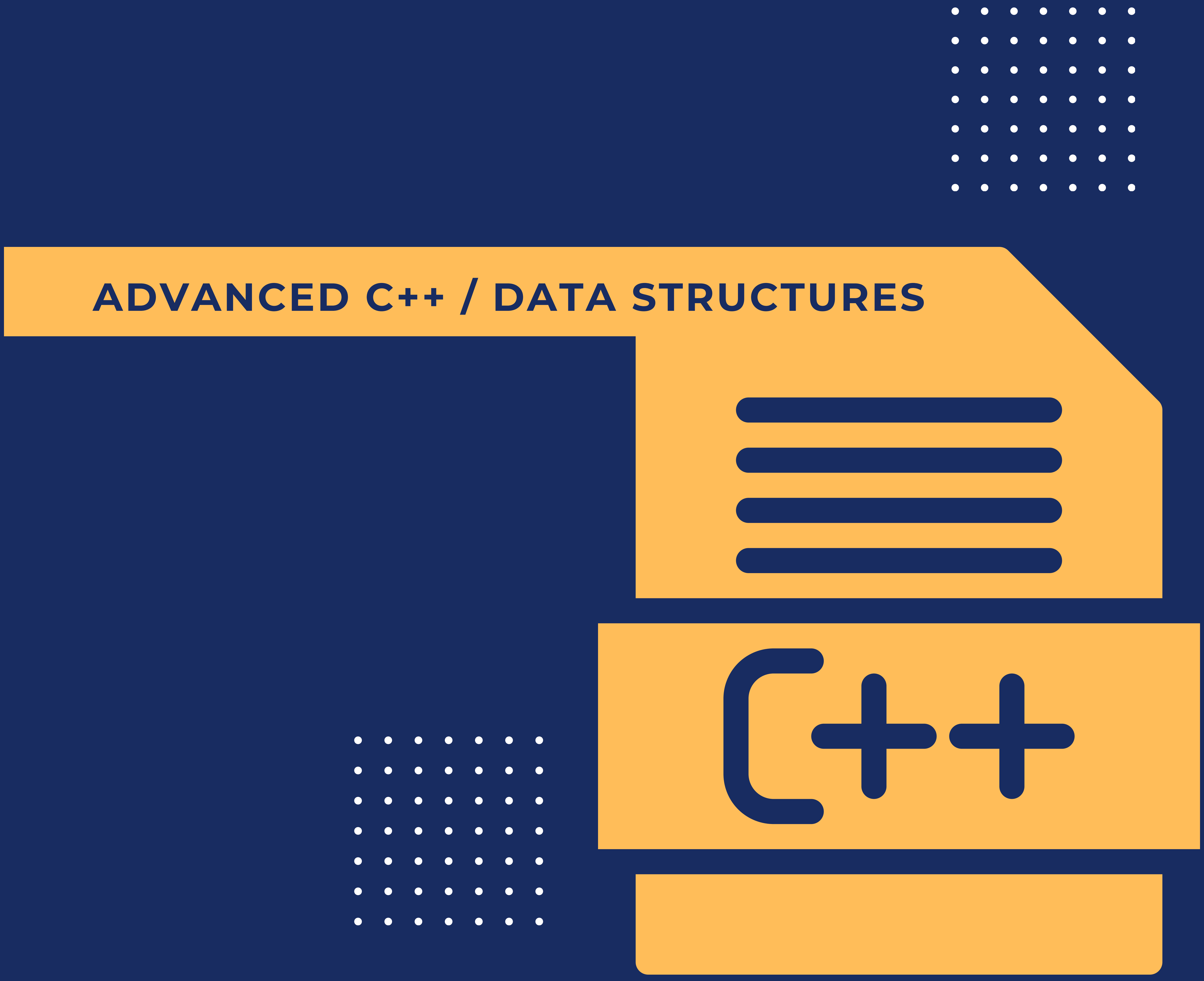
ADVANCED C++ / DATA STRUCUTURES

A clear guide to data structures in C++, showing you how to build efficient programs by mastering the structures and algorithms that power real software.



www.meerkat.pub

DR. KEVIN ROARK | 2026 EDITION

Book Overview

This E-book teaches you the data structures that sit at the heart of modern software, using C++ as the practical tool for building them and working with them confidently. You begin by strengthening the fundamentals on which data structures depend, such as classes and objects, pointers and dynamic memory, templates, and algorithm analysis. From there, you explore the core structures used in real systems: arrays and vectors, linked lists, stacks, queues and deques, sets and maps, trees, graphs, and hash tables, with a focus on what each one is best at and what tradeoffs come with it.

