CHARUSAT
CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

cspit
Chandubhai S Patel Institute of Technology

Faculty of Technology and Engineering

U & P U. Patel Department of Computer Engineering

Date: 30 / 01 / 2025

Academic Year	:	2024-2025	Semester	:	2
Course code	:	CEUC102	Course name	:	Programming with C++

Practical-CO Mappings

Category	Description	COs Mapped	Hours
Overview of C++ Programming	Implement `BankAccount` class to demonstrate OOP concepts and compare procedural vs. OOP approaches for inventory management.	CO1, CO4	2
Classes and Objects	Defining and implementing classes such as Rectangle, Student, and BankAccount to demonstrate object-oriented principles.	CO1, CO2, CO4	5
Function and Recursion	Implement employee salary calculation with default arguments, recursive array sum, `BankAccount` class with friend functions, generic templates for managing collections, and compute the "super digit" of integers using recursion and STL benefits.	CO2, CO6	5
Inheritance and Polymorphism	Exercises on multi-level inheritance, virtual functions, operator overloading, and designing class hierarchies (e.g., grading systems).	CO4, CO6	5
Function Templates and STL	Using templates and STL containers (std::vector, std::map, std::queue) for operations like reversing arrays, word frequency analysis, and directory management.	CO2, CO6	5
Pointers and Dynamic Memory Allocation	Tasks include dynamic array management, merging sorted arrays, and implementing virtual	CO3	4

	destructors for resource management.		
Stream I/O and File Handling	Includes tasks like reading and processing file data, analyzing word frequencies, and generating formatted reports.	CO3, CO6	5
Error and Exception Handling	Scenarios involving division by zero, invalid inputs, custom exception handling, and simulating systems like banking.	CO5, CO6	5

OVERVIEW OF C++ PROGRAMMING

1.1

A bank wants to create a simple system to manage customer bank accounts. The system should allow customers to perform basic banking operations such as depositing money, withdrawing money, and checking their account balance.

Each bank account will need to have an account holder's name, a unique account number, and a balance. Deposits should increase the account balance, while withdrawals should only be allowed if there are sufficient funds in the account. If an attempt is made to withdraw more money than is available, an error message should be displayed. Customers should also have the ability to view their account balance whenever required.

The system must be designed using Object-Oriented Programming principles, focusing on creating a simple and efficient solution to manage the accounts effectively. The system should ensure that all account details are secure and accessible only through authorized methods.

1.2

A small retail store is facing challenges in managing its inventory effectively. The store sells a variety of products, each identified by a unique product ID, a name, the available quantity in stock, and the price per unit. To streamline their operations, the store needs a basic system to manage this inventory efficiently.

The system must provide the ability to add new products to the inventory, ensuring that each product has its ID, name, quantity, and price properly recorded. Additionally, the system should allow the store staff to update the quantity of any existing product, such as when new stock arrives or when items are sold.

Another essential feature of the system is the calculation of the total value of all products in the inventory, which is determined by multiplying the quantity of each product by its price and summing these values for all products.

The store management is exploring two approaches for this system: a procedural approach and an object-oriented approach. The goal is to evaluate these approaches by comparing their ease of implementation, code reusability, and overall complexity.

The system's design and implementation should consider these requirements and provide an effective solution (either procedural approach or an object-oriented approach) to the store's inventory management problems.

2. Class & Object

2.1

A local construction company frequently deals with rectangular plots and structures of varying dimensions. To streamline their planning and estimation processes, the company requires a simple system to manage and analyze rectangular shapes efficiently.

The system must be able to handle multiple rectangles, each with distinct dimensions. For each rectangle, the length and width need to be defined and stored securely. Additionally, the company needs the ability to calculate two key metrics for any given rectangle:

The area, to estimate usable space or material coverage. The perimeter, to estimate boundary lengths or material requirements for edges. To make this system functional, there should be a way to define or update the dimensions of any rectangle as required. The system should be capable of creating and managing multiple rectangle records, performing calculations for each, and displaying the results clearly for analysis and planning purposes.

2.2

A university managing academic data for its students. The administration seeks to digitize the records of student performance, including personal and academic details. The system must store each student's roll number, name, and marks in three subjects. Additionally, it should provide functionalities to calculate and display the average marks for performance analysis.

