of data (Heterogeneous) in order to derive meaningful information.

Let us consider the details of an employee of an organization. His details include employer's number (Integer type), employee's name (Character type) , basic pay (Integer type) and total salary (Float data type) . All these details seem to be of different data types and if we group them together, they will result in giving useful information about employee of the organization.

In above said situations, C provides a special data types called Structure, which is highly helpful to organize different types of variables under a single name.

5.2 Definition of Structure

A group of one or more variables of different data types organized together under a single name is called

Structure.

Or

A collection of heterogeneous (dissimilar) types of data grouped together under a single name is called a **Structure.**

A structure can be defined to be a group of logically related data items, which may be of different types, stored in contiguous memory locations, sharing a common name, but distinguished by its members.

Hence a structure can be viewed as a heterogeneous user-defined data type. It can be used to create variables, which can be manipulated in the same way as variables of built-in data types. It helps better organization and management of data in a program.

When a structure is defined the entire group is referenced through the structure name. The individual components present in the structure are called structure members and those can be accessed and processed separately.

5.3 Structure Declaration

The declaration of a structure specifies the grouping of various data items into a single unit without assigning any resources to them. The syntax for declaring a structure in C is as follows:

```
struct Structure Name
{
    Data Type member-1; Data Type member-2;
    ....
    Data Type member-n;
};
```

The structure declaration starts with the structure header, which consists of the keyword '**struct**' followed by a tag. The tag serves as a structure name, which can be used for creating structure variables. The individual members of the structure are enclosed between the curly braces and they can be of the similar or dissimilar data types. The data type of each variable is specified in the individual member declarations.

Example:

Let us consider an employee database consisting of employee number, name, and salary. A structure declaration to hold this information is shown below:

```
struct employee
```

```
{  
int eno;  
char name [80]; float sal;  
};
```

The data items enclosed between curly braces in the above structure declaration are called structure elements or structure members.

Employee is the name of the structure and is called structure tag. Note that, some members of employee structure are integer type and some are character array type.

The individual members of a structure can be variables of built – in data types (int, char, float etc.), pointers, arrays, or even other structures. All member names within a particular structure must be different. However, member names may be the same as those of variables declared outside the structure. The individual members cannot be initialized inside the structure declaration.

Note

Normally, structure declarations appear at the beginning of the program file, before any variables or functions are declared.

They may also appear before the main (), along with macro definitions, such as #define.

In such cases, the declaration is global and can be used by other functions as well.

Structure Variables

Similar to other types of variables, the structure data type variables can be declared using structure definition.

```
struct  
{  
int          rollno; char name[20]; float average; a, b;  
}
```

In the above structure definition, a and b are said to be structure type

variables. 'a' is a structure type variable containing rollno, name average as members, which are of different data types. Similarly 'b' is also a structure type variable with the same members of 'a'.

Structure Initialization

The members of the structure can be initialized like other variables. This can be done at the time of declaration.

Example 1

```
struct
{
int      day;
int      month;
int      year;
}
date = { 25,06,2012};
```

i.e

date. day = 25 date. month = 06 date. year = 2012

Example 2

```
struct address
{
char      name [20];
char      desgn [10];
char      place [10];
};
```

i.e

struct address my-add = { 'Sree', 'AKM', 'RREDDY'}; i.e

my-add . name = 'Sree' my-add . desgn = AKM

my-add . place = RREDDY

As seen above, the initial values for structure members must be enclosed with in a pair of curly braces. The values to be assigned to members must be placed in the same order as they are specified in structure definition, separated by commas. If some of the members of the structure are not initialized, then the c compiler automatically assigns a value 'zero' to them.

Accessing of Structure Members

As seen earlier, the structure can be individually identified using the period operator (.). After identification, we can access them by means of assigning some values to them as well as obtaining the stored values in structure members. The following program illustrates the accessing of the structure members.

