

R.M.K GROUP OF ENGINEERING INSTITUTIONS



R.M.K
GROUP OF
INSTITUTIONS

R.M.K GROUP OF INSTITUTIONS



R.M.K
GROUP OF
INSTITUTIONS



Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

22CS101

Problem Solving using C++

Department : Computer Science & Engineering

Batch / Year : 2023 – 2027 / I

Created by : Subject Handling Faculties

Date : 06.09.2023

1.CONTENTS

S.NO	CONTENTS
1	Contents
2	Course Objectives
3	Prerequisites
4	Syllabus
5	Course Outcomes
6	CO-PO Mapping
7	Lecture Plan
8	Activity Based Learning
9	Lecture Notes
10	Assignments
11	Part- A Questions & Answers
12	Part-B Questions
13	Supportive Online Courses
14	Real Time Applications
15	Content beyond the Syllabus
16	Assessment Schedule
17	Prescribed Text books & Reference Books
18	Mini Project Suggestions

Unit - 1 Content

Chapter No.	CONTENTS
1.1	Computational thinking for Problem Solving
1.2	Algorithmic thinking for Problem solving
1.3	Building Blocks -
1.4	Problem Solving and Decomposition
1.5	Dealing with Error
1.6	Evaluation.
1.7	Overview of C
1.8	Data types
1.9	Identifiers – Variables
1.10	Storage Class Specifiers
1.11	Constants
1.12	Operators
1.13	Expressions
1.14	Statements
1.15	Arrays and Strings
1.16	Single Dimensional Array
1.17	Two Dimensional Array
1.18	Arrays of strings
1.19	Multidimensional Arrays

2. Course Objectives

- To learn problem solving and programming fundamentals.
- To gain knowledge on pointers and functions.
- To apply the principles of object orientated programming.
- To understand operator overloading, inheritance and polymorphism.
- To use the functionalities of I/O operations, files build C++ programs using exceptions.



3. Prerequisites


22CS101 – Problem Solving Using C++

- ✓ Programming in C
- ✓ Logical Thinking
- ✓ Basic of Mathematics



4. Syllabus

22CS101	PROBLEM SOLVING USING C++	L	T	P	C
	(Lab Integrated)	3	0	2	4
OBJECTIVES: <ul style="list-style-type: none">To learn programming fundamentals in C.To gain knowledge on pointers and functions.To apply the principles of classes and objectsTo develop a C++ application with object oriented concepts.To use the functionalities of I/O operations, files build C++ programs using exceptions.					
UNIT I	PROGRAMMING FUNDAMENTALS				15
Computational thinking for Problem solving – Algorithmic thinking for problem Solving- Building Blocks - Problem Solving and Decomposition –Dealing with Error – Evaluation. Overview of C – Data types – Identifiers – Variables – Storage Class Specifiers – Constants – Operators - Expressions – Statements – Arrays and Strings – Single-Dimensional – Two-Dimensional Arrays – Arrays of Strings – Multidimensional Arrays.					
UNIT II	POINTERS AND FUNCTIONS				15
Pointers -Variables – Operators – Expressions – Pointers and Arrays – Functions - Scope Rules – Function Arguments – return Statement – Recursion – Structures – Unions – Enumerations.					
UNIT III	CLASSES AND OBJECTS				15
Concepts of Object Oriented Programming – Benefits of OOP – Simple C++ program - Classes and Objects - Member functions - Nesting of member functions - Private member functions - Memory Allocation for Objects - Static Data Members - Static Member functions - Array of Objects - Objects as function arguments - Returning objects - friend functions – Const Member functions - Constructors – Destructors					
UNIT IV	OPERATOR OVERLOADING , INHERITANCE AND POLYMORPHISM				15
Operator Overloading - Overloading Using Friend functions – Inheritance – Types of inheritance – Virtual Base Class - Abstract Class – Constructors in Derived Classes - member class: nesting of classes. Pointer to objects – this pointer- Pointer to derived Class - Virtual functions – Pure Virtual Functions – Polymorphism					
UNIT V	I/O, FILES AND EXCEPTIONS				15
C++ Streams – Unformatted I/O - Formatted Console I/O – Opening and Closing File – File modes - File pointers and their manipulations – Templates – Class Templates – Function Templates - Exception handling.					
Lab Exercises <ol style="list-style-type: none">Write C/C++ programs for the following:<ol style="list-style-type: none">Find the sum of individual digits of a positive integer.Compute the GCD of two numbers.Find the roots of a number (Newton’s method)Write C/C++ programs using arrays:<ol style="list-style-type: none">Find the maximum of an array of numbers.Remove duplicates from an array of numbers.					



R.M.K
GROUP OF
INSTITUTIONS

4. Syllabus contd...

3. Write C/C++ programs using strings:
 - a. Checking for palindrome.
 - b. Count the occurrences of each character in a given word.
4. Generate salary slip of employees using structures and pointers. Create a structure Employee with the following members:
EID, Ename, Designation, DOB, DOJ, Basicpay
Note that DOB and DOJ should be implemented using structure within structure.
5. Compute internal marks of students for five different subjects using structures and functions.
6. Write a program Illustrating Class Declarations, Definition, and Accessing Class Members.
7. Program to illustrate default constructor, parameterized constructor and copy constructors
8. Write a Program to Demonstrate the i) Operator Overloading. ii) Function Overloading.
9. Write a Program to Demonstrate Friend Function and Friend Class.
10. Program to demonstrate inline functions.
11. Program for Overriding of member functions.
12. Write C++ programs that illustrate how the following forms of inheritance are supported:
 - a) Single inheritance b) Multiple inheritance c) Multi level inheritance d) Hierarchical inheritance
13. Program to demonstrate pure virtual function implementation.
14. Count the number of account holders whose balance is less than the minimum balance using sequential access file.
15. Write a Program to Demonstrate the Catching of all Exceptions.
16. Mini project.

TOTAL: 45+30 = 75 PERIODS

TEXT BOOKS:

1. Herbert Schildt, "The Complete Reference C++", 4th edition, MH, 2015. (Unit 1 & 2)
2. E Balagurusamy, "Object Oriented Programming with C++", 4th Edition, Tata McGraw-Hill Education, 2008. (Unit 3, 4 & 5)

REFERENCES:

1. Nell Dale, Chip Weems, "Programming and Problem Solving with C++", 5th Edition, Jones and Bartlett Publishers, 2010.
2. John Hubbard, "Schaum's Outline of Programming with C++", MH, 2016.
3. Yashavant P. Kanetkar, "Let us C++", BPB Publications, 2020
4. ISRD Group, "Introduction to Object-oriented Programming and C++", Tata McGraw-Hill Publishing Company Ltd., 2007.
5. D. S. Malik, "C++ Programming: From Problem Analysis to Program Design", Third Edition, Thomson Course Technology, 2007.
6. https://infyspringboard.onwingspan.com/web/en/aptoc_lex_auth_01297200240671948837_shared/overview

5. Course Outcomes

CO1: Solve problems using basic constructs in C.

CO2: Implement C programs using pointers and functions.

CO3: Apply object-oriented concepts and solve real world problems.

CO4: Develop C++ programs using operator overloading and polymorphism.

CO5: Implement C++ programs using Files and Exceptions.



6. CO – PO Mapping

	POs and PSOs														
COs	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PS 01	PS 02	PS 03
C01	3	3	3					1	1	1		1	1	1	1
C02	3	3	3					1	3	3		3	1	1	1
C03	2	2	2					3	3	1		1	1	1	1
C04	3	3	3					3	3	3		1	1	1	1
C05	3	3	3					1	1	1		1	1	1	1



R.M.K.
GROUP OF
INSTITUTIONS

7. Lecture Plan - Unit I

S. No.	Topic	No. of Periods	Proposed Date	Actual Lecture Date	Pertaining CO	Taxonomy Level	Mode of Delivery
1	Computational thinking for Problem solving	1			CO1	K2	Chalk & Talk
2	Algorithmic thinking for Problem solving	1			CO1	K3	Chalk & Talk
3	Building Blocks - Problem Solving and Decomposition	1			CO1	K2	Chalk & Talk
4	Dealing with Error – Evaluation.	1			CO1	K2	Chalk & Talk
5	Overview of C – Data types	1			CO1	K2	Chalk & Talk
6	Identifiers-Variables	1			CO1	K2	Chalk & Talk
7 & 8	Storage Class Specifiers	1			CO1	K3	Chalk & Talk
9	Constants	1			CO1	K2	Chalk & Talk
10	Operators - Expressions	1			CO1	K2	Chalk & Talk
11	Statements	1			CO1	K3	Chalk & Talk
12	Arrays :Single-Dimensional	1			CO1	K2	Chalk & Talk
13	Two-Dimensional Arrays	1			CO1	K2	Chalk & Talk
14	Multidimensional Arrays.	1			CO1	K2	Chalk & Talk
15	Arrays of Strings	1			CO1	K2	Chalk & Talk

8. Activity Based Learning

Learning Method	Activity
Learn by solving problems	Tutorial Sessions can be conducted Tutorial sessions available in Skillrack for practice
Learn by questioning	Quiz
Learn by doing hands-on	Practice in Lab



R.M.K.
GROUP OF
INSTITUTIONS

9. Lecture Notes

UNIT I PROGRAMMING FUNDAMENTALS

15

Computational thinking for Problem Solving – Algorithmic thinking for problem solving – Building Blocks – Problem solving and Decomposition – Dealing with Error-Evaluation.

Overview of C – Data types – Identifiers – Variables – Storage Class Specifiers – Constants – Operators - Expressions – Statements – Arrays and Strings – Single-Dimensional – Two-Dimensional Arrays – Arrays of Strings – Multidimensional Arrays.



R.M.K.
GROUP OF
INSTITUTIONS

1.0 Introduction to Computational Thinking for Problem Solving

In the rapidly evolving landscape of computer science and engineering, the ability to think critically and creatively about complex problems is paramount. This is where the concept of "Computational Thinking" emerges as a powerful mental framework that equips individuals to address a wide array of challenges in these fields. In this introductory page, we will explore the fundamental principles and significance of Computational Thinking, and its pivotal role in problem-solving within the realms of computer science and engineering.

Computational Thinking

Computational Thinking is a problem-solving approach inspired by the thought processes of computer scientists and engineers. It encapsulates a set of cognitive skills and strategies that help individuals dissect intricate problems, formulate precise solutions, and translate these solutions into algorithmic or systematic procedures. While it has deep roots in computer science, it transcends the boundaries of a single discipline and finds applications across various domains.

The Significance of Computational Thinking

Computational Thinking is not confined to the realm of academia or coding. It has broader applications in everyday life and is increasingly essential in a technology-driven world. Here are a few reasons why it holds such significance:

Problem-Solving Powerhouse: It equips individuals with the tools to tackle complex problems in diverse fields, from healthcare and finance to transportation and entertainment.

Cross-Disciplinary Relevance: Computational Thinking transcends disciplines, making it a valuable skill for professionals in computer science, engineering, data science, and beyond.

Innovation and Automation: It underpins innovation by enabling the creation of new technologies and drives automation to enhance efficiency in various industries.

Critical Thinking and Creativity: It fosters critical thinking, creativity, and adaptability, empowering individuals to navigate the ever-changing landscape of technology.

In this era of rapid technological advancement, Computational Thinking is not just a skill; it's a mindset that empowers problem solvers to shape the future. As we delve deeper into this topic, we will explore its applications, methodologies, and real-world examples that illustrate its transformative potential in computer science and engineering.

1.1 Computational thinking for Problem solving

Definition :

Computational thinking (CT) is the mental skill to apply concepts, methods, problem solving techniques, and logic reasoning, derived from computing and computer science, to solve problems in all areas, including our daily lives.

In education, CT is a set of problem-solving methods that involve expressing problems and their solutions in ways that a computer could also execute. It involves automation of processes, but also using computing to explore, analyze, and understand processes (natural and artificial).

Computational thinking is an interrelated set of skills and practices for solving complex problems, a way to learn topics in many disciplines, and a necessity for fully participating in a computational world.

Four cornerstones of computational thinking: decomposition, pattern recognition, abstraction, and algorithms. Decomposition invites students to break down complex problems into smaller, simpler problems. Numerous research papers discuss the importance of computational thinking in many diverse careers, such as:

➤ **Natural sciences:**

- ❖ computational biology;
- ❖ genomics;
- ❖ applied physics;
- ❖ climate change;
- ❖ astronomy.

➤ **Social Sciences :**

- ❖ population analysis.
- ❖ Social studies

➤ **Medicine :**

- ❖ disease analysis;
- ❖ medical imaging;
- ❖ clinical practice.

➤ **linguistics;**

➤ **law;**

➤ **music;**

➤ **teaching.**

The following are examples from other sources discussing specific examples of CT in action.

Example: Pipelining a graduation ceremony

Dean Randy Bryant was pondering how to make the diploma ceremony at commencement go faster. By careful placement of where individuals stood, he designed an efficient pipeline so that upon the reading of each graduate's name and honors by Assistant Dean Mark Stehlik, each person could receive his or her diploma, then get a handshake or hug from Mark, and then get his or her picture taken.

This pipeline allowed a steady stream of students to march across the stage (though a pipeline stall occurred whenever the graduate's cap would topple while getting hug from Mark).

Example : Predicting Climate change:

Predicting global climate change is only possible because of advanced computer models. According to the UK Met Office, 'The only way to predict the day-to-day weather and changes to the climate over longer timescales is to use computer models.

1.2 Algorithmic thinking for Problem solving

If we want to build functioning systems based on rules, then logic alone isn't sufficient. We need something that can integrate all these rules and execute actions based on the outcomes of evaluating them. That something is algorithms, and they are the power behind all real-world computational systems.

Algorithms are a way of specifying a multi-step task, and are especially useful when we wish to explain to a third party (be it human or machine) how to carry out steps with extreme precision. As with logic, humans already have an intuitive understanding of algorithms. But, at the same time, a rich and precise science dictates exactly how algorithms work.

Gaining a deeper understanding of this will improve your algorithmic thinking. This is important because a correct algorithm is the ultimate basis of any computer-based solution.

Algorithm Definition:

An algorithm is a step by step procedure for solving any problem. It consists of finite set of unambiguous rules (instructions) which specify a finite sequence of operations that provides the solution to a problem.

It is also a precise rule (or set of rules) specifying how to solve a problem.

An algorithm is a sequence of finite instructions, often used for calculation and data processing.

The word algorithm derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850.

Characteristics of Algorithm

- ❖ The algorithm should be written in sequence. It looks like normal English. The instruction in an algorithm should be repeatedly infinitely. Each step of an algorithm must be exact.
- ❖ An algorithm must be precisely and unambiguously described, so that there remains no uncertainty. An algorithm must terminate.
- ❖ An algorithm must contain a finite number of steps in its execution. An algorithm must be effective. An algorithm must provide the correct answer to the problem.
- ❖ An algorithm must be general.
- ❖ This means that it must solve every instance of the problem.