The institution's IT team proposed a solution where student records could be initialized in two ways: either with default values for testing purposes or by accepting specific input details for each student. Once initialized, the system would allow for viewing comprehensive student details, including their roll number, name, marks, and calculated average. This functionality was designed to help faculty and administrators track individual student performance efficiently.

To simulate real-world usage, the team decided to create multiple student records. They planned to populate the system with a mix of students initialized using both default and specific values. The system's ability to accurately calculate averages and display detailed student information was to be tested with this data.

2.3

In a growing city, a newly established bank sought to streamline its operations by implementing a digital system to manage customer accounts. The management envisioned a system where each account would store details about the account holder, their unique account number, and the current balance. To ensure flexibility and accuracy, they required functionalities for creating accounts with varying initial balances, depositing money, withdrawing funds (with checks for sufficient balance), and generating detailed account summaries.

The bank's IT team faced the challenge of developing a robust solution. They began by sketching out the essential features of the system. The team highlighted that new accounts could be created in two ways: one with no initial balance, and another with specified account details and a starting amount. Additionally, they recognized the need for reliable mechanisms to handle deposits and withdrawals, ensuring proper validation to prevent overdrafts. Displaying account details in a clear format was also prioritized for customer communication.

To ensure scalability, the team decided to simulate the system by creating multiple accounts using the proposed methods. They tested various scenarios, such as depositing and withdrawing different amounts, handling edge cases like insufficient funds, and verifying that the account summaries were accurate. This iterative approach helped them refine the system and ensure its readiness for deployment.

2.4

A mid-sized retail store faced challenges in efficiently managing its inventory of items. The store's management sought to build a system that could keep track of individual items, including details like a unique item ID, item name, price, and the quantity available in stock. The need for a streamlined process arose due to frequent stock discrepancies, which led to issues with customer satisfaction and operational efficiency.

To address this, the store hired a team of developers to create a digital inventory management system. The envisioned solution would allow for the initialization of item details, either with default values or specific attributes like ID, name, price, and starting quantity. This system needed to handle operations like increasing stock levels when new shipments arrived and decreasing stock when items were sold, ensuring sufficient inventory was available for each transaction. Additionally, the system would provide clear, detailed summaries of each item's status, aiding in decision-making and reporting.

The developers faced real-world scenarios where they had to manage multiple inventory items simultaneously. They planned to design an array of inventory items and simulate common tasks such as adding stock, processing sales transactions, and displaying the current inventory details. Handling edge cases, such as attempting to sell more items than available in stock, became a critical part of the implementation to ensure reliability.

2.5

A regional banking institution sought to improve its loan management process by developing a system that could efficiently handle loan details for applicants. The system was expected to streamline the calculation of monthly EMIs (Equated Monthly Instalments) and provide detailed loan summaries for customers. This initiative aimed to enhance customer experience by offering accurate and transparent information about their loans.

To meet these requirements, the bank's technology team was tasked with designing a solution. They envisioned a system where each loan would be uniquely identified by a loan ID, and additional details such as the applicant's name, total loan amount, annual interest rate, and loan tenure in months would be stored. The team also emphasized the importance of accurately calculating the EMI using a standard formula based on the loan amount, interest rate, and tenure. The formula incorporated compound interest principles to determine the fixed monthly payment for the loan term.

$$\text{EMI} = \frac{P * R * (1 + R)\text{^}T}{((1+R)\text{^}T) - 1}$$

As part of the implementation, the developers planned to initialize loans either with default values for testing purposes or with actual customer data provided at the time of application. The system needed to include a feature to display comprehensive loan details, including the calculated EMI, in a customer-friendly format. This functionality was intended to aid both customers and bank staff in managing loan-related queries effectively.

3. Function and Recursion

3.1

A growing organization sought to enhance its payroll process to improve accuracy and efficiency in calculating employee salaries. The goal was to manage details such as the employee's name, basic salary, and bonus amount. While most employees would receive a default bonus, there was a need for flexibility to provide customized bonuses for exceptional performers or those in specialized roles.