Example: Write a C program, using structure definition to accept the time and display it.

```
/* Program to accept time and display it */ # include <stdio.h>
main()
{
struct
{
int hour, min; float seconds;
} time;
printf ( "Enter time in Hours, min and Seconds\n");
scanf ( " %d %d %f", &time . hour, & time . min, & time . seconds); printf ( " The
accepted time is %d %d %f", time . hour, time . min, time
. seconds ");
}
```

Nested Structures

The structure is going to certain number of elements /members of different data types. If the members of a structure are of structure data type, it can be termed as structure with structure or nested structure.

Example

```
struct
```

```

{
int          rollno; char name[20]; float avgmarks; struct
{
int day, mon, year;
} dob'
} student;

```

In the above declaration, student is a variable of structure type consisting of the members namely rollno, name, avgmarks and the structure variable dob.

The dob structure is within another structure **student** and thus structure is nested. In this type of definitions, the elements of the require structure can be referenced by specifying appropriate qualifications to it, using the period operator (.) .

For example, **student.dob.day** refers to the element day of the inner structure dob.

5.4 Structures and Arrays

Array is group of identical stored in consecutive memory locations with a single / common variable name. This concept can be used in connection with the structure in the following ways.

- a. Array of structures
- b. structures containing arrays (or) arrays within a structure
- c. Arrays of structures contain arrays.

Array of Structures

Student details in a class can be stored using structure data type and the student details of entire class can be seen as an array of structure.

Example

```

struct student
{
int rollno; int year; int tmarks;
}

```

```
struct student class[40];
```

In the above class [40] is structure variable accommodating a structure type student up to 40.

```
The above type of array of structure can be initialized as under struct student class [2]
= { 001,2011,786},
{ 002, 2012, 710}
};
```

```
i.e class[0] . rollno = 001 class[0] . year = 2011
```

```
class[0] . tmarks = 777 and class[1] . rollno = 002
```

```
class[1] . year = 2012
```

```
class[1] . tmarks = 777 .
```

Structures containing Arrays

A structure data type can hold an array type variable as its member or members. We can declare the member of a structure as array data type similar to int, float or char.

Example

```
struct employee
{
char ename [20]; int eno;
};
```

In above, the structure variable employee contains character array type ename as its member. The initialization of this type can be done as usual.

```
struct employee = { 'Rajashekar', 7777};
```

Arrays of Structures Contain Arrays

Arrays of structures can be defined and in that type of structure variables of array type can be used as members.

Example

```
struct rk
```

```

{
int empno;
char ename[20]; float salary;
} mark[50];

```

In the above, mark is an array of 50 elements and such element in the array is of structure type rk. The structure type rk, in turn contains ename as array type which is a member of the structure. Thus mark is an array of structures and these structures in turn holds character names in array ename.

The initialization of the above type can be done as:

```

{
7777, ' Prasad' , 56800.00}
};

```

i.e mark[0] . empno = 7777; mark[0] . ename = 'Prasad'; mark[0] . salary = 56800.00

Program

Write a C program to accept the student name, rollno, average marks present in the class of student and to print the name of students whose average marks are greater than 40 by using structure concept with arrays.

```

#include <stdio.h> main()
{

int i, n, struct
{
char name [20];
int rollno;
float avgmarks;
}
class [40];

printf (" Enter the no. of students in the class\n"); scanf ( " %d", & n );
for ( i = 0, i < n, i++)

```



```

{
print ( " Enter students name, rollno, avgmarks\n");
scanf( " %s %d", &class[i].name, class[i].rollno, &class[i].avgmarks)
}
printf ( " The name of the students whose average"); printf ( " marks is greater than 40
\n");
for ( i = 0, i < n, i++)
if ( class[i].avgmarks > 40) printf ( " %s", class[i].name);
}