Qualities of good algorithm

- ❖ The following are the primary factors that are used to judge the quality of an algorithm, Time Memory Accuracy Sequence

Properties of an Algorithm

- ❖ An algorithm is not a program, as they cannot be executed by a computer.
- ❖ Steps must be ordered, unambiguous and finite in number. Ensure that the algorithm has proper termination.
- ❖ There must be no ambiguity in any instruction.
- ❖ There should not be any uncertainty about which instruction to be executed next. An algorithm cannot be open ended.
- ❖ Algorithms can have steps that repeat(iterate) or require decisions(logic and comparison). Algorithm differs in their requirements of time and space.
- ❖ The algorithm should conclude after a finite number of steps.

Example

Problem Statement : Addition of Two Numbers

Step 1: Start

Step 2: Read A,

Step 3: Compute $C = A +$

B

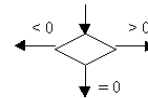
Step 4: Print C

Step 5: Stop

Flowchart Definition:

Flowchart is a pictorial representation of an algorithm in which the steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows.

Boxes represent operations



❖ Arrows represent the sequence in which the operations are implemented. It helps in understanding the logic of the program.

❖ It is not necessary to include all the required steps in detail. It outlines the general procedure.

❖ It provides visual representation of the program, so it is valuable.

Guidelines for preparing Flowcharts



❖ The flow chart should be clear, neat and easy to follow

❖ The flowchart must have logical start and finish

❖ In drawing a proper flowchart, all necessary requirements should be listed in logical order.

❖ Only one flow line should come out from a process symbol. Or Only one flow line should enter a decision symbol. Two or three flow lines may leave the decision symbol

- ❖ Connector symbols are used to reduce the number of flow lines. Useful in testing the validity of the flow chart with normal and unusual test data.
- ❖ The pictorial representation helps in understanding the logic clearly.
- ❖ Assists in finding the key elements of a process by drawing clear flow lines.
- ❖ Communication: It encourages communication among programmers and users

Effective Analysis:

- ❖ It reveals redundant or misplaced steps
- ❖ It helps in analysing the logic of the program by other persons who may or may not know programming techniques.
- ❖ It establishes important areas for monitoring or data collection
- ❖ It identifies areas for improvement.



R.M.K.
GROUP OF
INSTITUTIONS

Useful in Coding:

- ❖ It ensures that no steps are missed while coding. Coding can be done faster.
- ❖ Proper testing and Debugging:
- ❖ It helps in detecting errors.
- ❖ Test the logic with normal/unusual data.

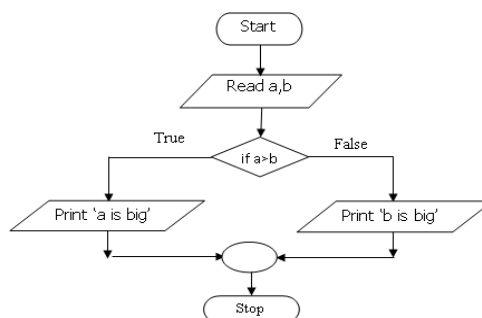
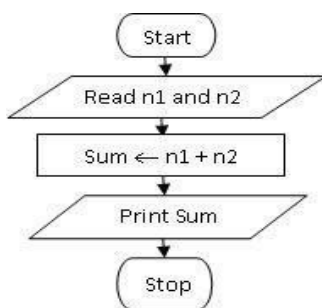
Appropriate Documentation:

- ❖ It serves as a good program documentation tool.
- ❖ It helps in knowing what the program does and how to use the program.
- ❖ Efficient Program Maintenance
- ❖ The maintenance of operating program becomes easy with the help of flowchart.
- ❖ It helps the programmer to put efforts more efficiently on that part.































Limitations

- ❖ Complex
- ❖ It is laborious to draw flow chart for larger programs
- ❖ For large programs it continues to pages making difficult to understand.
- ❖ Difficult to Modify
- ❖ Any changes or modification to a flowchart usually require redrawing the entire logic again.
- ❖ It is not easy to draw thousands of flow lines and symbols, especially for a large complex program.
- ❖ No update
- ❖ Usually, programs are updated regularly but corresponding update of flowcharts may not take place, especially in case of large programs.
- ❖ As a result, the logic used in the flowchart may not match with the actual program's logic.

Ex :To find the sum of two numbers Find the biggest of two numbers



Flowchart Symbols

Sl. no	Symbol	Explanation	Sl. no	Symbol	Explanation
1		Process/operation	16		Card
2		Alternative process	17		Punched tape
3		Decision	18		Summing junction
4		Data	19		Or
5		Predefined process	20		Collate
6		Internal storage	21		Sort
7		Document	22		Extract
8		Multi document	23		Merge
9		Terminator	24		Stored data
10		Preparation	25		Delay
11		Manual input	26		Sequential access
12		Manual operation	27		Magnetic disc
13		Connector	28		Direct access
14		Off-page connector	29		Display
15		Transfer of materials	30		Direction of flow

1.3 Building Blocks :

An algorithm can be constructed from any one of the following basic building blocks. The building blocks of algorithm are listed below:

- (i) Sequence
 - (ii) Selection / decision
 - (iii) Repetition / Iteration
- Instructions / Statements

Instructions are the statements of actions which are to be performed in a sequential order.

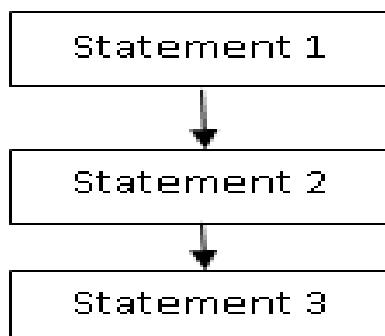
A statement is a unit of code/instruction that has an effect, like creating a variable or displaying a value.

Building Block	Common name
Sequence	Action
Selection	Decision
Iteration	Repetition or Loop

Sequence

- ❖ In sequence construct, statements are executed one by one in the order from top to bottom.
- ❖ The actions are performed in the same sequence (top to bottom) in which they are written.

Flow diagram



Example: Accept two numbers and find their product

Algorithm

step1. Input first number as A

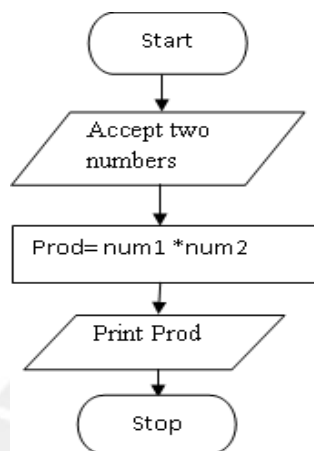
step2. Input second number as B

step3. Set $\text{Prod} = A * B$

step4. Print Prod

step5. Stop

Flowchart

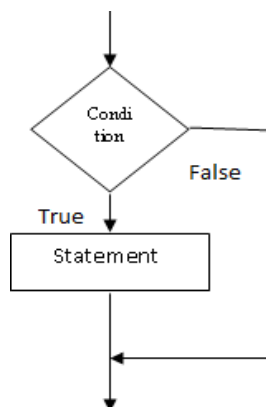


Selection / Decision

Statements are used when the outcome of the process depends on some condition. A condition in this context may evaluate either to a true or a false value.

The statement will be executed when the condition becomes true. If the condition fails, the control shifts out of the consecutive statement.

Flow Diagram :



Example: Accept two numbers, Display the first number if greater than the second.

Algorithm

step1. Start

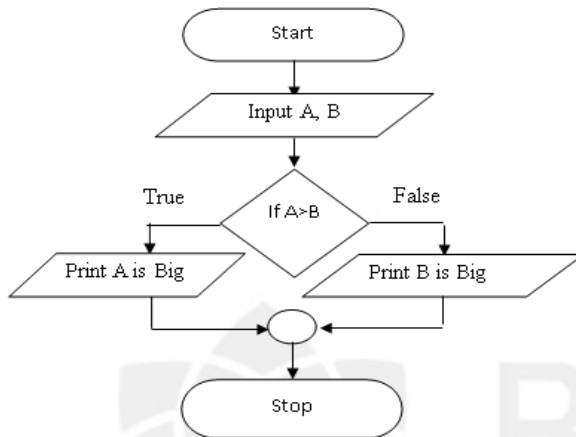
step2. Accept two numbers and store the variables in A and B.

step3. If $(A > B)$ then Display A is greater than B.

step5. Else Display B is greater than A

step6. Stop

Flowchart

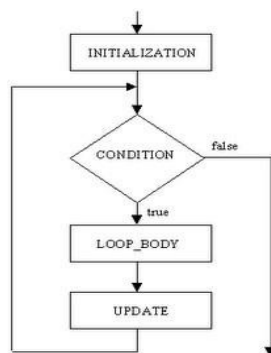


Repetition/Loops

Repetition involves in executing one or more steps for a number of times.

These statements execute one or more steps until some condition is true.

Flow Diagram



Example: Factorial of a given number (Repetition)

Algorithm

Step 1: Start

Step 2: Get the value of n

Step 3: Assign $f=1$, $i=1$

Step 4: check $i \leq n$ repeat step 5 and 6

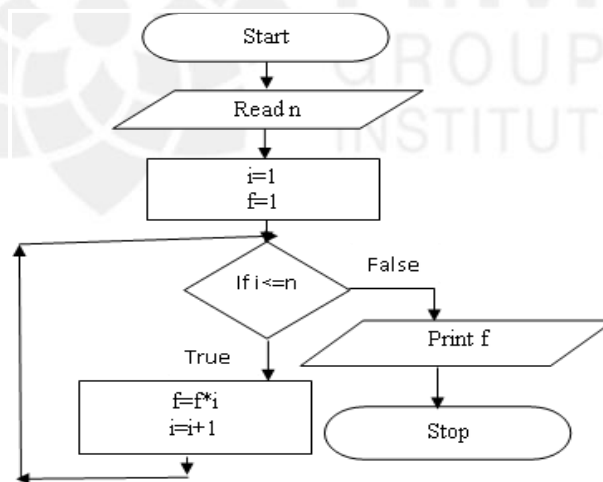
Step 5: calculate $f=f*i$

Step 6: calculate $i=i+1$

Step 7: Print the value of f

Step 8: Stop

Flow chart: Factorial of a given number (Repetition)



1.4 Problem Solving and Decomposition

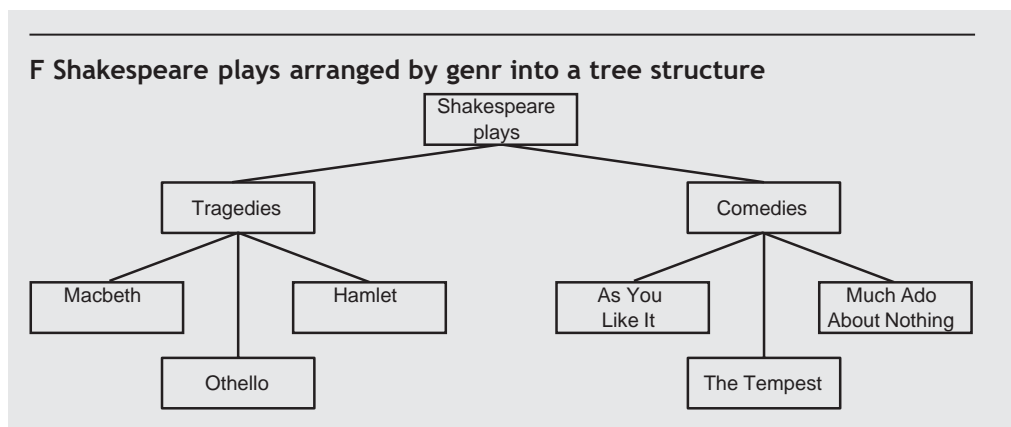
Computational thinking promotes one of these heuristics to a core practice: **decomposition**, which is an approach that seeks to break a complex problem down into simpler parts that are easier to deal with. Its particular importance to CT comes from the experiences of computer science. Programmers and computer scientists usually deal with large, complex problems that feature multiple interrelated parts.

While some other heuristics prove useful some of the time, decomposition almost invariably helps in managing a complex problem where a computerized solution is the goal.

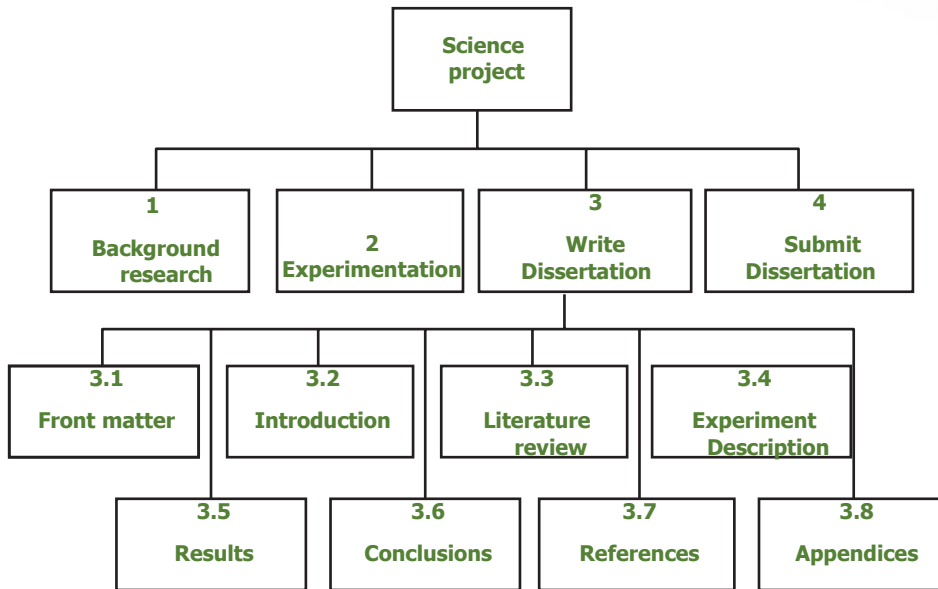
Decomposition is a divide-and-conquer strategy, something seen in numerous places outside computing:

- ❖ Generals employ it on the battlefield when outnumbered by the enemy. By engaging only part of the enemy forces, they neutralise their opponent's advantage of numbers and defeat them one group at a time.
- ❖ Politicians use it to break opposition up into weaker parties who might otherwise unite into a stronger whole.

When faced with a large, diverse audience, marketers segment their potential customers into different stereotypes and target each one differently.



Science project task breakdown



This has decomposed the task somewhat, but some sub-tasks may be still too big. Those tasks should be broken down further by applying the same decomposition process. For example, the front matter of a dissertation is made up of several parts, such as:

- ❖ 3.1.1 write title page;
- ❖ 3.1.2 write copyright section;
- ❖ 3.1.3 write abstract;
- ❖ 3.1.4 write contents section;
- ❖ 3.1.5 write list of figures section.

Decomposition won't result in a complete plan of action, but it can give you a starting point for formulating one. For example, a fully decomposed problem definition can show all the individual tasks, but doesn't necessarily show the order in which you should tackle them. However, relationships between individual sub-problems should become apparent.

Decomposition aids collaboration. If you decompose a problem well (so that the sub-problems can be solved independently), then different people can work on different tasks, possibly in parallel.

1.5 Dealing with Errors:

Designing out the Bugs:

This section examines the various types of bugs that can cause erroneous behavior

Typos

During the early planning phases, your solution exists in various design forms: documents, notes, diagrams, algorithm outlines and so on. The simplest way of preventing errors at this stage is to look for elementary mistakes, in both the problem definition and your solution.