To ensure real-time computation of each employee's total salary, the system would dynamically calculate the total salary by adding the basic salary and the bonus. An inline function was chosen for this task, allowing the system to compute the total salary instantly during processing. The system was designed to initialize employee records using a constructor, where the bonus could either be set to a default value or adjusted according to specific employee criteria.

As the organization expanded, managing employee records efficiently became a key consideration. The solution involved storing employee details dynamically, ensuring the system could handle a growing number of records while maintaining scalability and flexibility. Each employee record would include their personal details, basic salary, and bonus information, with the system displaying a breakdown of each employee’s details along with their total salary.

3.2

A software development company was tasked with conducting a performance analysis of recursive algorithms versus their iterative counterparts. The specific focus was on calculating the sum of integers in an array, where the array's size and elements were to be provided dynamically by the user. To facilitate memory management and enable dynamic resizing of the array, the team decided to use a flexible container for storing the array elements.

The system was designed to first prompt users for the array's size and then request the input of individual elements. A recursive function was to be implemented to compute the sum by dividing the problem into smaller sub-problems, recursively summing subsets of the array until reaching the base

case. In addition to the recursive implementation, an iterative version of the function would be created for comparison.

The main objective of the study was to assess and compare the computational performance and implementation complexity of both recursive and non-recursive approaches. By evaluating execution times, memory usage, and code complexity, the team hoped to gain insights into the trade-offs between recursion and iteration, particularly in terms of efficiency and applicability to real-world problems.

3.3

A community bank sought to enhance its account management system with a digital solution to improve efficiency and provide better customer service. The system was required to manage the essential details of each account, including the account number, account holder's name, and balance. Additionally, the bank wanted to provide a secure mechanism for transferring money between accounts, allowing customers to easily manage their funds.

The bank also needed a way to track the total number of accounts created, which would be important for generating reports and understanding the growth of their customer base. This feature was aimed at helping the bank maintain an overview of their account portfolio and analyze trends over time.

To ensure smooth and reliable operations, the system was designed to store account information in a way that would allow easy access and updates. When new accounts were created, they would be added to the system dynamically. The management team planned for future scalability and performance improvements by considering more efficient storage and retrieval methods after the initial system was built, ensuring that the bank could easily accommodate more accounts and transactions as the customer base grew.

3.4

A technology firm aimed to develop a flexible and reusable solution for managing collections of various data types, including integers, floating-point numbers, and characters. The system was intended to perform fundamental operations on these collections, such as finding the maximum value, reversing the collection, and displaying all elements. To achieve versatility and avoid redundancy in code, the solution was designed to use function templates, allowing the same logic to be applied seamlessly to different data types.

The team recognized the importance of using dynamic arrays to store the collections, enabling efficient management of varying collection sizes. The design emphasized scalability and flexibility, ensuring that the system could handle different data types and their associated operations with minimal changes to the core logic.

In practice, the system allowed for the creation of collections for various data types, such as integers, floating-point numbers, and characters. The operations on these collections included determining the maximum value, reversing the order of elements, and printing the collection contents.

3.5

A data analytics company was tasked with developing a unique digital signature system based on the concept of "super digits." The system required finding a single-digit representation of a given number through recursive digit summation. The algorithm was defined as follows:

If the number has only one digit, it is its super digit. Otherwise, the super digit is the super digit of the sum of its digits, repeated recursively until a single digit is obtained.

The challenge involved an additional complexity—constructing the number by concatenating a given string representation of an integer multiple times. For example, if the number n was represented as a string and concatenated k times, the super digit of the resulting number needed to be calculated.

For instance:

Given n = 9875 and k = 4, the number is represented as 9875987598759875.

The sum of digits in this number is calculated recursively until a single digit remains:

9 + 8 + 7 + 5 + 9 + 8 + 7 + 5 + 9 + 8 + 7 + 5 + 9 + 8 + 7 + 5 = 116

1 + 1 + 6 = 8

The super digit is 8.

The system was required to handle large numbers efficiently by leveraging mathematical insights rather than explicitly constructing large concatenated strings. This case study called for implementing a recursive solution to calculate the super digit, supported by a mathematical approach to optimize the handling of repeated sums.