```

Advantages of Structure Type over Array Type Variables

1. Using structures, we can group items of different types within a single entity, which is not possible with arrays, as array stores similar elements.
2. The position of a particular structure type variable within a group is not needed in order to access it, whereas the position of an array member in the group is required, in order to refer to it.
3. In order to store the data about a particular entity such as a 'Book', using an array type, we need three arrays, one for storing the 'name', another for storing the 'price' and a third one for storing the 'number of pages' etc., hence, the overhead is high. This overhead can be reduced by using structure type variable.
4. Once a new structure has been defined, one or more variables can be declared to be of that type.
5. A structure type variable can be used as a normal variable for accepting the user's input, for displaying the output etc.,
6. The assignment of one 'struct' variable to another, reduces the burden of the programmer in filling the variable's fields again and again.
7. It is possible to initialize some or all fields of a structure variable at once, when it is declared.
8. Structure type allows the efficient insertion and deletion of elements but arrays cause the inefficiency.

9. For random array accessing, large hash tables are needed. Hence, large storage space and costs are required.
10. When structure variable is created, all of the member variables are created automatically and are grouped under the given variable's name.

5.5 Structure Contains Pointers

A pointer variable can also be used as a member in the structure. Example:

```
struct
{
int *p1; int * p2;
} *rr;
```

In the above, *rr is a pointer variable of structure type which holds inside it another two pointer variables p1 and p2 as its members.

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

```
{
sturct
int *p1, *p2;
} *rr;
int a, b ;
a = 70;
b = 100'
rr — p1 = &a; rr — p2 = & b;
printf( " The contents of pointer variables");
```

```
printf( " Present in the structure as members are \n"); printf ( '%d %d', *rr — p1, *rr —
p2);
}
```

In the above program, two pointer variables p1 and p2 are declared as members of the structure and their contents / variables are printed after assignment in the program.

Self Referential Structures

Structures can have members which are of the type the same structure itself in which they are included, This is possible with pointers and the phenomenon is called as self referential structures.

A self referential structure is a structure which includes a member as pointer to the present structure type.

The general format of self referential structure is struct parent

```
{  
memeber1; memeber2;  
_____  
_____  
struct parent *name;  
};
```

The structure of type parent is contains a member, which is pointing to another structure of the same type i.e. parent type and name refers to the name of the pointer variable.

Here, name is a pointer which points to a structure type and is also an element of the same structure.

Example

```
struct element  
{  
char name{20}; int num;  
struct element * value;  
}
```

Element is of structure type variable. This structure contains three members

- a 20 elements character array called **name**
- An integer element called **num**

a pointer to another structure which is same type called **value**. Hence it is self referential structure.

These structure are mainly used in applications where there is need to arrange data in ordered manner.

5.6 Unions

Introduction

A Union is a collection of heterogeneous elements. That is, it is a group of elements; each element is of different type. They are similar to structures. However, there is a difference in the way the structures members and union members are stored. Each member within a structure is assigned its own memory location. But the union members all share the common memory location. Thus, unions are used to save memory. Unions are chosen for applications involving multiple members, where values need to be assigned to all of the members at any one time.

5.7 Definition of Union

Union is a data type through which objects of different types and sizes can be stored at different times.

The general form of union type variable declaration is Union name

```
{
data type member-1;
    data type member-2; data type member-3;
.....
    ..... data type member-n;
}
```

The declaration includes a key word Union to declare the union data type. It is followed by user defined name, followed by curly braces which includes the members of the union

Example

```
union          value
{
int no; float sal; char sex;
};
```

Characteristics of Union

1. Union stores values of different types in a single location in memory
2. A union may contain one of many different types of values but only one is stored at a time.
3. The union only holds a value for one data type. If a new assignment is made the previous value has no validity.
4. Any number of union members can be present. But, union type variable takes the largest memory occupied by its members.