Typos include:

- spelling errors;
- incorrect capitalizations;
- missing words;
- wrong words (for example, 'phase' instead of 'phrase');
- numbered lists with incorrect numbering, and so on.

Such mistakes may often be trivial, but any of them could cause problems.

Poor grammar and ambiguities

Like typos, some types of grammatical error are quite easy to locate and correct. It's easy to find the mistake in this sentence. However, some grammatical errors are quite subtle.

Inconsistencies

Eliminating ambiguities helps to counter another source of bugs, namely inconsistencies. An inconsistency arises when parts of a solution separately make statements that make no sense when considered together.

These bugs are harder to locate because each part of the inconsistency might appear in very different contexts.

Logical Error

Simple bugs might be found in logical expressions. For example, the following algorithm is supposed to print the maximum value of two numbers, x and y .

If $x > y$, then print y

Otherwise if $y > x$, then print x

This actually contains two bugs. One should be obvious: the algorithm prints the minimum instead of the maximum.

The second is slightly more subtle: what happens if both numbers are equal? You would expect a value to be printed, but in fact, nothing is printed at all. This example demonstrates that errors can result from expressions that are valid (that is, they can be evaluated without a problem) but whose semantic meaning is incorrect.

Mathematical error

Mathematical errors can also result from seemingly valid expressions. An in-car computer that evaluates the car's efficiency based on the trip counter might use a formula like this:

$$e = m / g$$

where efficiency (e) is the miles travelled (m), divided by the gallons of fuel consumed (g). This will mostly work fine, except that when the trip counter is reset, the gallons consumed becomes zero and dividing by zero is mathematically invalid. The solution would have to take this special case into account.

Being exhaustive

The path that a computer takes through an algorithm varies at different times. It all depends on the parameters passed, as well as other changeable factors in the environment.

Therefore, algorithms should be exhaustive and contain instructions for all conceivable outcomes.

If you fail to account for any of them, the computer's behaviour becomes unpredictable: it might take an unwanted action or it might do nothing when it should in fact do something.

MITIGATING ERRORS

Getting defensive

One way to minimise the effects of errors is to anticipate where a problem might occur and then put some 'protective barriers' in place. Computer science calls this defensive programming.

The approach verifies that certain conditions are as expected before carrying out the main business of the solution. A failure to meet those conditions could either indicate anything from a warning to a potential system failure.

Reacting to problems

whether the system is interactive or not (which dictates whether or not it can ask the user what to do);

whether the system is transactional or not. If it is, then it's possible to undo the changes made during a defective step.

Checking user input

Humans may make mistakes, so it's best to assume that user will enter faulty input at some point. Think of all the ways an input could be wrong.

(i) Entering '90' instead of '90' is an easy enough mistake to make.

(ii) Making sure an account number is the expected eight digits in length is quick and easy. All you have to do is count the number of digits in the input and ensure it's the right amount.

(iii) Incorrect formats: phone numbers, website URLs, email addresses, dates and times are all examples of data that must match a specific format.

TESTING

Testing, a method for locating hidden bugs in a fully or partially working system. We can choose between two approaches to testing, depending on what you want to achieve.

The first is top-down testing, where you test the solution as a whole to ensure that it works. This is most effective at finding design flaws and verifying that the system hangs together well.

The second approach is bottom-up, which requires you to begin by testing the smallest parts of the solution individually. This allows you to verify that they correctly fulfil their own obligations within the whole system and to show that your solution is built on solid foundations.

Advantages and disadvantages of top-down and bottom-up testing

	Advantages	Disadvantages
Top-down	<ul style="list-style-type: none">➤ Effective for finding design flaws.➤ Can be encouraging to test a full system, even if it is incomplete or contains bugs.	<ul style="list-style-type: none">➤ Difficult to apply when the system is incomplete.➤ Requires you to use placeholders³⁶ for the unfinished parts to simulate their behaviour.➤ Harder to localise bugs when errors occur.
Bottom-up	<ul style="list-style-type: none">➤ Easy to apply in early stages of development.➤ Effective for localising problems.	<ul style="list-style-type: none">• Requires you to simulate the controlling parts of the solution.³⁷• Doesn't expose design flaws until much later.

Testing individual parts

By testing only small parts of the solution at once, you make localising the problem easier. Tests that exercise only individual parts are called unit tests

1.6 EVALUATION

You did it. You analysed the problem, broke it down into pieces, came up with a solution design, which you then implemented, tested and debugged. You have something that functions. So, you're finished, right?

Well, not quite.

The work is not over just because you have a solution. Before you can really finish, you have to make sure you've produced a good solution. You must evaluate its quality.

There are many aspects to quality. **Evaluating a solution** involves asking several basic questions, each addressing a specific aspect.

Important questions about the solution include:

- Is it correct? Does it actually solve the problem you set out to solve?
- Is it efficient? Does it use resources reasonably?
- Is it elegant? Is it simple yet effective?
- Is it usable? Does it provide a satisfactory way for the target audience to use it?

Is it Correct :

Technically speaking, program correctness is a deeply theoretical issue in computer science. In academic circles, talk of proving a program to be correct usually refers to mathematical proof. Such a proof, if successful, establishes a program's correctness as incontrovertible fact.

Is it Efficient

Every algorithm requires some amount of resources to do its work. Different algorithms, even ones that solve the same problem, can perform differently in terms of efficiency. This performance is usually measured in terms of time and space.

- Time: the duration of an algorithm's running time, from start to end. The duration can be measured as the number of steps taken during execution.
- Space: the amount of memory storage required by an algorithm to do its work. The question of an algorithm's **efficiency** comes down to its **time** and **space** requirements. If, when you analyse an algorithm you find that its use of resources is acceptable, then it may be deemed efficient.

Is it Elegant :

some aspects of evaluation cause things to become a little fuzzier in this regard. One of these aspects is elegance, something you might associate more with artistic pursuits.

Two different solutions might both solve a problem, but they could be judged apart by the elegance of their respective approaches. This applies not only to software; it's true of other 'functional' disciplines like engineering, science and mathematics.

Roughly speaking, elegance maximises both effectiveness and simplicity at the same time.

Is it Usable:

The users of your solution will be people. Your solution must therefore cater to humans and their particular ways. In other words, it must usable and ought to give users a positive experience.

Usability measures how well something can be used by people in order to achieve their goals.

Usability captures these desires. It formalises them into several components that measure how easy and satisfying it is for people to use a solution. Despite involving humans in the equation, these components are somewhat measurable. They are:

- **Learnability:** how easy is it for users to accomplish basic tasks the first time they encounter the solution?
- **Efficiency:** once users have learned the solution, how quickly can they perform tasks?
- **Memorability:** when users return to the solution after a period of not using it, how easily can they re-establish proficiency?
- **Errors:** how many errors do users make, how severe are these errors and how easily can they recover from the errors?
- **Satisfaction:** how pleasant is it to use the design?

1.7 Overview of C

History of C

- The root of all modern languages is ALGOL, introduced in 1960. ALGOL was the first computer language to use a block structure.
- In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software.
- C was evolved from ALGOL, BCPL and B by Dennis Ritchie at Bell Laboratories in 1972 and it was known as "traditional C".
- The language became more popular after publication of the book 'The C Programming Language' by Brian Kerningham and Dennis Ritchie in 1978. The language came to know as "K&R C".
- To assure that the C language remains standard in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in 1989 known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990.

Features and Application of C Language

- C is a general purpose, structured programming language.
- C is highly portable (i.e., It can be run in different operating systems environments).
- C is a robust language whose rich set of built in functions and operators can be used to write any complex program.
- C has the ability to extend itself. We can continuously add our own functions to the existing library functions.
- C program can be run on different operating systems of the different computers with little or no alteration.
- C is a middle level language, i.e. it supports both the low level language and high level language features.
- C language allows dynamic memory allocation i.e. a program can request the operating system to allocate or release memory at runtime.
- C language allows manipulation of data at the lowest level i.e., bit level manipulation. This feature is extensively useful in writing system software program.
- C has got rich set of operators.
- C can be applied in systems programming areas like Compilers, Interpreters and Assemblers.

Structure of a C Program :

Documentation section
Header Section
Definition section
Global Declaration section
main() function section
<pre>{ Declaration part Executable part }</pre>
Subprogram section
(User defined Functions)

Documentation section

- It consists of comment lines giving the name of the program, author & other details which the programmer would like to use.
- Comments are not necessary in the program.
- It is useful for documentation.
- Comments are non executable statements which are placed between the delimiters `/* */` for multi line comments and started with `//` for single line comments.
- The compiler does not execute comments

Link section

- It provides instruction to the compiler to link functions from the system library.
- Program depends upon some header files for function definition that are used in program.
- Each header file by default is extended with `.h`
- The header file should be included using `#include` directive.

Example : `#include<stdio.h>` or `#include "stdio.h"`

In this example `<stdio.h>` file is included, that is, all the definitions and prototype of function defined in this file are available in the Current program.

Definition section

It defines all symbolic constants.

Global declaration section

- The section declares some variables that are used in more than one function. These variables are known as global variable.
- This section must be declared outside of all the functions

Function main

- Every program written in C language must contain `main()` function.
- Empty parentheses after `main` are necessary.
- The execution of the program always begins with the function `main()`. Declaration part
- The declaration part declares the entire variables that are used in executable part.
- The initializations of variables are done in this section. Initialization means providing initial value to the variables.

Executable part

- This part contains a set of statements or a single statement.
- These statements are enclosed between the braces `{}`.
- All the statements in the program end with a semicolon.

User- Defined function

- The functions defined by the user are called user-defined functions.
- These functions are generally defined after the main() function.
- They can also be defined before main() function. This portion is not compulsory.

Programming Rules

- All statements should be written in lower case letters. Upper case letters are only used for symbolic constants.
- Blank spaces may be inserted between the words.
- The program statements can written anywhere between the two braces following the declaration part.
- The opening and closing braces should be balanced. eg., if opening braces are four, then closing braces should be four.

Sample C Program

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

Character Set

A character denotes any alphabet, digits, white spaces or special symbol used to represent information.

Letters	A to Z and a to z
Digits	0 to 9
White spaces	Blank space, Horizontal tabs, vertical tab, New line, Form feed.
Special character	, comma & Ampersand . period or dot ^ caret ; semicolon * Asterisk : colon - minus " quotation mark + plus etc.,

Tokens, Keywords

C Tokens:

- The C Language program can contain the individual units called the C tokens.
- The tokens are usually referred as individual text and punctuation in the text. C Tokens are,
 - Keyword
 - Identifier
 - Constants
 - Operators

Keyword:

- The c keywords are reserved words by the compiler.
- The keywords cannot be used as variable name.
- All keywords must be written in lower case.
- Keywords are predefined and have standard meaning

auto	double	int	Struct
continue	if	volatile	Break
else	long	Switch	default
signed	while	Case	enum
register	typedef	do	sizeof
char	extern	return	union
for	static	Const	float
short	unsigned	goto	void

Constants

In computer programming, a constant is a value that should not be altered by the program during normal execution, i.e., the value is constant

Rules for declaring a

constant:

Rule 1: Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters

Rule 2: No blank spaces are permitted in between the # symbol and define keyword

Rule 3: Blank space must be used between #define and constant name and constant value

Rule 4: #define is a pre-processor compiler directive and not a statement. Therefore, it does not end with a semi-colon.

Identifiers

An "Identifiers" or "symbols" are the names you supply for variables, types, functions, and labels in your program. Identifier names must differ in spelling and case from any keywords. You cannot use keywords (either C or Microsoft) as identifiers; they are reserved for special use.

Rules

- A variable name can be any combination of alphabets, digits and underscore.
- Identifier should not start with a digit.
- The first character must be an alphabet or an underscore (_).
- No commas or blank space are allowed within a Identifier name.
- No special symbol can be used in a identifier.

1.8 Data Types :

Data types are fundamental building blocks in computer programming and data manipulation. They define the kind of values that variables or data structures can hold and specify the operations that can be performed on those values. In essence, data types categorize data into distinct groups based on their characteristics, such as numeric, text, boolean, or more complex structures like arrays and objects. These distinctions are crucial because they determine how data is stored in memory, how it can be processed, and the constraints applied to it. Data types play a critical role in ensuring the integrity and efficiency of computer programs, as they dictate how data can be used and manipulated within the code, thus enabling programmers to work with data in a structured and meaningful way.

Type	Size	Range
char	1 byte	-127 to 127 or 0 to 255
unsigned	1 byte	0 to 255
signed char	1 byte	-127 to 127
int	4 bytes	-2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
signed int	4 bytes	-2147483648 to 2147483647
short int	2 bytes	-32768 to 32767
unsigned short int	2 bytes	0 to 65535
signed short int	2 bytes	-32768 to 32767
long int	4 bytes	-2147483647 to 2147483647
signed long int	4 bytes	-2147483647 to 2147483647
unsigned long int	4 bytes	0 to 4294967295
float	4 bytes	+/-3.4e +/-38
double	8 bytes	+/-1.7e +/-308
long double	8 bytes	+/-1.7e +/-308

1.9 Identifiers- Variables :

Definition

In programming, a variable is a container (storage area) to hold data. To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location.

- A variable is a data name used for storing a data values.
- A variable may take different values at different times during the execution.
- Variable name may declare based on the meaning of the operation.

Rules for naming a variable:

- A variable name can be any combination of alphabets, digits and underscore.
- Variable should not start with a digit.
- The first character must be an alphabet or an underscore (_).
- No commas or blank space are allowed within a variable name.
- No special symbol can be used in a variable.

Example:

Valid Variable Names:

Rollno, _Rollno, Rolln0123
name, _name, name0

Invalid Variable Names

-Rollno, Roll No, 123RollNo
+name, name-, 7name

Variable Declaration

- The variables must be declared before they are used in the program.
- Declaration provides two things
 - (i) compiler obtain the variable name
 - (ii) It tells the compiler the data type of the variable.

Syntax:

```
Data_ type  variableName;  
Data_type   - Valid C datatypes  
variableName – valid variable name
```

Example: int age;

Variable initializing

- Initialization of variable can be done using the assignment operator(=).
- The variable can be initialized when it is declared.