4. Inheritance

4.1

A team of engineers was tasked with developing a program to calculate and manage the areas of multiple circles for a design project. To achieve this, they devised a solution using a structured, object-oriented approach. At the foundation of their solution was a base class that represented a generic shape, responsible for storing and managing the radius of the circle. Building upon this, a specialized class for circles was created to extend functionality by introducing a method for calculating the area of a circle based on its radius. Using this framework, the team designed a system to handle multiple circles, ensuring that the process of storing, calculating, and displaying the areas was efficient and adaptable. They explored two different approaches for managing the collection of circles—one focusing on flexibility and dynamic handling, while the other used a more static structure. By implementing and comparing these methods, the engineers gained insights into the benefits of using efficient techniques for organizing and processing geometric data, enhancing their problem-solving capabilities.

4.2

A growing organization wanted to develop a system to manage its hierarchy and represent its structure in a programmatic way. To achieve this, a multilevel approach was designed, reflecting the natural progression of roles within the organization. At the foundation, a class was created to represent a person, capturing the basic details such as name and age. Building on this, an intermediate level was introduced to represent employees, adding a unique identifier for each. Finally, at the topmost level, a class for managers was created, which included additional details such as the department they oversee.

The system needed to handle the initialization of all these attributes through constructors at each level, ensuring the proper propagation of information across the hierarchy. Additionally, the functionality to display details at every level was included to provide clear insights into the organization's structure. Two approaches were explored for managing multiple managers: one relied on an efficient method for retrieval and organization based on employee identifiers, while the other used a straightforward and static method for storage.

4.3

A vehicle manufacturing company sought to create a robust system to organize and manage the details of various cars produced under its brand. To accomplish this, a hierarchical structure was conceptualized, reflecting the essential components of a vehicle. At the foundation, a class was designed to represent the type of fuel a vehicle uses. Another class was created to capture the brand name of the vehicle. These two foundational elements were then combined into a derived class specifically representing cars, integrating both fuel type and brand information.

Constructors were used at each level to ensure proper initialization of attributes, allowing seamless integration of all details. Additionally, the ability to display complete information about a car, including its fuel type and brand, was incorporated into the system. To simulate a real-world scenario such as a service queue, multiple cars were organized and processed sequentially using a structured approach. This not only streamlined the handling of cars but also provided an opportunity to compare different methods of managing the collection and processing of vehicle data.

4.4

In a bid to design an efficient and user-friendly banking system, a structure was proposed that mirrors the real-world operations of various account types. The foundation of the system is a base class representing a generic bank account, encapsulating essential details such as account number and balance. Building on this foundation, two specialized account types were created: a savings account, which includes an interest rate as an additional feature, and a current account, which allows an overdraft limit to accommodate specific customer needs.

To ensure proper initialization and cleanup of account objects, constructors and destructors were implemented. Essential banking operations such as deposits and withdrawals were made available for both account types, allowing users to perform and manage their transactions effectively. The system also accounted for the need to track and manage transaction history, enabling operations such as undoing the last transaction. This was achieved by simulating a mechanism to store a sequence of transactions for each account type, providing insight into different ways of managing and organizing data.

4.5

In an educational setting, an advanced grading system was conceptualized to accommodate the diverse evaluation criteria for students at different academic levels. At the heart of the system is an abstract base class that defines the grading framework. This class includes a protected member to store marks and declares a pure virtual function for computing grades, ensuring that specific grading logic is implemented by derived classes.

Two distinct derived classes were introduced to handle the unique grading needs of undergraduate and postgraduate students. Each class defines its own implementation of the grade computation method, reflecting the varying academic expectations for these groups. The system enables users to input student data, compute grades based on the respective criteria, and manage a collection of student records.

5. Polymorphism

5.1

A software development team aimed to design a versatile utility that could perform basic arithmetic operations while demonstrating the concept of function overloading. This effort resulted in a class-based calculator system capable of handling various combinations of input types, such as integers and floating-point numbers. The system includes multiple overloaded add functions, each tailored to accept distinct input types and perform addition operations accordingly. This approach ensures the calculator is adaptable and provides consistent functionality regardless of the input types.