5.8 Differences between Structure and Unions

Structure	Union
<p>1. Struct Structure Name</p> <pre>{ datatype member-1; datatype member-2; datatype member-n; };</pre> <p>2. Every structure member is allocated memory when a structure variable is defined</p> <p>3. All the members can be</p>	<p>1. Union name { datatype member-1; datatype member-2; datatype member-n; };</p> <p>2. The memory equivalent to the largest item is allocated commonly for all members</p> <p>3. Values assigned to one member may cause the change in value of other members.</p> <p>4. Only one union member can be</p>

<p>assigned values at a time</p> <p>4. All members of a structure can be initialized at the same time</p> <p>5. value assigned to one member will not cause the change in other members</p> <p>6. The usage of structure is efficient when all members are actively used in the program</p>	<p>initialized at a time</p> <p>5. Value assigned to one member may cause the change in value of other members.</p> <p>6. The usage of union is efficient when members of it are not required to be accessed at the same time.</p>
---	--

Self assessment questions:

1. Importance of Structures

1. Why are structures used in C programming, and how do they improve code organization?
2. Give an example of a real-world scenario where using a structure would be advantageous. Describe the structure and its members.

2. Definition of Structure

1. Write a C code snippet that defines a structure named Book with the following members: title (a string), author (a string), and price (a float).
2. How would you declare a variable of the Book structure and initialize its members?

3. Implementation of Arrays in Structures

1. Modify the Book structure from the previous question to include an array of integers representing the book's chapter numbers. Declare the array to hold up to 10 chapters.
2. Write a C code snippet that initializes the chapters array within a Book structure with values from 1 to 10.

4. Definition of Union

1. Define a union named Info that can store either an integer, a float, or a string. Provide a code example of how you would declare and initialize this union.
2. Explain what happens to the value of a union when you assign a new value to a different member.

5. Union Declaration

1. Write a C code snippet that declares a union named Data with members: integerValue, floatValue, and charValue. Demonstrate how to assign a value to each member and print the results.
2. What is the size of the union Data if integerValue is an int (4 bytes), floatValue is a float (4 bytes), and charValue is a char (1 byte)?

6. Differences Between Structures and Unions

1. Compare and contrast the memory allocation and access of members in structures and unions. How does this affect their use cases?
2. Given the following structure and union:

c

Copy code

```
struct Employee {  
    char name[50];  
    int id;  
    float salary;  
};
```

```
union Value {  
    int intValue;  
    float floatValue;  
    char charValue;  
};
```

How would you access and modify the name member of the Employee structure and the intValue member of the Value union? What happens if you access floatValue after setting intValue?

Answers for Self-Assessment Questions

1. Importance of Structures

1. **Answer:** Structures group related data into a single unit, which simplifies managing complex data and enhances code organization. They allow you to handle multiple attributes together rather than separately, improving readability and maintainability.
2. **Answer:** For example, a structure to represent a Car might include members such as make, model, and year. This structure allows you to manage all attributes of a car together, simplifying functions that operate on car data.

2. Definition of Structure

1. **Answer:**

c

Copy code

```
struct Book {  
    char title[100];  
    char author[100];  
    float price;  
};
```

2. **Answer:**

c

Copy code

```
struct Book myBook;  
myBook.price = 19.99;  
strcpy(myBook.title, "The Great Gatsby");  
strcpy(myBook.author, "F. Scott Fitzgerald");
```

3. Implementation of Arrays in Structures

1. **Answer:**

c

Copy code

```
struct Book {  
    char title[100];  
    char author[100];  
    float price;  
    int chapters[10]; // Array to hold up to 10 chapter numbers  
};
```

2. **Answer:**

c

Copy code

```
struct Book myBook;  
for (int i = 0; i < 10; i++) {  
    myBook.chapters[i] = i + 1;  
}
```

4. Definition of Union

1. **Answer:**

c

Copy code

```
union Info {  
    int integerValue;  
    float floatValue;  
    char stringValue[100];  
};
```

```
union Info myInfo;
```

```
myInfo.integerValue = 10; // Initialize integerValue
```

2. **Answer:** When a new value is assigned to a different member of a union, the previous value is overwritten because all members share the same memory space. Only the last assigned value is valid.