As you move through the chapters, the book emphasizes reasoning and decision-making, not just mechanics. You learn to predict performance using Big-O analysis, choose the proper structure for a task, and understand how design choices affect memory usage and runtime. Searching and sorting strategies are covered in a way that connects directly to the structures they operate on, and you see how small changes in data organization can make significant differences in program efficiency and maintainability.

The latter material ties everything together with recursion and more advanced algorithmic thinking. You see how recursion naturally fits problems involving trees, graphs, and divide-and-conquer approaches, and you practice applying these ideas to build solutions that scale. By the end, you will have a cohesive toolkit of data structures and algorithms, along with the confidence to combine them to solve real programming problems in a clean, professional way.

Key Features of *Introduction to C++*

Clear, Concept-First Explanations - Concepts are introduced in plain, approachable language, helping you understand *why* a data structure or algorithm works before focusing on how to implement it.

Comprehensive Coverage of Core Data Structures - Covers essential structures including arrays, vectors, linked lists, stacks, queues, deques, sets, maps, trees, graphs, and hash tables, with consistent explanations and comparisons throughout.

Strong Emphasis on Algorithmic Thinking - Big-O analysis, performance tradeoffs, and design decisions are woven into every topic. Hence, you develop the ability to choose the proper structure or algorithm for a given problem.

Integrated Recursion and Advanced Algorithms - Recursion is taught as a practical problem-solving tool and connected naturally to trees, graphs, and divide-and-conquer algorithms, reinforcing a deeper understanding.

Real-World Motivated Examples - Examples and demos are grounded in realistic scenarios that mirror how data structures are used in actual software systems, helping you see direct relevance beyond the classroom.

Structured, Progressive Learning Path - Topics build logically from foundational concepts to more advanced material, allowing you to gain confidence steadily without being overwhelmed.

Designed for College-Level Courses - Written to align with data structures curricula, making it suitable for classroom use, self-study, or guided instruction, while reinforcing professional programming practices.

Available online and as an iOS/Android app. Students can easily access the textbook on any device, whether via the web or via iOS or Android apps. The platform remembers where they paused and visually shows their progress through each module, making learning more seamless and encouraging.

Who This Book Is For

This book is for learners who have completed an introductory programming course and are ready to move beyond basic syntax into deeper problem-solving with data structures and algorithms. If you are comfortable writing programs in C++ and want to understand how real software organizes data efficiently, this book is designed for you.

It is well-suited for college-level data structures courses, transfer-oriented computer science programs, and anyone preparing for upper-division coursework. The material supports those who want a clear, structured explanation of core concepts, emphasizing reasoning, performance, and sound design practices over memorization.

This book is also a strong fit for self-directed learners and career-focused programmers who want to strengthen their technical foundation. If your goal is to write cleaner, faster, and more scalable programs, or to prepare for technical interviews and advanced algorithms courses, this book provides the depth and guidance needed to build that confidence.

Course/Textbook Overview

Module 1 Overview: C++ Review

This module provides a focused refresher on essential C++ programming concepts needed for success in more advanced coursework. You'll revisit how C++ programs are structured using multiple files, review core syntax and data types, and practice using control structures to guide program flow. The module also reinforces effective use of functions, structures, and the Standard Template Library, emphasizing clean organization, readable code, and safe input handling. Through hands-on labs and a real-world mini project, you'll apply these fundamentals in practical scenarios that mirror professional programming practices and prepare you for more complex data structures and algorithms.