The case revolves around testing and validating the versatility of this calculator. Participants are expected to create instances of the calculator, invoke the appropriate overloaded functions for various input scenarios, and store the resulting values for further analysis. To organize and display these results, the system incorporates methods to sequentially process and present the outcomes.

5.2

A mathematical research group aimed to create a software model for handling and performing operations on complex numbers efficiently. To achieve this, they designed a Complex class that encapsulates the real and imaginary parts of a complex number. The class supports essential operator overloading to enhance usability, including the addition and subtraction of complex numbers and custom input and output functionality through the << and >> operators. These overloaded operators ensure seamless arithmetic and user interaction with the system.

The task involves implementing this system and exploring its capabilities by performing various operations on complex numbers. Participants are expected to overload the specified operators and use them to add and subtract complex numbers, as well as to facilitate user-friendly input and output. Additionally, the solution encourages experimenting with managing collections of complex numbers to perform batch operations.

5.3

A team of software developers was tasked with creating a graphical simulation where operations on 2D points play a crucial role. To facilitate this, they designed a class Point that encapsulates the coordinates x and y. The class provides flexibility through overloaded operators to manipulate points efficiently. The unary operator - is overloaded to negate the coordinates of a point, while the binary operator + enables the addition of two points. Additionally, the equality operator == is overloaded to compare whether two points have identical coordinates.

The development process required performing various operations between multiple points. To manage these operations effectively, a mechanism was needed to track and potentially undo them. This challenge prompted two approaches: using a ready-made dynamic stack structure or building a custom stack implementation. By managing a sequence of operations in reverse, the stack-based design allowed for a systematic undo capability, crucial for simulations involving iterative adjustments.

5.4

A team of developers is tasked with building a temperature conversion system for a weather application. To achieve this, the team creates two classes: Celsius and Fahrenheit. These classes handle the conversion between temperature units, with the ability to convert from Celsius to Fahrenheit and vice versa using type conversion. The team utilizes operator overloading to define the conversion operators for both classes, enabling seamless conversions between the two units.

The system also requires the ability to compare two temperature objects to check if they are equal. This is achieved by overloading the equality operator ==, which compares the values of the temperatures in their respective units.

To ensure smooth processing of temperature conversions, the team needs to manage and store multiple converted temperature objects. Two approaches are considered for handling this task. The first approach involves using a dynamic data structure, a queue, to process the conversions in a first-in-first-out (FIFO) manner. Alternatively, a basic array is used to store the converted objects in a static manner.

5.5

A software development team is tasked with designing a system that can handle various geometric shapes and compute their areas in a flexible way. The challenge is to create a system that can easily extend to accommodate new types of shapes without altering the core functionality for each shape. To accomplish this, the system is designed with a base class called Shape, which includes a virtual function Area(). This function is meant to be overridden by each specific shape class to provide the correct formula for calculating the area.

The derived classes, Rectangle and Circle, each implement the Area() function with their own formulas: the Rectangle calculates the area using its length and width, while the Circle uses its radius. This structure allows the system to treat all shapes uniformly through a common interface, enabling easy extensibility. The goal is to use a single reference to the base class (Shape*) to calculate the area

of any shape, regardless of its type. This approach makes the system more adaptable, as new shapes can be added later without disrupting existing code.

In managing a collection of shapes, there are two primary approaches: one method involves dynamically managing a collection of shapes, allowing for easy addition and removal of shapes, while the other relies on a static method that requires manually managing the size of the collection. Both approaches aim to store and manage the shapes effectively, with one allowing automatic resizing as needed while the other requires more manual handling.

Through this case study, the team will implement the base class Shape and the derived classes Rectangle and Circle, each with their own Area() function. The students will gain a deeper understanding of polymorphism, inheritance, and memory management while working with different methods to store and manage the collection of shapes.

6. Pointers and DAM

6.1

In a rapidly growing software development firm, a team of engineers is tasked with building a lightweight, custom memory management system for handling dynamic datasets. The existing framework lacks flexibility in managing arrays, often leading to inefficient memory usage and performance bottlenecks. To address this, the engineers need to develop a solution that allows seamless insertion and deletion of elements while ensuring efficient memory utilization.