5. Union Declaration

1. **Answer:**

c

Copy code

```
union Data {  
    int integerValue;  
    float floatValue;  
    char charValue;  
};
```

```
union Data myData;
```

```
myData.integerValue = 10;
```

```
printf("Integer Value: %d\n", myData.integerValue);
```

```
myData.floatValue = 3.14;
```

```
printf("Float Value: %f\n", myData.floatValue);
```

2. **Answer:** The size of the union Data would be the size of its largest member. In this case, if intValue and floatValue are each 4 bytes and charValue is 1 byte, the size of Data is 4 bytes.

6. Differences Between Structures and Unions

1. **Answer:** Structures allocate separate memory for each member, allowing simultaneous access to all members. Unions share the same memory location for all members, so only one member can be accessed at a time. Structures are used for grouping related data, while unions are used for storing different data types in the same memory space.

2. **Answer:**

- To access and modify name in Employee:

c

Copy code

```
struct Employee emp;  
strcpy(emp.name, "Alice");
```

- To access and modify intValue in Value:

c

Copy code

```
union Value val;  
val.intValue = 10;  
printf("Int Value: %d\n", val.intValue);
```

- Accessing floatValue after setting intValue will show unpredictable results because the value of floatValue may not be meaningful; only the last set value is valid.

○

Activities and Exercises

1. Importance of Structures

Activity:

- **Create a Structure for a Library System:** Define a structure to represent a book in a library system. Include fields for title, author, ISBN, and yearPublished. Write functions to create and display books.

Exercise:

- **Enhance the Library System:** Extend your structure to include a list of books (use an array of structures). Write functions to add a book to the list, find a book by ISBN, and display all books in the list.

2. Definition of Structure

Activity:

- **Define a Student Structure:** Create a structure named Student with fields for name, rollNumber, and marks (array of 5 integers). Write a program to input details for 3 students and display their information.

Exercise:

- **Manipulate the Structure:** Write a function that calculates the average marks of a student and another function that finds the highest marks among the students. Use the Student structure you created.

3. Implementation of Arrays in Structures

Activity:

- **Student Grades Tracker:** Define a structure to represent a student with an array of grades for different subjects. Write a program to input grades for a student, calculate the average grade, and display the student's details.

Exercise:

- **Classroom Analysis:** Extend the previous activity by defining an array of students. Write functions to calculate the average grade for each student and the overall average grade for the entire class.

4. Definition of Union

Activity:

- **Define and Use a Union:** Create a union named Data that can store an int, float, or char. Write a program to demonstrate storing and accessing each type of data, and display the values.

Exercise:

- **Union with Different Types:** Write a function that takes a union as a parameter and prints the value based on a flag that indicates the type of data stored in the union (int, float, or char).

5. Union Declaration

Activity:

- **Union Variable Initialization:** Define a union with members for different data types. Create a union variable and initialize it with values for each type. Display the values to show how the memory is shared.

Exercise:

- **Union Size and Access:** Write a program that prints the size of a union and demonstrates the effects of accessing different members of the union. Observe how the value of one member affects the others.

6. Differences Between Structures and Unions

Activity:

- **Structure vs. Union Comparison:** Implement two similar programs: one using structures and the other using unions. For example, create a Vehicle structure and union with members for make, model, and year. Compare the memory usage and access methods.

Exercise:

- **Practical Comparison:** Write a program that uses a structure to represent a Person with multiple attributes and a union to represent a Value that can be either an int, float, or char. Demonstrate how each approach works and discuss their advantages.

Case Studies

Case Study 1: Employee Management System

Scenario: You are developing an employee management system where you need to store employee details and manage different types of employee data.

Tasks:

1. **Define Structures:** Create a structure for employee details with fields like name, employeeID, and department. Implement an array of structures to handle multiple employees.
2. **Data Management:** Write functions to add new employees, display employee information, and search for employees by ID.