Syntax: variableName=constant or data_type variable Name=constant

Example: int a ; a=2; int b=10;

Storage classes in C

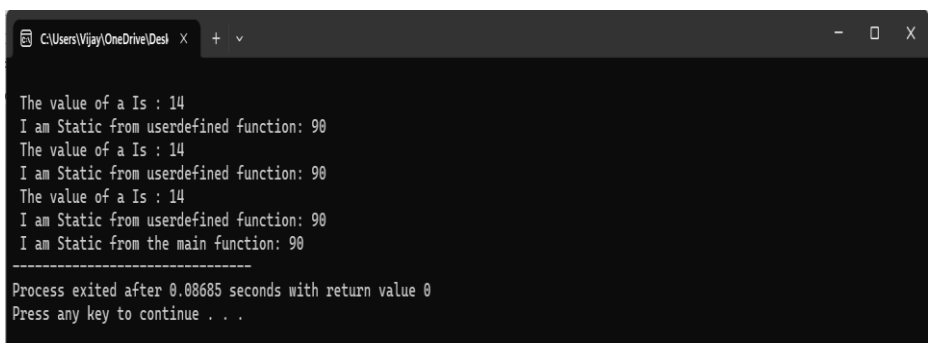
Storage classes in C programming determine the lifetime, scope, and visibility of variables within a program. These classes define where and how a variable is stored in memory during program execution. C provides four primary storage classes: auto, register, static, and extern. Understanding these storage classes is essential for efficient memory management and variable usage in C programs.

```
#include<stdio.h>

int main()
{
    int fun();
    fun();
    fun();
    fun();
    return 0;
}

int fun()
{
    auto int a=13;
    a++;
    printf("\n The value of a Is : %d",a);
    return 0;
}
```

Output



```
C:\Users\Vijay\OneDrive\Desktop >
The value of a Is : 14
I am Static from userdefined function: 90
The value of a Is : 14
I am Static from userdefined function: 90
The value of a Is : 14
I am Static from userdefined function: 90
I am Static from the main function: 90
-----
Process exited after 0.00685 seconds with return value 0
Press any key to continue . . .
```

register Storage Class:

- Variables declared with the register storage class are stored in CPU registers for faster access.
- They are used for frequently accessed variables.

Example:

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

```
int main()
{
    // works (inside a block)
    register int i = 10;
    int x=160;
    printf("%d\n", i);
    printf("%d", x);
    return 0;
}
```

Advantages of the register storage

- The register storage class in C is used to suggest to the compiler that a variable should be stored in a CPU register for faster access. While the use of the register keyword is a hint to the compiler and doesn't guarantee that a variable will be placed in a register (modern compilers are highly optimized and make their own decisions), there are advantages to using register storage when appropriate:
- **Faster Access:** Variables stored in CPU registers can be accessed much faster than those stored in memory. This is because registers are part of the CPU, and accessing them doesn't involve the latency associated with fetching data from RAM.
- 1. **Reduced Memory Access:** Accessing data from registers eliminates the need to read from or write to memory, which can significantly improve program performance, especially in tight loops or frequently used variables.
- 2. **Optimized Code:** Using register can lead to more optimized code generated by the compiler. It may choose to perform certain operations using the register variables more efficiently than if they were stored in memory.
- 3. **Critical Variables:** register is typically used for critical variables that are accessed frequently within a small scope, such as loop counters or frequently used pointers. By hinting to the compiler that these variables should be in registers, you can potentially speed up critical sections of code.

static Storage Class

Variables declared with the static storage class have a lifetime throughout the program execution.

They retain their values between function calls and persist in memory.

Example

```
include<stdio.h>
```

```
static int t=90;
```

```
int main()
```

```
{
```

```
    int fun();
```

```
    fun();
```

```
    fun();
```

```
    fun();
```

```
    printf("\n I am Static from the main function: %d",t);
```

```
    return 0;
```

```
}
```

```
int fun()
```

```
{
```

```
    auto int a=13;
```

```
    a++;
```

```
    printf("\n The value of a Is : %d",a);
```

```
    printf("\n I am Static from userdefined function: %d",t);
```

```
    return 0;
```

```
}
```

Output

```
C:\Users\Vijay\OneDrive\Desktop x + -
The value of a Is : 14
I am Static from userdefined function: 90
The value of a Is : 14
I am Static from userdefined function: 90
The value of a Is : 14
I am Static from userdefined function: 90
I am Static from the main function: 90
-----
Process exited after 0.05005 seconds with return value 0
Press any key to continue . . . |
```

Advantages of the static storage

In C++ programming, using static storage for variables provides several advantages, depending on the context and the specific use case. Here are some of the key advantages of using static storage:

- **Persistent Data Storage:** Variables with static storage duration persist throughout the entire program's execution. They are initialized only once, typically to zero if not explicitly initialized, and retain their values between function calls. This makes them suitable for maintaining state or data that needs to persist across function invocations.
- **Initialization Control:** You have precise control over when static variables are initialized. They are initialized at program startup, before the main function is called, ensuring that they have known initial values.
- **Efficient Memory Management:** Static variables are allocated memory in a special area of memory known as the "data segment" or "BSS segment." This allocation is typically more efficient in terms of memory usage and access time compared to dynamically allocated memory (e.g., using malloc) or automatic variables (e.g., those declared within functions).
- **Global Accessibility:** Static variables can be accessed from any function within the same translation unit (source file), making them suitable for sharing data across multiple functions in the same file without exposing the data to external code.
- **Reduced Function Call Overhead:** When used within functions, static variables can help reduce function call overhead. For example, you can use them to cache values that are expensive to compute, avoiding redundant computations across multiple calls.

Extern Storage Class:

- The extern storage class is used to declare variables that are defined in other source files.
- It provides visibility to variables declared in other files.

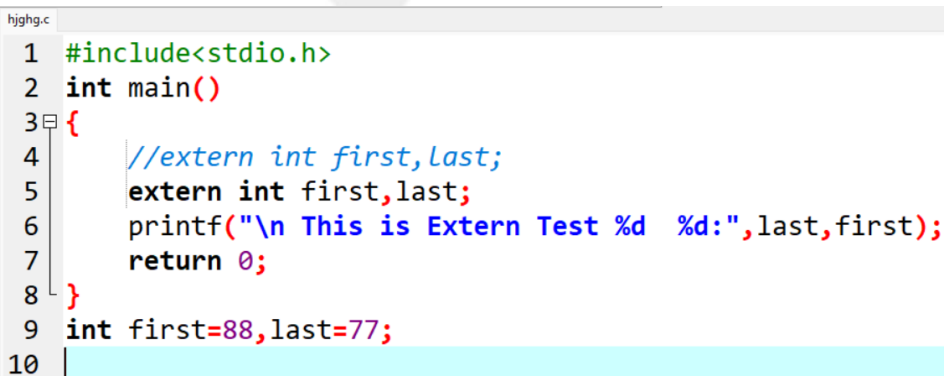
Example

```
#include<stdio.h>

int main()
{
    //extern int first,last;
    extern int first,last;
    printf("\n This is Extern Test %d  %d:",last,first);
    return 0;
}

int first=88,last=77;
```

Output on Dev C++



```
hghg.c
1  #include<stdio.h>
2  int main()
3  {
4      //extern int first,last;
5      extern int first,last;
6      printf("\n This is Extern Test %d  %d:",last,first);
7      return 0;
8  }
9  int first=88,last=77;
10
```

This is Extern Test 77 88:

Process exited after 0.04365 seconds with return value 0
Press any key to continue . . .

Features comparison among the storage classes

Feature	static	auto	extern	register
Storage	Memory	Memory	Memory	CPU register
Default value	Zero	Garbage Value	Zero	Garbage Value
scope	Local	Local	Global	Local
Life time	Static	automatic	static	automatic

Basic I/O Statements

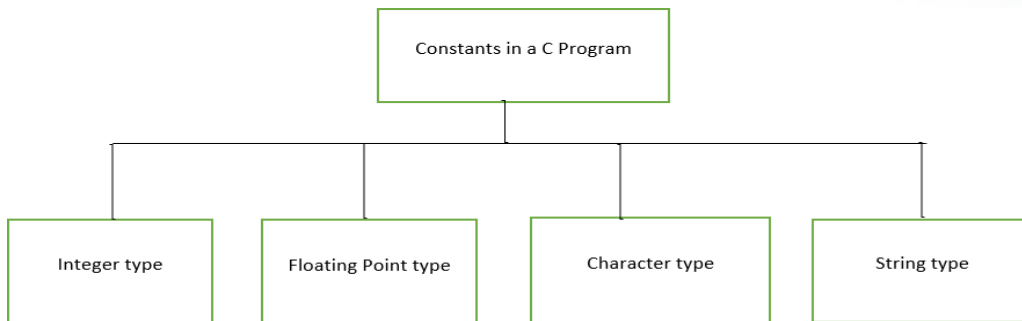
Formatted Input Function

scanf()

- scanf() is used to read all types of data values.
- The general syntax is :
- scanf("Conversion symbol", &variable name);
- Example: scanf("%d", &a);
- scanf() statement reads all types of data values.
- It is used for runtime assignment of variables.
- Scanf() requires conversion symbol to identify the data to be read during the execution of a program.
- Conversion symbol represents the field format in which the data is to be entered.
- Variables should be separated by comma.
- Conversion symbol contains the conversion character %, a data type character and an optional number specifying the field width.
- & operator specifies the memory location to store the variable.
- The data type character indicates the type of data assigned to the variable.
- The format specifiers or conversion symbols used in scanf ()are:

1.11 Constants

Constants are identifiers whose value does not change.



Integer type Constant

A constant of integer type consists of a sequence of digits.

Example:

1,34,546,8909 etc. are valid integer constants.

Floating point constant

Integer numbers are inadequate to express numbers that have a fractional point. A floating point constant therefore consists of an integer part, a decimal point, a fractional part, and an exponent field containing an e or E (e means exponents) followed by an integer where the fraction part and integer part are a sequence of digits.

Example:

Floating point numbers are 0.02, -0.23, 123.345, +0.34 etc.

Character Constant

A character constant consists of a single character enclosed in single quotes. For example, 'a', '@' are character constants. In computers, characters are stored using machine character set using ASCII codes.

String Constant

A string constant is a sequence of characters enclosed in double quotes. So "a" is not the same as 'a'. The characters comprising the string constant are stored in successive memory locations. When a string constant is encountered in a C program, the compiler records the address of the first character and appends a null character ('\0') to the string to mark the end of the string.

Declaring constant

```
#define PI 3.14159
#define service_tax
    0.15
#define MAX 10
```

FORMATTED OUTPUT FUNCTION

printf()

- It specifies print formatted and prints all types of data values
- It requires conversion symbol and variable names to print the data
- The conversion symbol and variable names should be same in number.
- The syntax is printf("Conversion symbol", variable name);
- Format specifiers or conversion symbol define the format of the values to be displayed
 - Escape sequence characters like \n, \t etc, can be used in printf() function.

Example: printf("%d", a);

Conversion Symbol	Description
%d	integer
%c	Character
%f	float
%lf	Double
%s	string

UNFORMATTED INPUT FUNCTIONS:

getchar()

This function gets a single character input from the standard input

Syntax :

```
char c;  
c=getchar();
```

getc()

This function gets a single character from the keyboard or from a file.

getch()

This function gets a character from keyboard and doesnot echo it on the screen.

gets()

This function gets a string input from the keyboard.

UNFORMATTED OUTPUT FUNCTION

putchar()

This function outputs a single character on the standard output.

Syntax:

```
    putchar( c ); // c-character variable
```

putc()

This function outputs a single character to monitor or to a file.

putch()

This function outputs a single character on the screen.

puts()

This function outputs a string on the standard output (Monitor).

C program for getchar() & putchar()

```
#include<stdio.h>

void main()
{
    char c;
    printf("Enter the grade:");
    c=getchar();
    printf("The grade is:");
    putchar();
}
```

C program for gets() & puts()

```
#include<stdio.h>

void main()
{
char name[30];
printf("Enter your name:");
Gets(name);
printf("\n your name is..");
puts(name);
}
```



R.M.K.
GROUP OF
INSTITUTIONS

1.12 Operators

An operator is a symbol that specifies an operation to be performed on the operands. An operand is a data item on which operators perform the operations.

Types

C provides a rich set of operators to manipulate data.

1. Arithmetic Operators
2. Unary Operator
3. Relational Operators
4. Logical Operators
5. Assignment Operator
6. Bitwise Operators
7. Conditional Operators

1. Arithmetic Operators

The following table list arithmetic operators

Operator	Description	Example
+	Addition	A + B
-	Subtraction	A – B
*	Multiplication	A * B
/	Division	A/B
%	Modulus	A%B

```
/* Example */
#include<stdio.h>
void main()
{
int a = 10, b=3;
printf("a + b = ", (a + b) );
printf("a - b = ",(a - b) );
printf("a * b = ",(a * b) );
printf("a / b = ",(a / b) );
printf("a % b = ",(a % b) );
}
```

Output:

$a + b = 13$ $a - b = 7$

$a * b = 30$ $a / b = 3$ a

$a \% b = 1$

2. Unary Operators

The following are the unary operators

Operator	Description	Example
+	Unary plus operator	+A
-	Unary minus operator	-A
++	Increment operator	++A or A++
--	Decrement operator	--A or A--

++ and -- works in two different modes

- Pre increment/decrement – When it is part of a statement, increment/decrement gets evaluated first, followed by the execution of the statement.
- Post increment/decrement – When the operator is part of a statement, the statement gets processed first, followed by increment/decrement operation.

// Example for pre increment

increment/decrement

#include<stdio.h>

#include<conio.h>

void main()

{

int a = 10, b=3;

printf("a++ = ", (a ++));

}

```
printf("a - - = " , (a - -) );
}
```

Output:

a++ = 11

b-- = 2

3. Relational Operators

Relational operators are used to test condition and results true or false value, The following table lists relational operators

Operator	Description	Example
==	Two values are checked, and if equal, then the condition becomes true	(A == B)
!=	Two values are checked to determine whether they are equal or not, and if not equal, then the condition becomes true	(A != B)
>	Two values are checked and if the value on the left is greater than the value on the right, then the condition becomes true.	(A > B)
<	Two values are checked and if the value on the left is less than the value on the right, then the condition becomes true	(A < B)
>=	Two values are checked and if the value on the left is greater than equal to the value on the right, then the condition becomes true	(A >= B)
<=	Two values are checked and if the value on the left is less than equal to the value on the right, then the condition becomes true	(A <= B)

```
/* Example to understand Relational operator */ #include<stdio.h>
#include<conio.h> void main()
{
int a = 10, b=20; printf("a= = b=", (a ==b) );
printf("a !=b= " , (a!=b) );
printf("a>b=", (a>b));
printf("a>=b=", (a>=b));
printf("a<b=", (a<b));
printf("a<=b=", (a<=b))
}
```

Output:

a == b = false

a != b = true

a > b = false

a < b = true

b >= a = true

b <= a = false

4. Logical Operators

- Operators which are used to combine two or more relational operators are called as logical operators.
- These operators are used to test more than one condition at a time.

&&	Logical AND
	Logical OR
!	Logical NOT

Truth Table

A	B	A&&B	A B	!A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Example:

A	B	Expression	Result
10	3	(A>B) && (B>5)	0 (false)
10	5	(A!=B) (B<10)	1 (true)
10	5	!A (A==B)	0 (false)

5. Conditional or Ternary Operators

It contains a conditions followed by two statements. A ternary operator pair " ? : " is available in C Syntax: Condition? exp1 :exp2;

Example

$x = (a > b) ? a : b$

exp1 is evaluated first if it is true ,or else expression exp2 is evaluated.

6. Assignment Operators

Simple Assignment

=, assigns right hand side value to left hand side variable. Eg. `int a; a=10;`

Compound Assignment

`+=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=`, assigns right hand side value after the computation to left hand side variable

Example:

`int a; int b;`

`a += 10; // means a = a + 10;`

`a &= b; // means a = a & b;`

7. Bitwise Operators

Bitwise operator act on integral operands and perform binary operations. The lists of bitwise operators are

Bitwise AND	&
Bitwise OR	
Bitwise EXOR	^
Bitwise NOT	~ (unary operator)
Shift Left	<<
Shift Right	>>

PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

- An expression is a combination of constants, variables and operators.
- Consider the expression, $a + b * c$, has higher precedence.
- Hence $(b*c)$ will be evaluated first. This result will then be added to a .
- In order to override the precedence, parentheses can be used.
- **Associativity** refers to the order in which the operators with the same precedence are evaluated in expression.
- It is also called as **grouping**.
- The operators in an expression are evaluated either from left-to-right or right-to-left.
- In case of nested parentheses, the innermost parentheses are evaluated first.