The primary challenge is to design a structure that supports adding new data points dynamically at the end of the dataset and removing specific elements based on their position. Since the system operates in a resource-constrained environment, using standard template libraries is not an option, and direct dynamic memory management must be implemented using pointers. The solution should ensure that memory is allocated and deallocated appropriately to prevent leaks and unnecessary overhead.

The development team must carefully decide on the structure of their implementation, defining how data will be stored, accessed, and modified efficiently. They need to outline the appropriate class design, determine essential data members, and define necessary member functions to handle the operations effectively.

6.2

In a digital mapping application, a team of developers is working on a feature that allows objects to be repositioned dynamically on a coordinate plane. The application requires precise control over the movement of individual points representing user-defined markers. To achieve this, the team needs to design a system that allows a point’s coordinates to be adjusted efficiently while maintaining a fluid and chainable update mechanism.

To ensure seamless movement, the developers must implement a structure that allows each point to shift by specified offsets in the x and y directions. Additionally, the solution should be designed in a way that supports method chaining, enabling multiple transformations to be applied in a single statement. This requires careful handling of object references, utilizing this pointer to return the updated object.

Furthermore, the implementation must be optimized for direct memory management, ensuring that objects are updated through pass-by-reference using pointers.

6.3

A financial analytics company is working on a system that processes large volumes of sorted numerical data from different sources. The challenge is to efficiently combine two independently sorted datasets into a single, fully sorted dataset while ensuring optimal memory management. Since the size of the datasets varies dynamically based on incoming data streams, the solution must be flexible and manage memory efficiently without relying on built-in container libraries.

To achieve this, the development team needs to implement a mechanism that accepts two sorted arrays of different sizes and merges them into a new dynamically allocated array while maintaining the sorted order. The system should read input values specifying the sizes of both datasets, followed by the elements of each array. After merging, the final sorted array should be displayed as output.

Efficient memory handling is crucial in this implementation, requiring the use of new and delete operators for dynamic allocation and deallocation of memory.

6.4

A software development team is working on an advanced simulation system that involves hierarchical object management. The system includes a base class that defines general behavior and derived classes that introduce specialized functionality, including dynamic resource allocation. During testing, the team encounters unexpected memory leaks when deleting objects through base class pointers, raising concerns about proper resource management.

To investigate the issue, the team examines how object destruction works in an inheritance hierarchy. They create a base class that serves as a foundation for derived classes, but they notice that when a Base* pointer is used to delete a Derived object, the destructor of the derived class is not invoked. This results in dynamically allocated resources remaining in memory, causing resource leakage.

To resolve this problem, the team explores the concept of virtual destructors. By marking the base class destructor as virtual, they ensure that the destructor of the derived class is correctly called when deleting an object through a base class pointer. This guarantees proper deallocation of dynamically allocated resources, preventing memory leaks.

Since the project requires manual memory management, the team uses raw pointers and the new and delete operators to allocate and release resources.

7. Stream Input/Output and File Processing

7.1

A media research team is developing a tool to analyze word frequency in large textual datasets, such as news articles and research papers. The goal is to process a given paragraph, identify individual words, and count their occurrences while ensuring case-insensitive matching. Since the tool is intended for both constrained environments and high-performance systems, two different approaches are considered—one utilizing dynamic memory management and another relying on manually structured arrays.

The first challenge is reading an entire paragraph from the console as a single unformatted string. Once acquired, the text must be split into individual words, ensuring that uppercase and lowercase variations are treated as the same. To store and process words dynamically, the team designs a

mechanism using raw pointers and dynamic memory allocation, allowing the program to handle variable input sizes effectively.

In one approach, a dynamically allocated array is used to store words, with additional logic to count occurrences efficiently. The array expands as needed, ensuring that new words can be accommodated. The frequency counting is implemented manually by searching for existing words in the array and updating counts accordingly.

7.2

A data analysis firm is developing a tool to process large text files and extract key statistics, such as the total number of characters, words, and lines. This tool is essential for tasks like document indexing, text summarization, and data validation. Given the varying sizes of input files, the system must handle large datasets efficiently while ensuring error detection when files are missing or inaccessible.

