College of Engineering & Applied Sciences

CSPB 2270

Computer Science 2: Data Structures

Taylor Larrechea

2023

Sections

CSPB 2270 Details

Below is the class description for CSPB 2270 - Computer Science 2: Data Structures. The class description for when the class was offered may be slightly different from the description of the course website. The program website is found at **CU Boulder Applied Computer Science**. The program curriculum can be found at **B.S. ACS Curriculum** and the CSPB 2270 - Computer Science 2: Data Structurescourse description can be found at **CSPB 2270 Course Description**.

Class Description

CSPB 2270 - Computer Science 2: Data Structures - Prerequisites: **CSPB 1300** Credits: **4**

1.0.1 Brief Description of Course Content

Studies data abstractions (e.g., stacks, queues, lists, trees) and their representation techniques (e.g., linking, arrays). Introduces concepts used in algorithm design and analysis including criteria for selecting data structures to fit their applications.

Topics include data and program representations, computer organization effect on performance and mechanisms used for program isolation and memory management.

1.0.2 Specific Goals

Below are some specific goals of CSPB 2270 - Computer Science 2: Data Structures. The first specific goal pertains to Specific Outcomes of Instruction.

Specific Outcomes of Instruction

The following are the Specific Outcomes of Instruction for CSPB 2270 - Computer Science 2: Data Structures.

- Document code including precondition/postcondition contracts for functions and invariants

for classes.

- Determine quadratic, linear and logarithmic running time behavior in simple algorithms, write big-O expressions to describe this behavior, and state the running time behaviors for all basic operations on the data structures presented in the course.
- Create and recognize appropriate test data for simple problems, including testing boundary conditions and creating/running test cases, and writing simple interactive test programs to test any newly implemented class.
- Define basic data types (vector, stack, queue, priority queue, map, list).
- Specify, design and test new classes using the principle of information hiding for the following data structures: array-based collections (including dynamic arrays), list-based collections (singly-linked lists, doubly-linked lists, circular-linked lists), stacks, queues, priority queues, binary search trees, heaps, hash tables, graphs (e.g. for depth-first and breadth-first search), and at least one balanced search tree.
- Be able to describe how basic data types are stored in memory (sequential or distributed), predict what may happen when they exceed those bounds.
- Correctly use and manipulate pointer variables to change variables and build dynamic data structures.
- Determine an appropriate data structure for given problems.
- Follow, explain, trace, and be able to implement standard computer science algorithms using standard data types, such as a stack-based evaluation of arithmetic expressions or a traversal of a graph.
- Recognize situations in which a subtask is nothing more than a simpler version of the larger problem and design recursive solutions for these problems.
- Follow, explain, trace, and be able to implement binary search and a variety of quadratic sorting algorithms including mergesort, quicksort and heapsort.

Next is a Brief List of Topics to be Covered for CSPB 2270 - Computer Science 2: Data Structures.

Brief List of Topics to be Covered

The following is a Brief List of Topics to be Covered for CSPB 2270 - Computer Science 2: Data Structures.

- Cost of algorithms and Big O notation.
- Memory and pointers, structs, and dynamic memory allocation.
- Linked lists, stacks and queues.
- Trees: Binary trees, binary search trees, tree traversal, recursion.
- Tree balancing: red-black trees.
- Graphs: graph traversal algorithms, depth-first and breadth-first search.
- Hash tables, hash functions, collision resolution algorithms.
- Algorithms for sorting, such as insertion sort, bubble sort, quick sort, and merge sort.

Lastly, the following is a list of Mathematical Concepts Used for CSPB 2270 - Computer Science 2:

Data Structures.

Mathematical Concepts Used

The following is a brief list of Mathematical Concepts Used in CSPB 2270 - Computer Science 2: Data Structures.

- Logarithms
- Big O
- Recursion
- Trees
- Graphs

1.0.3 Instructor Information

The following are the details of this courses instructor. This course was given for the Summer term of 2023.

- Name: Dr. Frank Jones
- Email: francis.jones@colorado.edu
- Office Hours:
  - Moddays: 7:00 PM - 8:00 PM MT
  - Wednesdays: 1:00 PM - 2:00 PM MT
  - By Appointment

1.0.4 Important Dates

The following are important dates for this course. This course runs from May 22, 2023 - August 18, 2023.