By the end of this module, you will be able to:

- Organize a C++ program using header and source files, clearly separating interface and implementation.
- Use core C++ data types, constants, casting, and type inference correctly and intentionally.
- Control program flow using conditionals, loops, and range-based iteration over collections.
- Design and use functions with appropriate parameter passing, including pass-by-value, pass-by-reference, and const references.
- Create and process structured data using struct, vector, iterators, and formatted input/output to produce clear, reliable program output.

Module 2 Overview: Classes and Objects

In this module, you are introduced to one of the most essential concepts in modern software development: classes and objects. Up to this point, programs have likely been written using individual variables and standalone functions. While that approach works for small programs, it quickly becomes challenging to manage as software grows in size and complexity. Classes provide a way to organize related data and behavior into a single, meaningful unit that mirrors how real-world systems are designed.

By learning how to design and use classes, you move toward object-oriented programming, a paradigm used extensively in professional software development. You will see how classes act as blueprints, how objects are created from those blueprints, and how encapsulation helps protect and manage data safely. Through hands-on examples, labs, and demos, you will build classes that include constructors, access control, member functions, operator overloading, and proper file organization. This module lays the foundation for more advanced topics later in the course, including inheritance, polymorphism, and large-scale program design.

By the end of this module, you will be able to:

- Explain the difference between a class and an object and describe how classes model real-world entities in object-oriented programming.
- Design a C++ class that uses appropriate access modifiers (public, private, and protected) to enforce encapsulation and data protection.
- Implement constructors and member functions to initialize objects and define meaningful behaviors.
- Organize class definitions and implementations using separate header and implementation files following professional coding standards.
- Use operator overloading, specifically the stream insertion operator (<<), to produce clean, readable output for objects and improve program usability.

Module 3 Overview: Pointers and Memory Management

In this module, you explore how C++ programs work directly with memory through pointers and dynamic allocation. You begin by learning what pointers are and how they relate to memory addresses, then move into managing memory on the heap using new and delete. The module builds from raw pointers to modern C++ features, such as smart pointers, and shows how each approach affects ownership, object lifetime, and program safety. Through concept pages, demos, and hands-on labs, you gain a practical understanding of how memory is allocated, shared, and released, and why careful memory management is critical for building efficient and reliable software.

By the end of this module, you will be able to:

- Explain what pointers are and how they reference memory locations in a program.
- Distinguish between stack and heap memory and identify when dynamic memory allocation is appropriate.
- Use raw pointers to access and modify data safely, including proper allocation and deallocation with new and delete.
- Apply pointer-safety techniques, such as nullptr checks, to avoid common runtime errors.
- Compare reference variables and pointers, and choose the appropriate tool for a given situation.
- Use smart pointers (unique_ptr, shared_ptr, and weak_ptr) to manage ownership and object lifetimes effectively.
- Identify common memory management problems, including memory leaks and dangling pointers, and explain strategies to prevent them in real-world programs.

Module 4 Overview: Inheritance and Polymorphism

This module introduces inheritance and polymorphism, two foundational concepts of object-oriented programming that allow software to be designed in a structured, extensible, and maintainable way. You learn how base classes define shared behavior and how derived classes specialize that behavior without duplicating code. Through hands-on examples and real-world scenarios, this module shows how polymorphism enables programs to work with different object types through a standard interface while still executing the correct behavior at runtime. By the end of the module, you will understand how inheritance hierarchies, virtual functions, abstract classes, and proper resource management work together to support professional-quality software design.

By the end of this module, you will be able to:

- Explain how inheritance is used to model “is-a” relationships between classes.
- Implement polymorphism using virtual functions and base-class pointers or references.
- Design and use abstract classes with pure virtual functions to enforce consistent interfaces.
- Apply best practices for memory management, including virtual destructors and smart pointers.
- Recognize how inheritance and polymorphism are applied in real-world systems and common object-oriented design patterns.

Module 5 Overview: Templates

Templates are one of the most powerful features in C++, allowing you to write code that works with many data types without duplication. Instead of writing separate functions or classes for integers, doubles, strings, and other types, templates let you define the logic once and let the compiler generate the type-specific versions when needed.