To begin, the program needs to open a specified text file and process its contents line by line. If the file cannot be found or opened due to permission issues, the system should notify the user with an appropriate error message and safely terminate execution. Once the file is successfully accessed, the program will analyze its contents to count the total number of lines, extract words while handling different delimiters, and compute the total number of characters, including spaces and punctuation.

For handling this data, the team explores two approaches. One approach dynamically stores the lines in a manually managed array, processing the information without relying on built-in containers. This requires careful memory allocation and deallocation to avoid leaks. The alternative approach uses a dynamic structure to hold the lines in memory, allowing for efficient access and further analysis, such as searching or editing.

7.3

A small retail business is looking for a simple yet effective inventory management system to keep track of its products. The system needs to support essential operations such as adding new items, viewing the complete inventory, and searching for specific products. Since the inventory data should persist across sessions, all operations must be performed using file storage. The business also requires an efficient way to retrieve item details without unnecessary file reads.

To manage inventory, the system must allow employees to add new products by recording details such as the item name, quantity, and price. These records should be appended to a file, ensuring that previously stored data remains intact. When viewing inventory, the system should read the file sequentially and display all stored items. Additionally, employees should be able to search for a product by name, retrieving its details without manually scanning the entire file.

For implementation, two approaches are considered. One method processes file operations directly, reading and searching line by line to retrieve item information efficiently. This minimizes memory usage but requires re-reading the file for each search operation. An alternative approach loads inventory data into a dynamically managed array, enabling faster searches and future enhancements such as sorting and filtering without repeated file access.

7.4

An educational institution requires a system to generate well-structured student performance reports from raw data stored in a file. The system should read student records, including their name, marks, and grade, and present the information in a neatly formatted tabular format. Since the institution

handles large datasets, the report generation must be efficient, ensuring clear alignment and readability while allowing for future enhancements such as sorting and filtering.

To achieve this, the system must first handle file input operations, reading student data while ensuring error handling if the file is missing or inaccessible. The program should then format and display the information in a structured manner, making use of a user-defined manipulator to align columns properly. This ensures that names, marks, and grades are clearly visible in the report, regardless of data length variations.

For implementation, two approaches are explored. The first method processes the data directly from the file, formatting and displaying each entry without storing it in memory. This minimizes memory usage but limits additional processing, such as sorting or filtering. The alternative approach dynamically stores student records using a manually managed data structure, allowing further modifications, such as sorting by marks or filtering based on grades, before presenting the final formatted report.

7.5

A university administration is developing a system to display student marks in a structured and visually appealing format. The goal is to ensure clarity in academic reports by properly aligning names and scores while also demonstrating the use of currency formatting for tuition fees or other financial data. To achieve this, built-in stream manipulators such as endl, setw, setfill, and setprecision must be used to create a neatly formatted output.

The system first reads student names and their corresponding marks, ensuring that the data is aligned in tabular form. Proper spacing between columns enhances readability, preventing misalignment issues caused by varying name lengths or mark values. Additionally, numerical values must be displayed with controlled decimal precision for consistency.

Beyond academic scores, the university also wants to format financial figures, such as tuition fees, in a standardized manner. To achieve this, the system includes a user-defined manipulator, currency(), which prepends a specified currency symbol (₹, $, etc.) before displaying monetary values. This custom formatting ensures that financial data is both readable and professionally presented.

8. Standard Template Library

8.1

A data analytics firm is developing a tool to process numerical sequences efficiently. One of the key requirements is to reverse a given sequence of integers while ensuring optimized performance using modern C++ techniques. The system should allow users to input a sequence of numbers, process the reversal using iterators, and display the transformed output.

To accomplish this, two approaches are explored. The first method leverages the built-in std::reverse() function, which efficiently reverses elements within a dynamically managed sequence. The second approach involves manually implementing the reversal using iterators, providing deeper insight into how iterators navigate and modify data structures.

The system uses a dynamic storage mechanism to handle sequences of varying sizes efficiently. Iterators facilitate traversal and modification, ensuring that elements are manipulated without direct indexing.

8.2

A digital publishing company is developing a tool to analyze text content for word frequency distribution. This tool aims to help writers and editors understand the prominence of specific words in an article, enabling them to refine their content for clarity and impact. The system should process an input sentence, count occurrences of each unique word, and display the results in an organized manner.