3. **Union for Employment Status:** Define a union to store employment status information, such as full-time, part-time, or contract. Include this union as a member in the employee structure.

Hints:

- Use functions to manage the employee data and the union for employment status.
- Ensure to handle data integrity when using the union.

Case Study 2: Product Inventory System

Scenario: You are tasked with creating a product inventory system where each product can have different types of attributes.

Tasks:

1. **Define Structures:** Create a structure for Product with fields for productID, name, and a union for price (which can be a float for retail or int for wholesale).
2. **Inventory Management:** Implement an array of Product structures. Write functions to add products, display product details, and update prices.
3. **Union Usage:** Demonstrate how the union can store different types of price values and how you handle them when displaying product information.

Hints:

- Use functions to handle product inventory and manage the union effectively.
- Consider edge cases where different price types might be used.

Case Study 3: Student Information System with Academic Records

Scenario: You are developing a system to manage student records, including personal information and academic results.

Tasks:

1. **Define Structures:** Create a Student structure with personal details (e.g., name, studentID) and an array of structures for academic records (e.g., courseName, grade).
2. **Academic Records:** Implement functions to input academic records, calculate GPA, and display student information.
3. **Union for Scholarship Status:** Define a union within the Student structure to represent scholarship status (e.g., scholarshipAmount, scholarshipType).

Hints:

- Use nested structures to manage academic records efficiently.
- Ensure proper handling of the union to store and retrieve scholarship status.

Summary

1. Importance of Structures

Structures in C are crucial for organizing and managing related data items under a single name. They allow you to group different data types together into a single entity, making it easier to handle complex data. For example, a structure can be used to represent a record in a database, such as a student with attributes like name, roll number, and grades, all of which are of different data types but are logically related.

Key Points:

- **Organizes Related Data:** Structures help in grouping related data items together, making programs more manageable.
- **Improves Code Readability:** By using structures, code becomes more readable and maintainable.
- **Facilitates Complex Data Management:** Structures are essential for handling complex data that involves multiple attributes.

2. Definition of Structure

A structure in C is a user-defined data type that allows the combination of variables of different types into a single unit. Each variable within a structure is known as a member or field. Structures are defined using the struct keyword.

Example:

c

Copy code

```
struct Student {  
    char name[50];  
    int rollNumber;  
    float grades;  
};
```

In this example, Student is a structure with three members: name, rollNumber, and grades.

3. Implementation of Arrays in Structures

Structures can include arrays as members, allowing you to manage collections of related data within a single structure. This is useful for representing data such as a list of subjects or scores for a student.

Example:

c

Copy code

```
struct Student {
    char name[50];
    int rollNumber;
    float grades[5]; // Array to hold grades for 5 subjects
};
```

Here, the grades member is an array of float, which allows storing multiple grades within each Student structure.

4. Definition of Union

A union in C is a special data type that allows storing different data types in the same memory location. Unlike structures, where each member has its own memory location, unions share the same memory location for all their members. This means that only one member can be used at a time, and the size of the union is determined by the size of its largest member.

Example:

c

Copy code

```
union Data {
    int intValue;
    float floatValue;
    char charValue;
};
```


In this example, Data is a union that can hold an int, a float, or a char, but only one of these types can be used at a time.

5. Union Declaration

A union is declared using the union keyword, followed by the union name and its members. The syntax is similar to that of structures but with shared memory for its members.

Example:

```
c
Copy code
union Data {
    int intValue;
    float floatValue;
    char charValue;
};
```

You can then create a variable of this union type and assign values to its members, but only one member can be accessed at a time.