In this module, you’ll learn how templates work, why they are essential to modern C++ programming, and how they are used behind the scenes in the Standard Template Library (STL). Through hands-on examples and real-world case studies, you’ll see how templates enable reusable data structures, generic algorithms, and type-safe code. By the end of the module, you’ll understand not only how to use templates, but also how to design your own template-based solutions responsibly and effectively.

By the end of this module, you will be able to:

- Explain what templates are and why they are used in C++
- Write function templates that operate on multiple data types
- Create class templates, including templates with multiple type parameters
- Apply function template specialization for type-specific behavior
- Describe how templates are compiled and instantiated by the compiler
- Identify real-world uses of templates, including examples from the STL
- Follow best practices and avoid common pitfalls when working with templates

Module 6 Overview: Standard Template Library Containers

In this module, you are introduced to the Standard Template Library (STL) containers in C++. These containers provide ready-made, well-tested data structures that enable efficient storage, organization, and access of data without building everything from scratch. Rather than focusing on how data structures are implemented internally, this module emphasizes when and why to use each container based on the problem at hand.

Through explanations, demos, labs, and reflection, you explore how different containers support different access patterns, performance characteristics, and use cases. By the end of the module, you should feel comfortable selecting an appropriate STL container and using it effectively in a small real-world application. This skill is essential for writing clean, efficient, and maintainable C++ programs.

By the end of this module, you will be able to:

- Identify and describe the purpose and key characteristics of common STL containers, including vector, list, deque, map, multimap, and unordered_map.
- Choose an appropriate STL container based on application requirements such as insertion and removal patterns, ordering needs, and lookup performance.
- Use STL containers in C++ programs to add, remove, search, and display data using container-specific operations.
- Explain trade-offs between containers, including differences in performance, ordering, and access patterns.
- Design and implement a small application that uses an STL container effectively, and justify the choice of container.

Module 7 Overview: Algorithm Analysis and Searching

In this module, you will be introduced to the fundamental concepts of algorithmic efficiency and to how programmers evaluate the performance of their solutions. Rather than focusing solely on whether a program works, this module emphasizes *how well* it performs as data sizes increase. You will learn how to reason about time and space usage using Big-O notation and apply these ideas to common searching algorithms.

The module then transitions from theory to practice by exploring two core search techniques: linear search and binary search. Through hands-on demos and labs, you will see how different algorithms behave under real conditions and why choosing the right approach can dramatically affect performance. By the end of this module, you will be able to analyze, compare, and justify search algorithm choices based on efficiency rather than guesswork.

By the end of these sections, you will be able to:

- Explain why algorithm analysis is important when working with large datasets and real-world applications.
- Interpret Big-O notation and identify common time and space complexities.
- Compare linear search and binary search with respect to efficiency, requirements, and use cases.
- Implement and evaluate searching algorithms while measuring and analyzing comparison counts.
- Select an appropriate search algorithm based on data size, ordering, and performance constraints.

Module 8 Overview: Sorting Algorithms

In this module, you will explore how sorting algorithms organize data into meaningful order and why this process is essential in nearly every area of computing. You will begin with simple, comparison-based algorithms that focus on the mechanics of swapping and positioning values, then progress to more efficient, divide-and-conquer approaches used in real-world systems. Along the way, you will also learn how the C++ Standard Library provides highly optimized sorting tools that are preferred in professional software development.

The purpose of this module is to help you understand both how sorting works and when to choose a particular approach. By implementing multiple sorting algorithms, analyzing their performance, and applying them to real-world data structures, you will develop a deeper appreciation of algorithmic efficiency, stability, and trade-offs between time and memory usage. These skills are foundational to subsequent topics, such as search, data analysis, and performance optimization, and mirror the decision-making process used by software engineers in production environments.