To achieve this, the system utilizes an associative container that maps words to their corresponding frequencies. As the text is processed, each word is extracted and stored as a key, while its occurrence count is maintained as the associated value. By leveraging iterators, the system efficiently traverses the data structure, displaying each word along with its computed frequency.

The use of a dynamic mapping approach ensures that words are stored in an ordered manner, allowing for fast retrieval and structured output.

8.3

A financial analytics firm is designing a data processing system to filter out duplicate transaction IDs from large datasets. The system must take an input list of integers, efficiently remove duplicates, and display the unique values in an ordered manner. Given the need for high performance in handling large datasets, the implementation must leverage efficient data structures that automatically eliminate redundancy.

To achieve this, the system employs an associative container that inherently ensures uniqueness while maintaining a sorted order. As integers are added, duplicate entries are automatically discarded, eliminating the need for explicit duplicate-checking logic. Once the unique values are identified, they are either displayed directly using iterators or converted back into a dynamically managed sequence for further processing.

By utilizing iterators, the system efficiently traverses the filtered dataset, ensuring that unique elements are accessed and displayed without additional computational overhead.

8.4

An educational institution is developing a system to rank students based on their scores in an examination. The goal is to efficiently store student names along with their respective scores, sort the records in descending order based on scores, and display the ranked list. The system should be flexible enough to handle varying numbers of students dynamically.

To achieve this, the system associates each student’s name with their score using a structured data representation. A dynamically resizable sequence is chosen to store student records, allowing efficient insertion and retrieval of entries. Each entry consists of a name-score pair, ensuring logical grouping of related information.

Once the data is collected, a sorting operation is performed using a custom comparison function. By leveraging an efficient sorting algorithm, the system ensures that students with the highest scores appear at the top of the list. Iterators are then used to traverse and display the sorted data in a structured format, highlighting rank-based ordering.

8.5

A software development team requires a lightweight directory management system to efficiently organize project files. The system should allow users to create folders, add files to specific folders, and display the directory structure in an organized manner.

Each folder serves as a unique entry, mapping to a list of associated files, ensuring structured storage and easy retrieval. A hierarchical mapping approach is used, where folder names act as keys, and dynamically resizable sequences store the corresponding filenames.

The system provides essential functionalities, including adding new folders, appending files to existing folders, and displaying the complete directory structure. To enhance accessibility, folder names are sorted alphabetically, allowing users to quickly locate specific folders and their contents. Iterators are used for efficient traversal and structured display of data. This approach ensures optimized performance, even for large datasets, while maintaining a logical and organized hierarchy.

9. Error & Exception Handling

9.1

Imagine you're developing a financial calculator for a bank's system that allows users to calculate the ratio of two monetary values they input. This tool must be reliable, as it's part of a crucial decision-making process for loan eligibility.

A bank customer is required to enter two values directly into the system:

1. The first value represents the total loan amount they wish to apply for.

2. The second value represents their annual income.

The system will compute and display the ratio of the loan amount to the income, which helps the bank assess the customer's loan-to-income ratio.

Major challenges that developer will face while developing the system would be, if the customer mistakenly enters non-numerical characters (e.g., "abc"), the system should handle the situation gracefully and prompt the user to correct their input. If the customer enters 0 as their annual income, the system must identify this issue and avoid performing a division that would lead to an invalid result.

9.2

Imagine you are tasked with developing a simple banking system for a local financial institution. The bank has observed an increase in customer complaints regarding unclear transaction records and unhandled errors during operations. They want a robust software module to manage bank accounts efficiently while maintaining a log of transactions.

As a software developer, your job is to create a banking system with the following features:

1. A way to represent individual bank accounts, including the ability to manage their balances.

2. Secure methods to deposit and withdraw funds.

3. An error-handling mechanism to address invalid operations, such as attempting to withdraw more than the account's current balance.

4. A feature that logs every function call when an error occurs to provide insights into the issue.

The bank also insists on maintaining a history of transactions using basic data structures without relying on advanced libraries or containers. Design and implement a banking system that fulfills the above requirements.

****