Priority	Operators	Description
1 st	* / %	multiplication, division, modular division
2 nd	+ -	addition, subtraction
3 rd	=	assignment

Example: $x = 9/2 + a-b;$

Precedence of Operators

Operator	Description	Precedence level	Associativity
() [] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Dot operator (Member selection via object name) Arrow operator (Member selection via pointer) Postfix increment/decrement	1	Left to Right
+ - ++ -- ! ~ * & (datatype) sizeof	Unary plus Unary minus Prefix increment/decrement Logical NOT One's complement Indirection Address (of operand) Type cast Determine size in bytes on this implementation	2	Right to Left
* / %	Multiplication Division Modulus	3	Left to Right
+ -	Addition Subtraction	4	Left to Right
<< >>	Left shift Right shift	5	Left to Right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	Left to Right
== !=	Equal to Not equal to	7	Left to Right
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
	Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
	Logical OR	12	Left to Right
?:	Conditional operator	13	Right to Left
= *= /= %= += -= &= ^= = <<= >>=	Assignment operators	14	Right to Left
,	Comma operator	15	Left to Right

1.13 Expressions

- An **expression** is a formula in which operands are linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.

Evaluate the following Expressions.

i) $4+8/2*7+4$

$$=4+4*7+4$$

$$=4+28+4$$

$$=32+4$$

$$=36$$

ii) $4\%3*5/2+(5*5)$

$$=4\%3*5/2+25$$

$$=1*5/2+25$$

$$=5/2+25$$

$$=2.5+25$$

$$=27.5$$

1.14 Statements

An expression such as $x=0$ or $i++$ or `printf(...)` becomes a statement when it is followed by semicolon.

Example

```
x=10;  
i++;  
printf(.....);
```

Braces { and } are used to group declarations and statements or block, so that they are syntactically equivalent to a single statement.

A block statement is generally used to group together several statements.

Example

Statement blocks

```
if( i == j)  
{  
printf("C Programming \n");  
}
```

The *statement block* containing the **printf** is only executed if the **i == j expression** evaluates to True.

Decision Making statements in C

The decision making statement checks the given condition and then executes its sub block.

The decision statement decides the statement to be executed after the success or failure of a given condition.

Decision making statements are classified into (i) if statements (ii) switch statement

If Statements :

Simple if statement

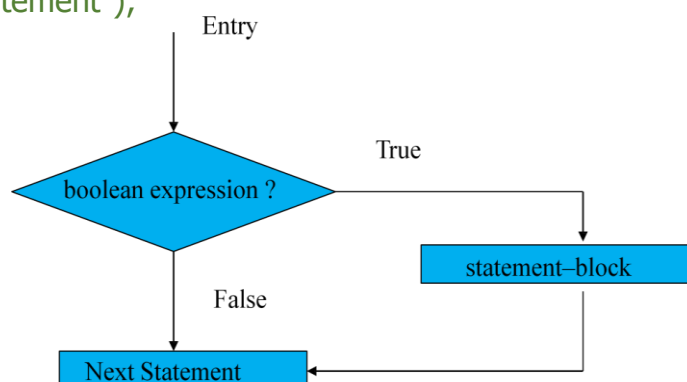
The if statement evaluates the test expression inside the parenthesis(). If the test expression is evaluated to true, statements inside the body of if are executed. If the test expression is evaluated to false, statements inside the body of if are not executed.

syntax :

```
if(Boolean  
expression)  
{  
statement-block;  
}  
Next statement;
```

//Example program

```
#include<stdio.h>  
void main()  
{  
int n=5;  
if(n<25)  
{  
printf("This is if statement");  
}  
}
```



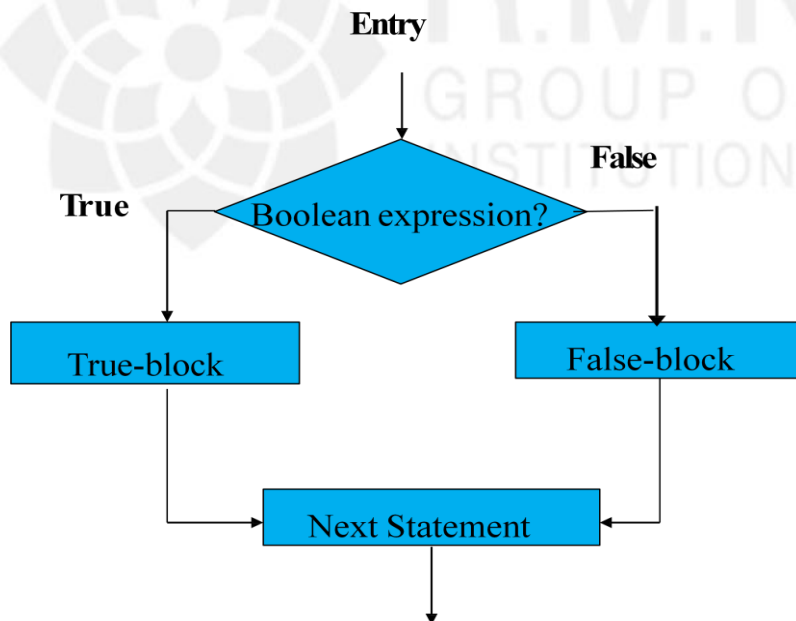
if .. else statement

If the test condition is evaluated to true, statements inside the body of if are executed.

If the test condition is evaluated to false, statements inside the body of else are executed.

Syntax

```
if(boolean expression)
{
  True-block statements;
}
else
{
  False-block statements;
}
Next statement;
```



Example program for if-else statement

```
#include<stdio.h>
void main()
{
    int age;
    printf("Enter the age");
    scanf("%d",&age);
    if(age>18)
    {
        printf("Eligible to vote");
    }
    else
    {
        printf("Not eligible to vote");
    }
}
```

Nested if.. else

In if-else statement, else block is executed by default after failure of condition, in order to execute the else block depending upon certain condition, if statement is repetitively added in else block. It is termed as Nested if-else statement.

A nested if-else can be chained with one another.

Syntax

```
if( condition )
    if(condition) statement;
    else statement;
if( condition )
    statement;
else
    if(condition)
        statement;
```

//program to find largest three number

```
#include<stdio.h>
void main()
{
    int n1,n2,n3;
    printf("Enter the number");
    scanf("%d%d%d",&n1,&n2,&n3);
    if(n1>n2 && n1>n3)
    {
        printf("%d is largest number",n1);
    }
    else
    {
        If(n2>n3)
            printf("%d is the largest
number",n2);
        else
            printf("%d is the largest
number",n3);
    }
}
```

else if ladder:

When a series of many conditions have to be checked use the else if ladder statement, which takes the following general form.

```
if (condition1)
    statement1;

else if (condition2)
    statement2;

else if (condition3)
    statement3;

else if (condition)
    statement n;

else
    default statement;
statement x;
```


Example:

```
#include<stdio.h>

void main()
{
    int m1;
    printf("enter the m1:");
    scanf("%d",&m1);
    if(m1<50)
        printf("m1 is fail mark");
    else if(m1>50 && m1<60)
        printf("m1 is average mark");
    else if(m1>60 && m1<75)
        printf("m1 is firstclass mark");
    else
        printf("m1 is high mark");
}
```

Switch Statement

- It is a multi-way branch statement.
- It requires only one argument of any data type, which is checked with number of case option.
- switch() statement evaluates expression and then for values among the case constants.
- If the value matches with case constant, the particular case statement is executed, if not default is executed.
- switch, case and default are reserved keywords.
- Every case statement terminates with ': '.
- The break statement is used to exit from the current case structure.

Syntax:

```
switch (expression)
{
    case value-1;
        block-1;
        break;
    case value-2;
        block-2;
        break;
    default:
        default-block;
} Statement- x
```

- The expression is an integer expression or character.
- value-1, value-2 is constants or constant expressions and is known as case labels.
- These values should be unique within a switch statement.
- block-1, block-2... are statement lists.

When the switch is executed, the value of the expression is successively compared against the values value-1, value-2...

If a case is found, whose value matches with the value of the expression, and then the block of statements that follows the case is executed.

The break statement, transfers the control to the statement-x following the switch. Default is an optional case.

example of a switch case statement

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

```
void main()
```

```
{
```

```
int w;
```

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

```
switch(w)
```

```
{
```

```
case 1:    printf("Sunday");  
          break;
```

```
case 2:  
          printf("Monday");  
          break;
```

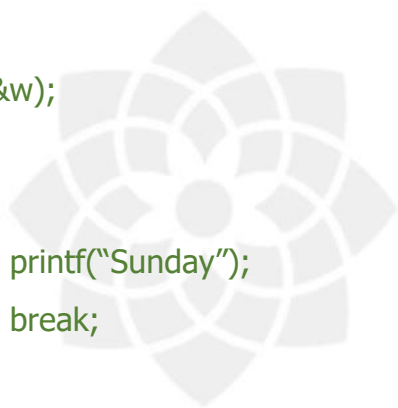
```
case 3:  
          printf("Tuesday");  
          break;
```

```
case 4:  
          printf("Wednesday");  
          break;
```

```
case 5:    printf("Thursday");  
          break;
```

```
case 6:    printf("Friday");  
          break;
```

```
}      }
```



R.M.K.
GROUP OF
INSTITUTIONS

Looping Statements

Loop: It is defined as a block of statements which are repeatedly executed for certain number of times.

Three looping constructs are:

➤ **while statement**

➤ **do-while statement**

➤ **for statement**

Steps of a Looping Process:

Loop Variable: It is a variable used in the loop.

Initialization: Assigning starting and final value to the loop variable.

Increment/Decrement: Numerical value added or subtracted from the variable in each round of loop.

while loop

It's a **entry controlled loop**, the condition in the while loop is evaluated, and if the condition is true, the code within the block is executed. This repeats until the condition becomes false

The loop control variable should be initialized outside the loop and incremented / decremented inside the loop.

Syntax

```
while(condition)
```

```
{
```

```
    Body of the loop
```

```
}
```

//Example program for while loop

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

```
void main()
```

```
{
```

```
    int i = 0;
```

```
    while (i < 5)
```

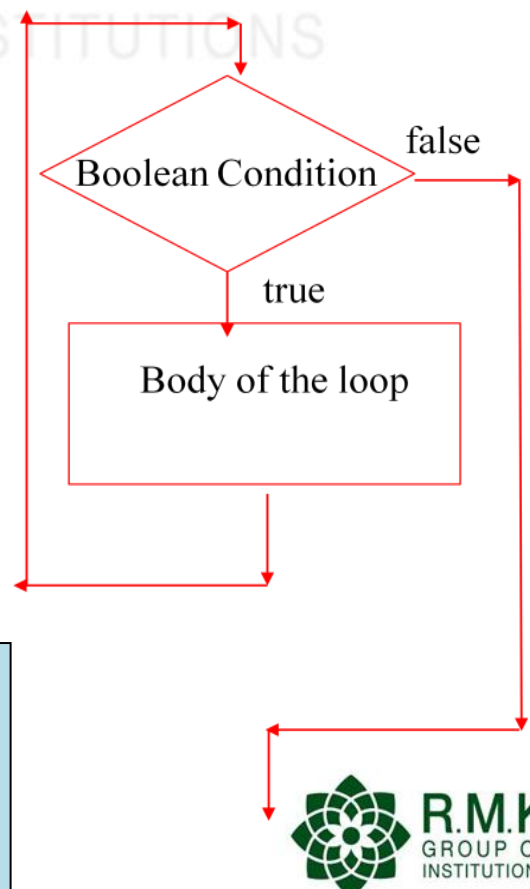
```
    {
```

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

```
        i = i + 1;
```

```
    }
```

```
}
```



Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

do.. while Loop

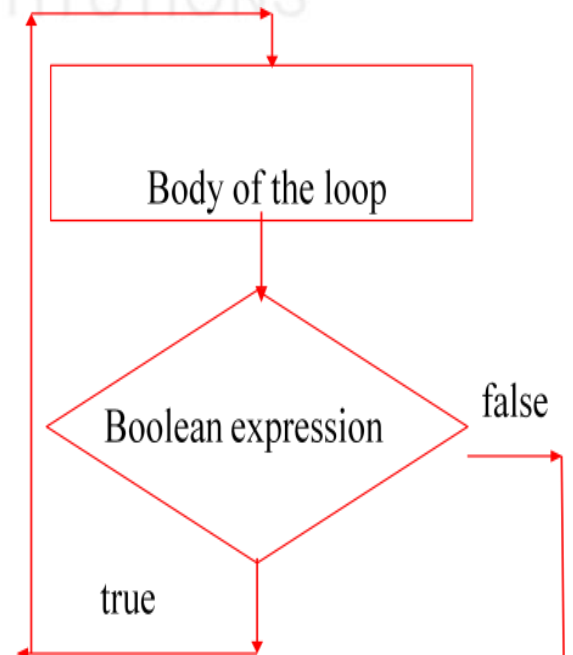
- It is an exit checked loop.
- The body of the loop is executed at least once even when the condition becomes false.
- The loop will be executed as long as the condition is true.
- When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

Syntax

```
do
{
    body of the loop
}
while(Boolean expression);
```

/* This is an example of a do-while loop */

```
#include<stdio.h>
void main()
{
    int i=5;
    do
    {
        printf("i:%d",i);
        i = i + 1;
    }
    while (i < 5);
}
```



for Loop

The for loop initialize the value before the first step. Then checking the condition against the current value of variable and execute the loop statement and then perform the step taken for each execution of loop body. For-loops are also typically used when the number of iterations is known before entering the loop.

Syntax

```
for(initialization;condition; increment/decrement)
```

```
{
```

Body of the loop

```
}
```

```
/* This is an example of a for loop */
```

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

```
void main()
```

```
{
```

```
    int i;
```

```
    for(i=0;i<=5;i++)
```

```
    {
```

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

```
    }
```

```
}
```

Output:

```
i: 1
```

```
i: 2
```

```
i: 3
```

```
i: 4
```

```
i: 5
```

Break statement

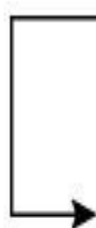
A break statement terminates the loop, and the control then automatically passes to the first statement after the loop.

Break can be associated with all conditional statements.

Syntax


break;

```
while (test expression) {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statement/s  
}
```




The flowchart shows a loop structure. It starts with a 'while (test expression)' block. Inside, there is a 'statement/s' block, followed by an 'if (test expression)' block containing a 'break;' statement. An arrow from the 'break;' statement points to the closing brace of the while loop, indicating an exit from the loop.

```
do {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statement/s  
} while (test expression);
```



The flowchart shows a do-while loop structure. It starts with a 'do' block containing 'statement/s', an 'if (test expression)' block with a 'break;' statement, and another 'statement/s' block. An arrow from the 'break;' statement points to the closing brace of the do block, indicating an exit from the loop.

```
for (initial expression; test expression; update expression) {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statements/  
}
```



The flowchart shows a for loop structure. It starts with a 'for (initial expression; test expression; update expression)' block. Inside, there is a 'statement/s' block, followed by an 'if (test expression)' block containing a 'break;' statement. An arrow from the 'break;' statement points to the closing brace of the for loop, indicating an exit from the loop.