6. Differences Between Structures and Unions

- **Memory Allocation:**
 - **Structure:** Each member of a structure has its own memory location, and the size of the structure is the sum of the sizes of all its members.
 - **Union:** All members of a union share the same memory location, so the size of the union is equal to the size of its largest member.
- **Data Access:**
 - **Structure:** All members of a structure can be accessed simultaneously.
 - **Union:** Only one member of a union can be accessed at a time because all members share the same memory space.
- **Use Cases:**
 - **Structure:** Suitable for storing and managing multiple related pieces of data that need to be accessed together.
 - **Union:** Useful when you need to store different types of data in the same memory location but only need to use one type at a time.

- **Initialization:**
 - **Structure:** Each member can be initialized independently.
 - **Union:** Only the first member can be initialized, as initializing one member affects the value of the others.

Glossary:

1. Importance of Structures

- **Definition:** Structures are essential in C programming for grouping different types of data into a single unit. They help in organizing complex data and improving code readability and maintainability.
- **Use Cases:** Structures are used to represent entities with multiple attributes, such as a record in a database, a student's information, or a configuration setting.

2. Definition of Structure

- **Definition:** A structure is a user-defined data type that combines variables of different types into a single entity. Each variable within a structure is called a member or field.

- **Syntax:**

c

Copy code

```
struct StructureName {
    dataType member1;
    dataType member2;
    // Additional members
};
```

Example:

c

Copy code

```
struct Student {
    char name[50];
    int rollNumber;
```

```
float grades;
};
```

3. Implementation of Arrays in Structures

- **Definition:** Structures can include arrays as members. This allows a structure to manage multiple values of the same type within a single field.

- **Syntax:**

```
c
Copy code
struct StructureName {
    dataType arrayName[arraySize];
};
```

Example:

```
c
Copy code
struct Student {
    char name[50];
    int rollNumber;
    float grades[5]; // Array of grades
};
```

4. Definition of Union

- **Definition:** A union is a special data type that allows storing different data types in the same memory location. All members of a union share the same memory, so only one member can be used at a time.

- **Syntax:**

```
c
Copy code
union UnionName {
    dataType member1;
    dataType member2;
    // Additional members
};
```

Example:

c

Copy code

```
union Data {  
    int intValue;  
    float floatValue;  
    char charValue;  
};
```

5. Union Declaration

- **Definition:** Declaring a union involves using the union keyword followed by the union's name and its members. Each member of the union shares the same memory space.
- **Syntax:**

c

Copy code

```
union UnionName {  
    dataType member1;  
    dataType member2;  
    // Additional members  
};
```

Example:

c

Copy code

```
union Data {  
    int intValue;  
    float floatValue;  
    char charValue;  
};
```

6. Differences Between Structures and Unions

- **Memory Allocation:**

- **Structure:** Each member has its own memory location. The total size of the structure is the sum of the sizes of all its members.
- **Union:** All members share the same memory location. The size of the union is equal to the size of its largest member.
- **Data Access:**
 - **Structure:** All members can be accessed simultaneously and independently.
 - **Union:** Only one member can be accessed at a time since all members use the same memory space.
- **Use Cases:**
 - **Structure:** Used when you need to store multiple attributes of an entity together, such as in a record or a complex data object.
 - **Union:** Used when you need to store different types of data in the same memory space but only need one type of data at a time.
- **Initialization:**
 - **Structure:** Each member can be initialized individually.
 - **Union:** Only the first member can be initialized, as initializing one member affects the value of the others.

Suggested Readings

1. "Problem Solving with Algorithms and Data Structures Using Python" by Bradley N. Miller and David L. Ranum, 2013.
2. "Problem Solving: A Statistician's Guide" by Steven A. Cohen, 2013.
3. "Computer Science: An Overview" by J. Glenn Brookshear, 2012.
4. "Structured Computer Organization" by Andrew S. Tanenbaum, 2013.

5. **"The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie, 2015.**
6. **"Algorithm Design: A Methodological Approach" by Jon Kleinberg and Éva Tardos, 2006.**
7. **"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, 2022.**
8. **"C Programming for the Absolute Beginner" by Michael Vine, 2008.**
9. **"Programming in C" by Stephen G. Kochan, 2014.**