By the end of this module, you will be able to:

- Explain why sorting is a foundational operation in computer science and how it supports searching, reporting, and data analysis.
- Implement and trace multiple sorting algorithms, including simple quadratic sorts and more efficient divide-and-conquer approaches.
- Compare sorting algorithms based on time complexity, space usage, stability, and real-world performance characteristics.
- Apply the C++ Standard Library's functions to organize primitive data and custom structures.
- Evaluate and justify the choice of a sorting algorithm for a given problem based on input size, data characteristics, and required stability.

Module 9 Overview: Linked Lists

In this module, you will explore linked lists, a foundational dynamic data structure in computer science. Unlike arrays and vectors, linked lists store elements as connected nodes, allowing the structure to grow and shrink efficiently at runtime. This module begins by introducing singly linked lists and gradually expands to more advanced variations, including doubly linked lists, circular linked lists, and the C++ Standard Library's list container. Through hands-on demos and labs, you will learn how linked lists manage memory, how nodes are connected and traversed, and how different list designs affect performance and use cases. By the end of the module, you will be able to choose when a linked list is the proper data structure and implement or use one confidently in C++.

By the end of this module, you will be able to:

- Explain how linked lists differ from arrays and vectors in terms of memory layout and performance trade-offs.
- Implement and traverse singly, doubly, and circular linked lists using dynamic memory.
- Perform common linked-list operations, including insertion, deletion, search, and traversal.
- Use the STL list container effectively, including iterators and built-in operations.
- Evaluate real-world scenarios to determine when a linked list is the most appropriate data structure.

Module 10 Overview: Stacks and Queues

In this module, you will explore stacks and queues, two fundamental linear data structures that control how data is stored, accessed, and processed. Unlike arrays or vectors, stacks and queues enforce specific access rules - Last-In, First-Out (LIFO) for stacks and First-In, First-Out (FIFO) for queues, which makes them ideal for managing history, task scheduling, buffering, and event handling. Through a combination of conceptual discussions, hands-on labs, and real-world simulations, you will learn how these structures operate internally and how they are applied in everyday software systems, including web browsers, operating systems, customer service systems, and priority-based scheduling.

By the end of this module, you will not only understand how stacks and queues work, but also when and why they should be used. This foundation prepares you for more advanced data structures and algorithms later in the course, where these concepts play a critical role in recursion, graph traversal, scheduling, and system design.

By the end of this module, you will be able to:

- Explain the core principles of LIFO and FIFO and describe how stacks and queues differ from other linear data structures.
- Implement stacks and queues using both manual approaches (arrays and linked lists) and STL containers (stack, queue, deque, and priority_queue).
- Apply stacks to solve real-world problems such as browser navigation, undo operations, and expression evaluation.
- Apply queues to model real-world systems, including customer lines, print queues, task scheduling, and message processing.
- Analyze the performance characteristics of stack and queue operations and identify the scenarios in which each structure is most appropriate.
- Extend basic stack and queue implementations by introducing additional features such as forward navigation or priority-based processing, demonstrating a deeper understanding of data flow and state management.

Module 11 Overview: Recursion

This module introduces recursion as a fundamental problem-solving technique in computer science. The purpose of this module is to help students understand how recursive functions work, how they differ from iterative solutions, and why recursion is especially effective for problems that naturally break into smaller subproblems. Through conceptual explanations, hands-on examples, and applied demonstrations using linked lists and searching algorithms, students develop a clear mental model of recursion and the call stack. This module prepares students for more advanced recursive structures and algorithms, including tree traversal and divide-and-conquer techniques, which are covered in later modules.

By the end of this module, you will be able to:

- Explain the concept of recursion, including the roles of the base case and the recursive case.
- Trace recursive function calls and returns by understanding how the call stack grows and unwinds.
- Compare recursive and iterative solutions and evaluate when each approach is appropriate.
- Implement recursive algorithms for common tasks, such as searching, linked-list traversal, counting, and summing values.
- Apply recursive thinking to prepare for advanced data structures and algorithms, including trees and divide-and-conquer methods.

Module 12 Overview: Sets

In this module, you will explore the concept of sets and how they are used to store and manage collections of unique elements in C++. The module begins with the mathematical foundations of sets and gradually transitions into practical applications using the C++ Standard Template Library. You will learn how sets automatically prevent duplicates, maintain order, and support efficient searching and comparison operations.