NOTE: The break statement may also be used inside body of else statement.

Program to demonstrate the working of break statement in C programming

```
main( )
{
    int i;

    for(i=1;i<10;i++)
    {
        if (i==5)
        {
            break; // breaks out of the for loop
        }
        printf(" Value of i = %d\n",i);
    }
}
```

Output

Value of i = 1
Value of i = 2
Value of i = 3
Value of i = 4

The Continue Statement

It is used for continuing next iteration of loop statements.

When it occurs in the loop, it does not terminate but skips the statements after this statement.

Program to demonstrate the working of continue statement in C programming

```
# include <stdio.h>
main( )
{
    int i;

    for(i=1;i<10;i++)
    {
        if (i% 4==0)
        {
            continue; // when i is divisible by 4 continue to next iteration
        }
        printf(" %d\n",i);
    }
}
```

```

→ while (test expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}

```

```

do {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
→ while (test expression);

```

```

→ for (initial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statements/
}

```

NOTE: The continue statement may also be used inside body of else statement.

goto Statement:

It is an unconditional branching statement.

This statement does not require any condition.

goto statement passes control anywhere in the program, ie., the control is transferred to another part of the program without testing any condition.

Syntax:

goto Label;

-

-

Label: Statement;

Label: may be anywhere in the program.

//Example:

```
main()
{
    int age;
    Vote:
        printf("you are eligible for voting");

    NoVote:
        printf("you are not eligible to vote");

    printf("Enter you age:");
    scanf("%d", &age);
    if(age>=18)
        goto Vote;
    else
        goto NoVote;
}
```

Explanation: In the above example, Vote and NoVote are labels. When the input is ≥ 18 , the goto statement is transferring the control to label – Vote, otherwise it transfers the control to label-NoVote.

1.15 Arrays and String

Arrays – Introduction, Declaration, Initialization, Input, Output Introduction

An array is a group (or collection) of same data types. For example an int array holds the elements of int types while a float array holds the elements of float types.

Consider a scenario where you need to find out the average of 100 integer numbers entered by user. There are two ways to do this:

- 1) Define 100 variables with int data type and then perform 100 scanf() operations to store the entered values in the variables and then at last calculate the average of them.
- 2) Have a single integer array to store all the values, loop the array to store all the entered values in array and later calculate the average.

Which solution is better? Obviously the second solution, it is convenient to store same data types in one single variable and later access them using array index.

1.16 Single Dimensional Array :

datatype arrayname[size];

Arrays should also be declared before they are used in the program. The general syntax for declaring array is:

int a[5]; //under the name a - 5 integers

26	89	15	39	57
a[0]	a[1]	a[2]	a[3]	a[4]

Array subscript (or index) is used to access any element stored in array. Subscript starts with 0, which means a[0] represents the first element in the array a.

In general a[n-1] can be used to access nth element of an array. where n is any integer number.

If the number of values provided is less than the number of Elements in the array, the un-assigned elements are filled with zeros.

Types of array :

- 1. one dimensional array

Syntax : datatype arrayname[size]; eg: int a[10];

- 2. Two dimensional array - to store the matrix inputs.(table of values having rows and columns)

syntax :

datatype arrayname[size1][size2];

int a[2][3]; - 6 nos - 2 rows & 3 columns

- 3.Multi dimensional array :

`datatype arrayname[size1][size2].....[sizen];`

Input data into the Single dimensional array

Here we are **iterating the array** from 0 to 3 because the size of the array is 4. Inside the loop we are displaying a message to the user to enter the values. All the input values are stored in the corresponding array elements using `scanf` function.

```
for (i=0; i<4;i++)  
{  
    scanf("%d", &num[i]);  
}
```

Reading out data from Single dimensional array

Suppose, if we want to display the elements of the array then we can use the `for` loop like this.

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

Single Dimensional Array:

Example Program:

```
#include<stdio.h>  
  
void main ( )  
{  
    int a[5],i;  
    printf ("enter 5 elements");  
    for ( i=0; i<5; i++)  
        scanf("%d", &a[i]);  
    printf("elements of the array are");  
    for (i=0; i<5; i++)  
        printf("%d", a[i]);  
}
```

Output

The output is as follows –

```
enter 5 elements          10 20 30 40 50  
elements of the array are : 10 20 30 40 50
```

Various ways to initialize a single dimensional array

In the above example, we have just declared the array and later we initialized it with the values input by user. However you can also initialize the array during declaration like this:

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

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

Un-initialized array always contain garbage values.

C Array – Memory representation

Array Index:	a[0]	a[1]	a[2]	a[3]	a[4]
Array Elements:	10	20	30	40	50
Memory Location:	1000	1002	1004	1006	1008

All the array elements occupy contiguous space in memory. There is a difference of 2 among the addresses of subsequent neighbours, this is because this array is of integer type and an integer holds 2 bytes of memory.

1.17 Two Dimensional Arrays (2D Arrays)

2D array – We can have multidimensional arrays like 2D and 3D array. However the most popular and frequently used array is 2D – two dimensional array. We will see how to declare, read and write data in 2D array along with various other features of it.

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {  
    {10, 11, 12, 13},  
    {14, 15, 16, 17}  
};
```

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Although both the above declarations are valid, the first method is more readable, because you can visualize the rows and columns of 2d array in this method.

We know that, when we initialize a normal array (one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration.

Example:

```
/* Valid declaration*/  
int abc[2][2] = {1, 2, 3 ,4 }
```

```
/* Valid declaration*/  
int abc[][2] = {1, 2, 3 ,4 }
```

```
/* Invalid declaration – you must specify second dimension*/  
int abc[][] = {1, 2, 3 ,4 }
```

```
/* Invalid declaration – you must specify second dimension*/  
int abc[2][] = {1, 2, 3 ,4 }
```

Two dimensional (2D) Array Example

This program demonstrates how to store the elements entered by user in a 2D array and how to display the elements of a two dimensional array.

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

```
int main()
```

```
{
```

```
    int disp[2][3];
```

```
    int i, j;
```

```
    for(i=0; i<2; i++)
```

```
    {
```

```
        for(j=0;j<3;j++)
```

```
        {
```

```
            printf("Enter value for disp[%d][%d]:", i, j);
```

```
            scanf("%d", &disp[i][j]);
```

```
        }
```

```
    }
```

```
    printf("Two Dimensional array elements:\n");
```

```
    for(i=0; i<2; i++)
```

```
    {
```

```
        for(j=0;j<3;j++)
```

```
        {
```

```
printf("%d ", disp[i][j]);  
    if(j==2)  
    {  
        printf("\n");  
    }  
}  
}  
return 0;  
}
```

Output:

```
Enter value for disp[0][0]:1  
Enter value for disp[0][1]:2  
Enter value for disp[0][2]:3  
Enter value for disp[1][0]:4  
Enter value for disp[1][1]:5  
Enter value for disp[1][2]:6
```

Two Dimensional array elements:

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

No of elements in a 2D Array

We can calculate how many elements a two dimensional array can have by using this formula:

The array `arr[n1][n2]` can have $n1*n2$ elements.

The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has first subscript value as 5 and second subscript value as 4. So the array `abc[5][4]` can have $5*4 = 20$ elements.

- **Matrix Operations (Addition, Subtraction)**

Matrix addition in C language to add two matrices, i.e., compute their sum and print it. A user inputs their orders (number of rows and columns) and the matrices. For example, if the order is 2, 2, i.e., two rows and two columns and

The matrices are:

First matrix:

1 2
3 4

Second matrix:

4 5
-1 5

The output is:

5 7
2 9

Matrix Addition Program:

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

```
int main()
```

```
{
```

```
int m, n, c, d, first[10][10], second[10][10], sum[10][10];
```

```
printf("Enter the number of rows and columns of matrix\n");
```

```
scanf("%d%d", &m, &n);
```

```
printf("Enter the elements of first matrix\n");
```

```
for (c = 0; c < m; c++)
```

```
for (d = 0; d < n; d++)
```

```
scanf("%d", &first[c][d]);
```

```
printf("Enter the elements of second matrix\n");
```

```
for (c = 0; c < m; c++)
```

```
for (d = 0 ; d < n; d++)
```

```
scanf("%d", &second[c][d]);
```

```
printf("Sum of entered matrices:-\n");
```

```
for (c = 0; c < m; c++) {
```

```
for (d = 0 ; d < n; d++) {
```

```
sum[c][d] = first[c][d] + second[c][d];
```

```
printf("%d\t", sum[c][d]);
```

```
} printf("\n");
```

```
}
```

```
return 0; }
```

Linear Search

Linear search is a very simple and basic search algorithm. Here we will implement a program that finds the position of an element in an array using a Linear Search Algorithm.

What is a Linear Search?

A linear search, also known as a sequential search, is a method of finding an element within a list. It checks each element of the list sequentially until a match is found or the whole list has been searched.

A simple approach to implement a linear search is

- Begin with the leftmost element of arr[] and one by one compare x with each element.
- If x matches with an element then return the index.
- If x does not match with any of the elements then return -1.

Implementing Linear Search in C

```
#include<stdio.h>

int main()
{
    int a[20],i,x,n;
    printf("How many elements?");
    scanf("%d",&n);
    printf("Enter array elements:n");
    for(i=0;i<n;++i)
        scanf("%d",&a[i]);
    printf("\nEnter element to search:");
    scanf("%d",&x);
    for(i=0;i<n;++i)
        if(a[i]==x)
            break;
    if(i<n)
        printf("Element found at index %d",i);
    else
        printf("Element not found");
    return 0;
}
```


Output:

How many elements ? 5

Enter Array Elements : 77 45 16 33 26

Enter Element to search : 16

Element found at index 2

The time required to search an element using a linear search algorithm depends on the size of the list. In the best-case scenario, the element is present at the beginning of the list and in the worst-case, it is present at the end.

The time complexity of a **linear search is $O(n)$** .

Binary Search

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example:

Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 nd half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 > 56 take 1 st half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

- We basically ignore half of the elements just after one comparison.
- Compare x with the middle element.

- If x matches with middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half sub-array after the mid element. So we recur for right half.
- Else (x is smaller) recur for the left half.

Binary Search Program:

```
#include <stdio.h>
#include<stdlib.h>
int main(void)
{
    int n;
    printf("Enter number of elements\n");
    scanf("%d",&n);
    int i,s,a[n],start=0,end=n-1,mid;
    printf("Enter %d elements one by one\n",n);
    printf("In ascending order\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter the element to be searched\n");
    scanf("%d",&s);
    while(start<=end)
    {
        mid=(start+end)/2;
        if(s==a[mid])
        {
            printf("%d present at position %d\n",s,mid+1);
            exit(0);
        }
        else if(s<a[mid]) end=mid-1;
        else start=mid+1;
    }
    printf("%d is not present in the array\n",s);
    return 0;
}
```

Output

Enter number of elements

5

Enter 5 elements one by one

In ascending order

88

99

111

222

555

Enter the element to be searched

222

222 is present at position 4



R.M.K.
GROUP OF
INSTITUTIONS

Matrix multiplication program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int a[10][10], b[10][10], mul[10][10], r, c, i, j, k;
printf("Enter the number of row = ");
scanf("%d",&r);
printf("Enter the number of column = ");
scanf("%d",&c);
printf("Enter the first matrix element = \n");
for(i=0; i<r; i++)
    for(j=0; j<c; j++)
        scanf("%d",&a[i][j]);
printf("Enter the second matrix element=\n");
for(i=0; i<r; i++)
    for(j=0; j<c; j++)
        scanf("%d",&b[i][j]);
printf("Multiplied - The matrix is: \n");
for(i=0; i<r; i++)
{
    for(j=0;j<c;j++)
    {
        mul[i][j]=0;
        for(k=0; k<c; k++)
            mul[i][j] += a[i][k] * b[k][j];
    }
}
for(i=0; i<r; i++)
{
    for(j=0; j<c; j++)
        printf("%d\t",mul[i][j]);

    printf("\n");
}
return 0;
}
```

1.18 Arrays of Strings – Character Array – Introduction, Declaration, Initialization, Input, Output:

String – Character Array – There is no separate data type for strings in C. The character array is considered as a String.

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type. A string is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

Example: The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

Declaring and Initializing a string variables

There are different ways to initialize a character array variable.

```
char name[16]="Students like C";  
char sname[6]={ 'A','s','h','o','k','\0'};
```

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

Some examples of illegal initialization of character array are,

```
char s1[3]="hello"; // Illegal  
char s2[5];  
s2="hello"; // Illegal
```

String Input and Output

Input function scanf() can be used with %s format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World" using scanf() function, it will only read Hello and terminate after encountering white spaces.

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

```
#include<string.h>
```

```
void main()
```

```
{  
    char str[20];  
    printf("Enter a string");  
    scanf("%[^\n]", &str); //scanning the whole string, including the white spaces  
    printf("%s", str);  
}
```

Another method to read character string with white spaces from terminal is by using the gets() function.

```
char text[20];  
gets(text);  
printf("%s", text);
```

- A string is a null-terminated character array. This means that after the last character, a null character ('\0') is stored to signify the end of the character array.
- For example:
char c[] = "c string"; When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

How to declare a string?

- Here's how you can declare strings:
- char str[size];
- char s[5];

Here, we have declared a string of 5 characters.

A string can be declared as a **character array** or with a **string pointer**.

How to initialize strings?

- You can initialize strings in a number of ways.
- char c[] = "abcd";
- char c[5] = {'a','b','c','d','\0'};
- **char *c="abcd";**

Let's take another example:

- char c[5] = "abcde";

Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters. This is bad and you should never do this.

Assigning Values to Strings

- Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,
- char c[100];
- c = "C programming"; // Error! array type is not assignable.

Note: Use the strcpy() function to copy the string instead.

<pre>char str[] = "HELLO";</pre>	<pre>char ch = 'H';</pre> <p>Here H is a character not a string. The character H requires only one memory location.</p>
<pre>char str[] = "H";</pre> <p>Here H is a string not a character. The string H requires two memory locations. One to store the character H and another to store the null character.</p>	<pre>char str[] = "";</pre> <p>Although C permits empty string, it does not allow an empty character.</p>

Difference between character storage and string storage

For example if we write,

```
char str[] = "HELLO";
```

We are declaring a character array with 5 characters namely, H, E, L, L and O. Besides, a null character ('\0') is stored at the end of the string. So, the internal representation of the string becomes- HELLO'\0'. Note that to store a string of length 5, we need 5 + 1 locations (1 extra for the null character).

The name of the character array (or the string) is a pointer to the beginning of the string.

<code>str[0]</code>	1000	H
<code>str[1]</code>	1001	E
<code>str[2]</code>	1002	L
<code>str[3]</code>	1003	L
<code>str[4]</code>	1004	O
<code>str[5]</code>	1005	\0

Memory representation of a character array

We can also declare a string with size much larger than the number of elements that are initialized.

For example, consider the statement below.

```
char str [10] = "HELLO";
```

In such cases, the compiler creates an array of size 10; stores "HELLO" in it and finally terminates the string with a null character. Rest of the elements in the array are automatically initialized to NULL.

Now consider the following statements:

```
char str[3]; str = "HELLO";
```

The above initialization statement is illegal in C and would generate a compile-time error because of two reasons. First, the array is initialized with more elements than it can store. Second, initialization cannot be separated from declaration.

Note: When allocating memory space for a string, reserve space to hold the null character also.

Let us try to print above mentioned string:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char str[10]={ 'H','E','L','L','O','\0'};
    printf("Greeting string message : %s", str);
    return 0;
}
```

Output :

Greeting string message : HELLO

Note: %s is used to print the string in C

WRITING STRINGS

The string can be displayed on screen using three ways

- **use printf() function**
- **using puts() function**
- **using putchar()function repeatedly**

The string can be displayed using printf() by writing

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


String functions

1) `strlen()` Function : `strlen()` function is used to find the length of a character string.

Example: `int n;`
 `char st[20] = "Bangalore";`
 `n = strlen(st);`

- This will return the length of the string 9 which is assigned to an integer variable `n`.

- Note that the null character `"\0"` available at the end of a string is not counted.

2) `strcpy()` Function : `strcpy()` function copies contents of one string into another string. Syntax for `strcpy` function is given below.

Syntax: `char * strcpy (char * destination, const char * source);`

Example:

`strcpy (str1, str2)` – It copies contents of `str2` into `str1`.

`strcpy (str2, str1)` – It copies contents of `str1` into `str2`.

If destination string length is less than source string, entire source string value won't be copied into destination string.

For example, consider destination string length is 20 and source string length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

Example: `char city[15];`
 `strcpy(city, "BANGALORE") ;`

This will assign the string `"BANGALORE"` to the character variable `city`.

3) `strcat()` Function : `strcat()` function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for `strcat()` function is given below.

Syntax: `char * strcat (char * destination, const char * source);`

Example:

`strcat (str2, str1);` - `str1` is concatenated at the end of `str2`.

`strcat (str1, str2);` - `str2` is concatenated at the end of `str1`.

- As you know, each string in C is ended up with null character (`'\0'`).
- In `strcat()` operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after `strcat()` operation.

Program: The following program is an example of strcat() function

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char source[ ] = " ftl" ;
    char target[ ]= " welcome to" ;
    printf ("\n Source string = %s", source ) ;
    printf ( "\n Target string = %s", target ) ;
    strcat ( target, source ) ;
    printf ( "\n Target string after strcat( ) = %s", target ) ;
}
```

Output:

Source string = ftl

Target string = welcome to

Target string after strcat() = welcome to ftl

4) Strncat() function : strncat() function in C language concatenates (appends) portion of one string at the end of another string.

Syntax: char * strncat (char * destination, const char * source, size_t num);

Example:

strncat (str2, str1, 3); – First 3 characters of str1 is concatenated at the end of str2.

strncat (str1, str2, 3); - First 3 characters of str2 is concatenated at the end of str1.

As you know, each string in C is ended up with null character ('\0').

In strncat() operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strncat() operation.

Program : The following program is an example of strncat() function

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char source[ ] = " ftl" ;
    char target[ ] = "welcome to" ;
    printf ( "\n Source string = %s", source ) ;
    printf ( "\n Target string = %s", target ) ;
    strncat ( target, source, 3 ) ;
    printf ( "\n Target string after strncat( ) = %s", target ) ;
}
```

Output :

Source string = ftl

Target string = welcome to

Target string after strncat()= welcome to ft

strcmp() Function : strcmp() function in C compares two given strings and returns zero if they are same. If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value.

Syntax: int strcmp (const char * str1, const char * str2);

strcmp() function is case sensitive. i.e., "A" and "a" are treated as different characters.

Example:

```
char city[20] = "Madras";
```

```
char town[20] = "Mangalore";
```

```
strcmp(city, town);
```

This will return an integer value "-10" which is the difference in the ASCII values of the first mismatching letters "D" and "N".

* Note that the integer value obtained as the difference may be assigned to an integer variable as follows:

```
int n;
```

```
n = strcmp(city, town);
```

6) strcmpi() function :

strcmpi() function in C is same as strcmp() function. But, strcmpi() function is not case sensitive. i.e., "A" and "a" are treated as same characters. Whereas, strcmp() function treats "A" and "a" as different characters.

- strcmpi() function is non standard function which may not available in standard library.
- Both functions compare two given strings and returns zero if they are same.
- If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value.

strcmp() function is case sensitive. i.e., "A" and "a" are treated as different characters.

Syntax: int strcmpi (const char * str1, const char * str2);

Example:

```
m=strcmpi(" DELHI ", " delhi "); • m = 0.
```

7) strlwr() function : strlwr() function converts a given string into lowercase.

Syntax: char *strlwr(char *string);

strlwr() function is non standard function which may not available in standard library in C.

Program :

In this program, string "MODIFY This String To LOwer" is converted into lower case using strlwr() function and result is displayed as "modify this string to lower".

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[ ] = "MODIFY This String To Lower";
    printf("%s\n", strlwr (str));
    return 0;
}
```

Output:

modify this string to lower

8)strupr() function :strupr() function converts a given string into uppercase.

Syntax : char *strupr(char *string);

strupr() function is non standard function which may not available in standard library in C.

Program : In this program, string "Modify This String To Upper" is converted into uppercase usingstrupr() function and result is displayed as "MODIFY THIS STRING TO UPPER".

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[ ] = "Modify This String To Upper";
    printf("%s\n",strupr(str));
    return 0;
}
```

Output:

MODIFY THIS STRING TO UPPER

9)strrev() function :strrev() function reverses a given string in C language.

Syntax: char *strrev(char *string);

strrev() function is non standard function which may not available in standard library in C.

Example:

char name[20]="ftl"; then

strrev(name)= ltf

Program:

In below program, string "Hello" is reversed using `strrev()` function and output is displayed as "olleH".

```
#include<stdio.h>
#include<string.h>
int main()
{
    char name[30] = "Hello";
    printf("String before strrev( ) : %s\n", name);
    printf("String after strrev( ) : %s", strrev(name));
    return 0;
}
```

Output:

String before `strrev()` : Hello

String after `strrev()` : olleH

10) strchr() function : strchr() function returns pointer to the first occurrence of the character in a given string.

Syntax: `char *strchr(const char *str, int character);`

Program:

In this program, `strchr()` function is used to locate first occurrence of the character 'i' in the string "This is a string ". Character 'i' is located at position 3 and pointer is returned at first occurrence of the character 'i'.

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char string[25] ="This is a string ";
    char *p;
    p = strchr (string,'i');
    printf ("Character i is found at position %d\n",p-string+1);
    printf ("First occurrence of character \"%i\" in \"%s\" is \" \"%s\"",string, p);
    return 0;
}
```

Output:

Character i is found at position 3

First occurrence of character "i" in "This is a string" is "is is a string"

11) strstr() function : strstr() function returns pointer to the first occurrence of the string in a given string.

Syntax: char *strstr(const char *str1, const char *str2);

Program: In this program, strstr() function is used to locate first occurrence of the string "test" in the string "This is a test string for testing". Pointer is returned at first occurrence of the string "test".

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char string[55] ="This is a test string for testing";
    char *p;
    p = strstr (string, "test");
    if(p)
    {
        printf("string found\n" );
        printf ("First occurrence of string \"test\" in \"%s\" is\" \" \"%s\"",string, p);
    }
    else
        printf("string not found\n" );
    return 0;
}
```

Output:

String found

First occurrence of "test" in "this is a test string for testing" is "testing string for testing"

12) atoi() function : It converts string-value to numeric-value and it converts a numeric-string value to equivalent integer-value.

Syntax: int atoi(string);

Example:

```
printf("output=%d", atoi("123")+atoi("234"));
```

This printf() will print 357

13) atol() function : converts a long int string value to equivalent long integer value.

Syntax: long int atol(string);

Example:

```
printf("output=%d", atol("486384")-atol("112233"));
```

This statement will print 374151

14) atof() function : converts a floating point text format value to double value.

Syntax: int atoi(string);

Example:

```
printf("%.1f",atof("3.1412")*5*5);
```

This statement will print 78.530000

15) itoa(),ltoa(),ultoa() : These functions converts a given number(int/long int/unsigned long int) to equivalent text format based on the given numbering system radix value.

These functions take three arguments, the numeric value, target string address in which the value to be stored and radix value. Finally returns the target string address, so that function-call can be used as argument/expression.

Syntax: char* itoa(int value, char *targetstringaddress, int radix);

Example:

```
Char temp[50];
```

```
Printf("output=%s", itoa(45,temp,2));
```

Output : 101101

Program: Write a C program to count the number of vowels present in a sentence.

```
# include<stdio.h>
# include<conio.h>
# include<string.h>
main( )
{
    char st[80], ch;
    int count = 0, i;
    clrscr( );
    printf("\n Enter the sentence: \n");
    gets(st);
    for( i=0; i<strlen(st); i++)
        switch(st [i ])
        {
            case 'A':
            case 'E':
            case 'I':
            case 'O':
            case 'U':
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                count ++;

            break;
        }
    printf("\n %d vowels are present in the sentence", count);
    getch( );
}
```



R.M.K
GROUP OF
INSTITUTIONS

- When this program is executed, the user has to enter the sentence.
- Note that gets() function is used to read the sentence because the string has white spaces between the words.
- The vowels are counted using a switch statement in a loop.
- Note that a count++ statement is given only once to execute it for the cases in the switch statement.

Output:

Enter the sentence:

This is a book

5 vowels are present in the sentence.

Program: Write a C program to count no of lines, words and characters in a given text.

```
# include<stdio.h>
# include<string.h>
# include<conio.h>
main()
{
    char txt[250], ch, st[30];
    int  ins, wds, chs, i;
    printf("\n Enter the text, type $ st end \n \n");
    i=0;
    while((txt[i++]= getchar( ) ) != '$');
    i--;
    st[ i ] = '\0';
    ins = wds = chs = 0;
    i=0;
```

```

while(txt[ i ]!='$')
{
    switch(txt[ i ])
    {
        case ` `:
        case `!`:
        case `\\t`:
        case `\\n`:
        {
            wds ++;
            chs ++;
            break;
        }
        case `?`:
        case `.`:
        {
            wds ++;
            chs ++;
            break;
        }
        default:
            chs ++;
            break;
    }
    i++;
}
printf("\\n\\n no of char (incl.blanks) = %d", chs);
printf("\\n No. of words = %d", wds);
printf("\\n No of lines = %d", ins);
getch() ;
}

```

Output:

Enter the text, type \$ at end

What is a string? How do you initialize it? Explain with example.

With example: \$

No of char: (inch. Blanks) = 63

No of words = 12

No of lines =1.

1.18 Multi Dimensional Array

A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays are stored in row-major order.

The **general form of declaring N-dimensional arrays** is:

`data_type array_name[size1][size2]....[sizeN];`

data_type: Type of data to be stored in the array.

array_name: Name of the array

size1, size2,... ,sizeN: Sizes of the dimension

- **Examples:**

Two dimensional array: `int two_d[10][20];`

Three dimensional array: `int three_d[10][20][30];`

Size of Multidimensional Arrays:

The total number of elements can be stored in a multidimensional array Can be calculated by multiplying the size of all dimensions.

For example:

The array `x[10][20]` can store total ($10 * 20$)=200 elements.

The array `x[5][10][10]` can store total ($5*10*20$) = 1000 elements.

Entering elements into a 3D Array