As the module progresses, you will work with different types of sets, including ordered sets, unordered sets, and multisets. You will also learn how to perform classic set operations such as union, intersection, and difference, and when to use sets rather than other containers, such as vectors, lists, and maps. Through demonstrations, labs, and real-world examples, this module emphasizes not only *how* sets function but also why they are essential tools for writing efficient, well-structured programs.

By the end of this module, you will be able to:

- Explain the mathematical concept of a set and how it applies to programming.
- Use C++ set, unordered_set, and multiset containers appropriately.
- Perform set operations, including union, intersection, and difference, using STL algorithms.
- Compare ordered and unordered sets with respect to structure, performance, and use cases.
- Select between sets and other containers (vectors, lists, maps) based on data requirements.
- Analyze real-world scenarios and identify when enforcing uniqueness is necessary.
- Design and describe set-based solutions that avoid duplicative data.

Module 13 Overview: Trees

In this module, you will explore trees, a fundamental data structure for representing hierarchical relationships and organizing data efficiently. Unlike linear structures such as arrays or linked lists, trees allow data to branch, making them well-suited for modeling real-world systems, including file directories, organizational charts, and decision-making processes. This module builds your understanding of how trees store, manage, and retrieve information in a structured and efficient way.

You will begin with the core concepts and terminology of trees, including nodes, roots, parents, children, and leaves. From there, you will study binary trees and binary search trees (BSTs), learning how ordering rules enable faster search than linear data structures. You will also examine tree traversals—inorder, preorder, and postorder—and understand how different traversal strategies are used for sorting, copying, and evaluating tree-based data.

As the module progresses, you will learn why tree balance is critical to performance. You will explore AVL trees, which automatically maintain balance through rotations, ensuring logarithmic time complexity for search, insertion, and deletion. You will then shift focus to heaps, a specialized tree structure designed for priority-based access, and examine how they underpin systems such as priority queues and scheduling algorithms.

Finally, you will connect these concepts to real-world programming by using C++ Standard Library tree-like containers, including set and map. These containers abstract complex tree behavior through balanced red-black tree implementations, allowing you to leverage efficient, ordered data structures without managing rotations or pointers manually.

By the end of this module, you will understand not only how different tree structures work, but also when and why to use them. You will gain the ability to choose the appropriate tree-based structure for a given problem, analyze its performance characteristics, and apply these concepts to practical, real-world applications in software development.

By the end of this module, you will be able to:

- Explain tree terminology and structure, including nodes, roots, parents, children, leaves, height, and depth, and describe how trees differ from linear data structures.
- Implement and traverse binary trees and binary search trees, applying inorder, preorder, and postorder traversal strategies to process tree data effectively.
- Analyze the impact of tree balance on performance, and explain how self-balancing trees such as AVL trees maintain efficient O(log n) search, insertion, and deletion operations.
- Apply heap-based structures to solve priority-driven problems, demonstrating how percolate-up and percolate-down operations maintain heap properties.
- Use C++ STL tree-like containers (set and map) to store, search, and retrieve ordered data efficiently, and compare their performance and use cases with custom tree implementations.

Module 14 Overview: Graphs

This module introduces graphs, one of the most powerful and flexible data structures in computer science for modeling relationships, networks, and connections. Unlike linear data structures, graphs represent complex systems in which data points are linked in multiple ways, such as social networks, transportation systems, communication networks, and dependency structures.

Throughout this module, you will learn how graphs are structured, how they are stored in memory, and how they can be explored and analyzed using fundamental graph algorithms. You will apply these concepts through practical examples, including social media networks, flight connections, delivery routes, and city infrastructure planning. By the end of the module, you will understand not only how graph algorithms work, but also why they matter and how they are used to solve real-world problems efficiently.