```
#include <stdio.h>
int main()
{
int c[2][4][3];
int i,j,k;
printf("Enter elements into 3-D array: ");
for(i=0;i<2;i++)
{
for(j=0;j<4;j++)
{
for(k=0;k<3;k++)
{
scanf("%d",&c[i][j][k]);
}
}
}
}
```

10. Assignment Questions

S.No.	Write programs for the given problems	K-Level	COs										
1.	<p>Write a program to generate an electricity bill for N customers and find the customer who paid the maximum amount among them. The inputs are current month and previous month reading. You have to calculate the bill amount based on the following tariff:</p> <table><tr><td>Units consumed</td><td>Rupees per unit</td></tr><tr><td>≤ 100</td><td>Nil</td></tr><tr><td>101 to 200</td><td>3</td></tr><tr><td>201 to 400</td><td>5</td></tr><tr><td>> 400</td><td>7</td></tr></table> <p>Eg. 325 units ($0 \times 100 + 3 \times 100 + 5 \times 125 = 925$)</p>	Units consumed	Rupees per unit	≤ 100	Nil	101 to 200	3	201 to 400	5	> 400	7	K3	CO1
Units consumed	Rupees per unit												
≤ 100	Nil												
101 to 200	3												
201 to 400	5												
> 400	7												
2.	<p>Given an input number N and N numbers following it as the input. You are also given an integer K which represents how many times you have to rotate the array. Rotate the array K values to the right if K is positive and K values to the left if K is negative. If K=0, do not rotate the array.</p>	K3	CO1										
3.	<p>Write a program to reverse the words of a string : Eg : "How are you" The Output is : "you are How"</p>	K3	CO1										

11. Part A

Question & Answer



R.M.K.
GROUP OF
INSTITUTIONS

Part A

1. Define an algorithm. (CO1)(K2)

An algorithm is a self-contained sequence of actions to be performed. Algorithms can perform calculation, data processing and automated reasoning tasks. It is an effective procedure for solving a problem in a finite number of steps.

2. List the characteristics of an algorithm. (CO1)(K2)

- Be precise
- Be unambiguous
- Not even a single instruction must be repeated infinitely.
- After the algorithm gets terminated, the desired result must be obtained.

3. What are the qualities of an algorithm? (CO1)(K2)

The following are the primary factors that are used to judge the quality of an algorithm,

- Time
- Memory
- Accuracy
- Sequence

4. List out the ways to represent an algorithm. (CO1)(K2)

- Flowchart
- Pseudocode
- Programming languages

5. What are the various control flow structures in an algorithm? (CO1)(K2)

- Sequence
- Selection
- Repetition

6. What is sequence control flow? (CO1)(K2)

In sequence construct, statements are executed one by one in the order from top to bottom.

7. What is selection control flow? (CO1)(K2)

- Statements are used when the outcome of the process depends on some condition.
- A condition in this context may evaluate either to a true or false value.

8. What is repetition? (CO1)(K2)

- It is used to execute a block of code or a part of the program several times.
- In other words, it iterates a code or group of code many times.

9. Write the guidelines for preparing flowcharts. (CO1)(K2)

- The flow chart should be clear, neat and easy to follow
- The flowchart must have logical start and finish
- In drawing a proper flowchart, all necessary requirements should be listed in logical order.
- Only one flow line should come out from a process symbol.
- Only one flow line should enter a decision symbol. Two or three flow lines may leave the decision symbol.
- Only one flow line should be used with a terminal symbol.

10. Define flowchart. (CO1)(K2)

It is a pictorial representation of an algorithm in which the steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows.

11. Write the advantages of flowchart. (CO1)(K2)

- The pictorial representation helps in understanding the logic clearly.
- Makes Logic Clear
- Assists in finding the key elements of a process by drawing clear flow lines.
- It encourages communication among programmers and users
- Effective Analysis. It reveals redundant or misplaced steps

12. List the limitations of flowchart. (CO1)(K2)

- Difficult to Modify
- Cannot be updated frequently
- Complex to design for larger programs
- Costly

13. List the advantages and disadvantages of iteration. (CO1)(K2)

Advantages

Complexity broken down

Code Reusability

Disadvantages

Less Parallelism

14.What is looping? (CO1)(K2)

Looping is a sequence of instructions that is continually repeated until a certain condition is reached.

15.Define Problem Solving. (CO1)(K2)

Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately.

16.Write an algorithm for interchanging / swapping two values. (CO1)(K2)

Step1: Start

Step2: Read the two numbers as a and b

Step3: set c=a Step4: set a=b Step5: set b=c Step6: print a, b Step7: Stop

17.Write an algorithm to find whether a number is even or odd. (CO1)(K2)

Step1: Start

Step2: Read a number as A

Step3: if $A \% 2 == 0$ then print "Even" else print "Odd"

End if

Step4: Stop

18.Write the algorithm to find the average of three numbers. (CO1)(K2)

Step 1: START

Step 2: INPUT a,b,c

Step 3: COMPUTE $\text{avg} = (a+b+c) / 3$

Step 4: PRINT avg

Step 5: STOP

19.Write an algorithm for finding the factorial of a given number. (CO1)(K2)

Step1: Start

Step2: Read the number.

Step3: Initialize $i=1$, $\text{fact}=1$

Step4: Repeat step4 through6 until $i < n$

Step5: $\text{fact}=\text{fact} \times i$ Step6: $i=i+1$ Step7: print fact Step8:

Stop

20.What are the Features and Application of C Language? (CO1)(K2)

C is a general purpose, structured programming language.

C is highly portable.

C has the ability to extend itself. We can continuously add our own functions to the existing library functions.

C is well suited for writing system software as well as application software.

21. What are C Tokens? (CO1)(K2)

The C Language program can contain the individual units called the C tokens.

The tokens are usually referred as individual text and punctuation in the text.

C tokens are classified into (i)Keywords (ii) Identifiers (iii) constant (iv)Operators (v)String (vi) special characters.

22. Define the rules of Keywords? (CO1)(K2)

The c keywords are reserved words by the compiler.

The keywords cannot be used as variable name.

All keywords must be written in lower case.

Keywords are predefined and standard meaning

· 23. Define the rules of Identifiers? (CO1)(K2)

Identifiers contain letters and digits

The first character should be a letter or underscore.

Underscore is the only special character that is allowed in an Identifier.

Length of the identifier is significant up to certain digits depends on the compiler.

Identifiers are case sensitive.

· 24. List the rules that are to be followed while declaring variable? (CO1)(K2)

A variable name can be any combination of alphabets, digits and underscore.

The first character must be an alphabet or an underscore (_).

No commas or blank space are allowed within a variable name.

No special symbol can be used in a variable.

· 25. what is variable and write syntax of the variable? (CO1)(K2)

A variable is an identifier.

A variable may take different values at different times during the execution. Syntax: Data_ type variableName;

Example: int age;

26. Define Constant and volatile variable? (CO1)(K2)

Constant value of a certain variable remains the same or remains unchanged during the execution of a program. It can be done by declaring the variable as a constant.

The keyword const is added before the declaration.

Example: `const int m=10;`

The volatile variables are those variables that are changed at any time by program. Example: `volatile int d;`

27. List different datatypes in c? (CO1)(K2)

Primitive/ basic datatype

Derived datatype

User defined datatype

28. what are the primitive datatype? (CO1)(K2)

Integer

character

float

double.

Void

29. If a=50, b=10 and c=20 evaluate the following expression. (CO1)(K3)

`c=(a>0&&a<=10)?++a:a/b`

`c=(50>0&&50<=10)?++50:50/10`

C=5

31. List out the Operators in C? (CO1)(K2)

Arithmetic Operators

Relational operators

Assignment Operators

Logical operators

Bitwise Operators

Conditional or Ternary Operators

Increment and Decrement operators

Special operators

32. Write Syntax of the Conditional operators with example. (CO1)(K2)

Syntax : $\text{exp1} ? \text{exp2}$

: exp3 ; Example $x =$

$(a > b) ? a : b$

Work : exp1 is evaluated first .if it is true , then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated.

33. Evaluate the following Expressions.

i) $4 + 8/2 * 7 + 4$

$$= 4 + 4 * 7 + 4$$

$$= 4 + 28 + 4$$

$$= 32 + 4$$

$$= 36$$

ii) $4 \% 3 * 5/2 + (5 * 5)$

$$= 4 \% 3 * 5/2 + 25$$

$$= 1 * 5/2 + 25$$

$$= 5/2 + 25$$

$$= 2.5 + 25$$

$$= 27.5$$

34. Define formatted input and output functions? (CO1)(K2)

- scanf()

scanf() stands for Scan Formatted.

scanf() is used to read values in a specified format.

The general syntax is :

```
int scanf("Control-String", argument_list);
```

Example: scanf("%d",&a);

- printf()

It specifies print formatted and is used to display a message.

The syntax is printf("Control-string", argument-list);

Control string consists of ,

Messages to be printed on the screen.

Format specifiers to define the format for the values to be displayed.

Escape sequence characters like \n, \t etc. Example:

```
printf("%d",a);
```

35. Explain the typedef datatype? (CO1)(K2)

User to define an identifier that would represent an existing datatype.

Renaming the existing datatype.cannot create new datatype.

Syntax: typedef datatype identifier;

typedef is keyword,datatype is existing datatype,identifier refers to the new name given to the data type.

Example: typedef int mark;

36) What do you mean by an Array? List Advantages and disadvantages of Array?

- Array is a set of similar data type.
- Arrays objects store multiple variables with the same type.
- It can hold primitive types and object references.
- Arrays are always fixed

Advantages:

- We can put in place other data structures like stacks, queues, linked lists, trees, graphs, etc. in Array.
- Arrays can sort multiple elements at a time.
- We can access an element of Array by using an index.

Disadvantages:

- We have to declare Size of an array in advance. However, we may not know what size we need at the time of array declaration.
- The array is static structure. It means array size is always fixed, so we cannot increase or decrease memory allocation

37) What are the main elements of an array declaration?

- o Array name
- o Type and
- o Size

38. How to initialize an array?

You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [].

39. Give an example of 2D array initialization.

```
int disp[2][4] = {  
    {10, 11, 12, 13},  
    {14, 15, 16, 17}  
};
```

Or `int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};`

12.PART – B Questions

1. Draw a flowchart, write an algorithm to find largest of three numbers.
2. Draw a flowchart, write an algorithm to find the factorial of a number.
3. Draw a flowchart, write an algorithm to reverse the given number.
4. Write an algorithm, flowchart to find the sum of first N natural numbers.
5. Write an algorithm, flowchart to display the Fibonacci series.
6. Draw a flowchart, write an algorithm to check the given number is positive, negative or zero.
7. Write an algorithm, flowchart for $1^2+2^2+3^2+ \dots +n^2$
8. Draw a flowchart, write an algorithm to check the given number is palindrome or not.
9. Describe the various types of operators with example supported by C.
10. Describe the features of primitive data types and user defined data types in C.
11. Explain the data types and constants supported by C.
12. Define Array. Explain the types of Array.
13. Write a C Program for arranging the elements into ascending order.

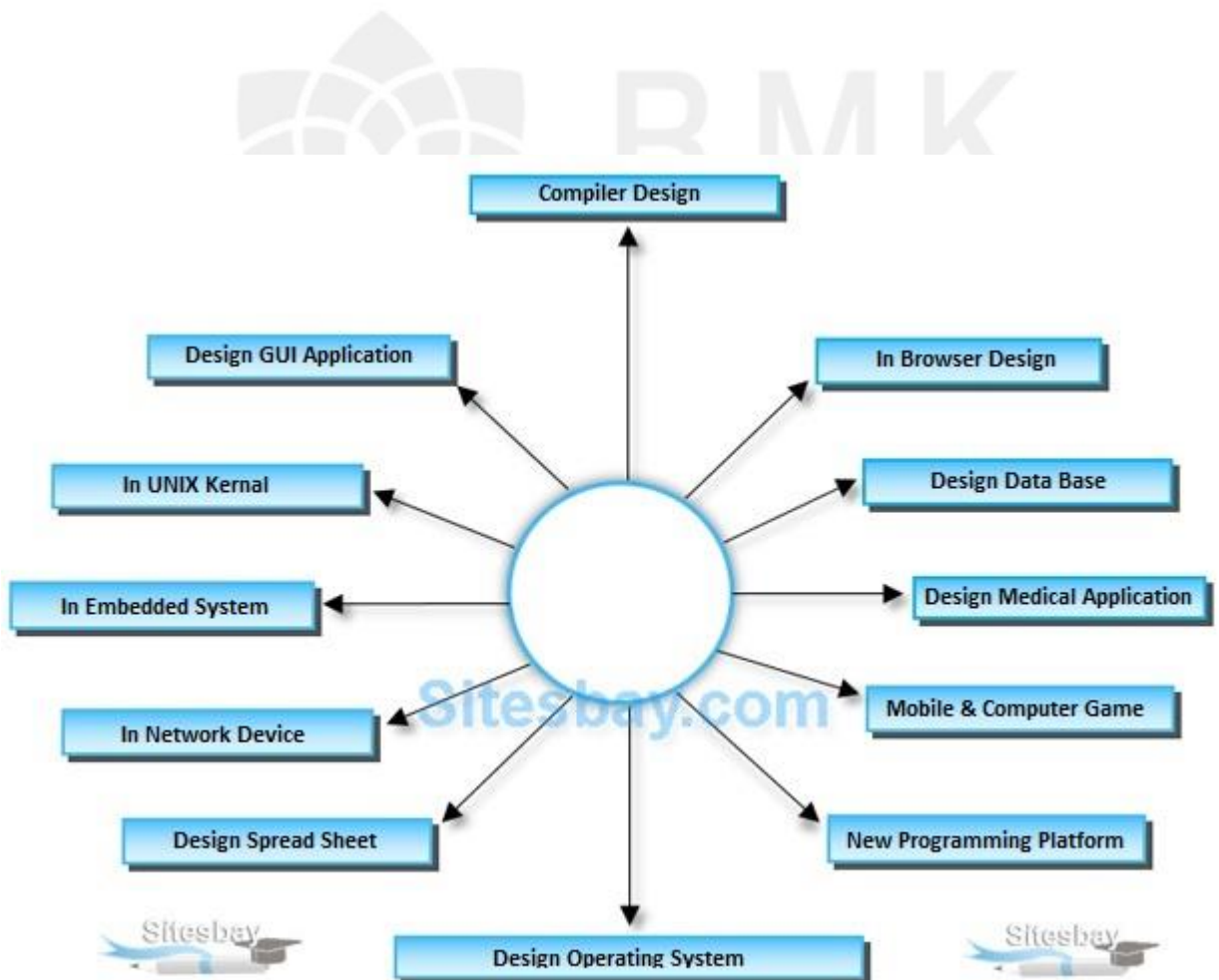
13. Supportive online courses

- ❖ Introduction to Programming in C – Coursera
- ❖ C for Everyone: Programming Fundamentals – Coursera
- ❖ C for Everyone: Structured Programming - Coursera
- ❖ Computational Thinking with Beginning C Programming - Coursera
- ❖ Problem solving through Programming In C – Swayam
- ❖ C – Sololearn
- ❖ C Programming For Beginners – Udemy
- ❖ C Programming For Beginners - Master the C Language – Udemy

External Links for additional resources

- ❖ <https://swayam.gov.in/>
- ❖ <https://www.coursera.org/>
- ❖ <https://www.udemy.com/>
- ❖ <https://unacademy.com/>
- ❖ <https://www.sololearn.com/>
- ❖ <https://www.tutorialspoint.com/cprogramming/index.htm>
- ❖ <https://www.w3schools.in/c-tutorial/>
- ❖ <https://www.geeksforgeeks.org/c-language-set-1-introduction/>
- ❖ <https://www.programiz.com/c-programming>

14. Real Time Applications



15. Content beyond syllabus

Preprocessors Directives:

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do require pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

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

```
#include "myheader.h"
```

These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
```

```
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
```

```
    #define MESSAGE "You wish!"
```

```
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
```

```
    /* Your debugging statements here */
```

```
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to the gcc compiler at the time of compilation. To show define DEBUG, so you can turn debugging on and off on the fly during compilation.

S.No.	Directive & Description
1	<code>#define</code> Substitutes a preprocessor macro.
2	<code>#include</code> Inserts a particular header from another file.
3	<code>#undef</code> Undefines a preprocessor macro.
4	<code>#ifdef</code> Returns true if this macro is defined.
5	<code>#ifndef</code> Returns true if this macro is not defined.
6	<code>#if</code> Tests if a compile time condition is true.
7	<code>#else</code> The alternative for <code>#if</code> .
8	<code>#elif</code> <code>#else</code> and <code>#if</code> in one statement.
9	<code>#endif</code> Ends preprocessor conditional.
10	<code>#error</code> Prints error message on stderr.
11	<code>#pragma</code> Issues special commands to the compiler, using a standardized method.

16. Assessment Schedule

Tentative schedule for the Assessment During 2022-2023 Odd semester

S.NO	Name of the Assessment	Start Date	End Date	Portion
1	UNIT TEST 1	24.09.2023	29.09.2023	UNIT 1
2	IAT 1	16.10.2023	21.10.2023	UNIT 1 & 2
3	UNIT TEST 2	30.10.2023	05.11.2023	UNIT 3
4	IAT 2	23.11.2023	29.11.2023	UNIT 3 & 4
5	REVISION 1	05.12.2023		UNIT 5 , 1 & 2
6	REVISION 2	08.12.2023		UNIT 3 & 4
7	MODEL	12.12.2023	21.12.2023	ALL 5 UNITS

17. Text books & References

Text Books:

1. Herbert Schildt, "The Complete Reference C++", 4th edition, MH, 2015. (Unit 1 & 2)
2. E Balagurusamy, "Object Oriented Programming with C++", 4th Edition, Tata McGraw-Hill Education, 2008. (Unit 3, 4 & 5)

Reference Books:

1. Nell Dale, Chip Weems, "Programming and Problem Solving with C++", 5th Edition, Jones and Barklett Publishers, 2010.
2. John Hubbard, "Schaum's Outline of Programming with C++", MH, 2016.
3. Yashavant P. Kanetkar, "Let us C++", BPB Publications, 2020
4. ISRD Group, "Introduction to Object-oriented Programming and C++", Tata McGraw-Hill Publishing Company Ltd., 2007.
5. D. S. Malik, "C++ Programming: From Problem Analysis to Program Design", Third Edition, Thomson Course Technology, 2007.
6. https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_01297200240671948837_shared/overview

18. Mini Project Suggestions

1. Simulate simple calculator using functions
2. Write an application program for placing order online
 1. Order placement
 2. Payment gateway
 3. Goods delivery
3. Write a program for online railway ticket booking





Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.