By the end of this module, you will be able to:

- Explain core graph concepts and terminology - Describe vertices, edges, paths, cycles, and distinguish between directed, undirected, weighted, and unweighted graphs.
- Represent graphs using common data structures - Implement graph representations using adjacency lists and adjacency matrices, and explain the tradeoffs between them.
- Traverse graphs using BFS and DFS - Apply Breadth-First Search and Depth-First Search to explore and analyze graph structures, and interpret the order in which nodes are visited.
- Apply graph algorithms to solve practical problems - Use Dijkstra's Algorithm to find shortest paths and Prim's or Kruskal's Algorithms to build minimum spanning trees in real-world scenarios.
- Select appropriate graph techniques for a given problem - Analyze a problem domain and determine which graph type and algorithm best fits the situation, justifying your choices

Module 15 Overview: Hash Maps

The purpose of this module is to introduce hash maps as one of the most efficient and widely used data structures in computer science. Hash maps enable programs to store and retrieve data quickly by using keys and hash functions, rather than relying on sequential search or ordered structures. This module focuses on helping you understand not only how hash maps work, but also why they are essential in real-world software systems.

Throughout this module, you will explore the core components of hashing, including hash functions, keys, collisions, and load factor. You will learn why collisions are unavoidable, how different collision-handling strategies work, and how chaining allows a hash table to remain reliable even when many items map to the same bucket. These concepts are reinforced through practical demos and labs that show how hashing behaves as data grows.

This module also connects theory to practice by introducing the C++ Standard Library containers unordered_map and unordered_set. By understanding how these containers operate internally, you will be better prepared to use them effectively and to make informed decisions about when hash-based structures are appropriate. By the end of the module, you will have a strong conceptual and practical foundation in hash maps that prepares you for advanced topics in performance optimization, system design, and large-scale data management.

By the end of this module, you will be able to:

- Explain how hashing works by describing the relationship between keys, hash functions, and bucket indices in a hash table.
- Identify and handle collisions by comparing chaining and open addressing, and explaining how chaining resolves collisions in practice.
- Analyze hash table performance by describing the impact of load factor, collisions, and rehashing on average and worst-case efficiency.
- Use C++ hash-based containers effectively by applying unordered_map and unordered_set to store and retrieve data in real-world scenarios.
- Evaluate when to use hash maps by comparing hash tables with tree-based and sequential data structures based on performance and ordering requirements.

Module 16 Overview: Advanced Algorithms

This module builds upon the data structures and foundational algorithms covered earlier in the course by introducing higher-level problem-solving strategies used in real-world software systems. Rather than focusing on individual data structures in isolation, this module emphasizes *how* different algorithmic approaches are selected, combined, and applied to solve complex problems efficiently.

You will explore several widely used algorithmic strategies, including greedy algorithms, dynamic programming, backtracking, branch-and-bound, and heuristic methods. Each strategy is presented with practical examples that highlight its strengths, limitations, and appropriate use cases. These approaches help explain how developers tackle problems involving optimization, search, decision making, and constraint satisfaction.

The module also introduces iconic real-world problems and applications, such as the Traveling Salesperson Problem and Huffman compression. These examples demonstrate how advanced algorithms rely on core data structures like arrays, trees, graphs, priority queues, and recursion to produce efficient and scalable solutions.

By the end of this module, you will have a clearer understanding of how advanced algorithms extend beyond basic sorting and searching, and how thoughtful algorithm selection plays a critical role in software performance, scalability, and maintainability. This module serves as a capstone, helping you connect everything learned throughout the course into a cohesive problem-solving toolkit used by professional software developers.

By the end of this module, you will be able to:

- Explain common advanced algorithm strategies, including greedy algorithms, dynamic programming, backtracking, branch and bound, and heuristic approaches, and describe when each strategy is most appropriate.
- Apply advanced algorithmic techniques to solve optimization and search problems, such as routing and scheduling scenarios.
- Analyze and compare algorithm tradeoffs, including correctness, efficiency, and practicality, when choosing between exact and approximate solutions.
- Demonstrate how data structures support advanced algorithms by identifying the role of structures such as trees, graphs, priority queues, arrays, and recursion stacks.
- Evaluate real-world applications of advanced algorithms, including compression and routing problems, and explain how algorithm selection impacts performance and scalability.