Assessment	Date
Exam 1	July 7th, 10 AM - 10 PM MT
Exam 2	August 4th, 10 AM - 10 PM MT
Interview Grade for Linked List Assignment	June 14-16
Interview Grade for Sorting Assignment	June 5-7
Interview Grade for Graph Assignment	July 26-28
Final Project	Aug 14-15
Quizzes	Usually Due on Mondays
Programming Assignments	Usually Due on Tuesdays
Assignment Interviews	Usually Held on Wednesdays & Thursdays

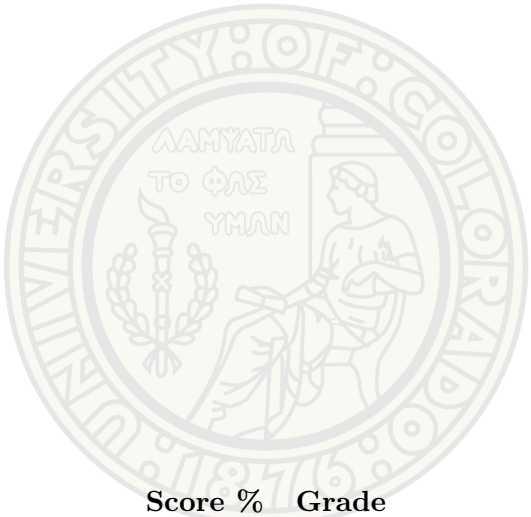
1.0.5 Grade Breakdown

The following consists of a grade breakdown for this class.

1.0.6 Grading Scale

The following is how grades will be assigned for this class.

Item	Percent of Grade	Notes
Reading Quizzes	10	Assignments From Textbook
Programming Assignments (10)	25	Autograded
Assignment Interviews (3)	15	Interviews Asking About Assignments
Exams (2)	30	Myriad of Types of Questions
Final Project	20	Entire Grade is Interview Based



Score %	Grade
93 - 100	A
90 - 93	A-
87 - 90	B+
83 - 87	B
80 - 83	B-
77 - 80	C+
73 - 77	C
70 - 73	C-
67 - 70	D+
63 - 67	D
60 - 63	D-

Sections

Week 1

C++ Review, Debugging, Unit Testing

2.0.1 Activities

The following are the activities that are planned for Week 1 of this course.

- Take the C++ assessment
- Read the C++ refresher or access other resources to improve your skills (book activities are graded but the grades are not included in your final grade for this course)
- Read the zyBook chapter(s) assigned and complete the reading quiz(s) by next Monday
- Access the GitHub Classroom and get your Assignment-0 repository created, cloned, edited, and graded by next Tuesday
- Watch the videos for Cloning GitHub Classroom Assignments, Setting up an IDE in Jupytherhub, and Unit Testing

2.0.2 Lectures

Here are the lectures that can be found for this week:

- Course Concepts
- GitHub Classroom
- GitHub Security
- Accepting an Assignment
- Accessing Git Files

- Cloning Into JupyterHub
- VSCode in JupyterHub
- Multi File Programming
- Unit Testing Basics

2.0.3 Programming Assignment

The programming assignment for Week 1 - **Using GitHub and GitHub Classroom**.

2.0.4 Notes

The first chapter of this week was Chapter 1 - Introduction to Data Structures.

Sec. 1.1 - Data Structures

We define Data Structures to be the following:

- A data structure is a method of organizing, storing, and performing operations on data.
- Operations performed on data structures include accessing or updating stored data, searching for specific data, inserting new data, and removing data.
- Understanding data structures is crucial for effectively managing and manipulating data.

To summarize, data structures are methods of organizing, storing, and manipulating data, including arrays, linked lists, stacks, queues, trees, graphs, hash tables, and heaps.

Basic Data Structures

Below are some examples of basic data structures:

Arrays - Sequential collections of elements with efficient access and modification.

- Sequential collection of elements with unique indices. Indexes from zero.
- Efficient access and modification of elements at specific locations.
- Less efficient for inserting or removing elements in the middle.

Linked Lists - Chain of nodes allowing efficient insertion and removal.

- Chain of nodes where each node contains data and a reference to the next node.
- Efficient insertion and removal of elements.
- Sequential traversal required for access specific elements.

Stacks - Follows Last-In-First-Out (LIFO) principle for efficient insertion and removal from the top.

- Follows Last-In-First-Out (LIFO) principle.
- Insert and remove elements from the top of the stack.
- Useful for tasks like function class and undo operations.

Queues - Follows First-In-First-Out (FIFO) principle for efficient insertion, and removal from the front and rear.

- Follows First-In-First-Out (FIFO) principle.
- Insert elements at the rear and remove elements from the front.
- Useful for tasks like process scheduling.

Trees - Hierarchical structure for enabling efficient searching, insertion, and deletion.

- Hierarchical structure consisting of nodes connected by edges.
- Efficient searching, insertion, and deletion operations.
- Suitable for organizing file systems or representing hierarchical relationships.

Graphs - Collection of nodes connected by edges, useful for representing complex relationships.

- Collection of nodes connected by edges.
- Each node can have multiple connections.
- Used to represent complex relationships like social networks or computer networks.

Hash Tables - Data structure that uses hashing for efficient insertion, retrieval, and deletion of key-value pairs.

- Efficient data structure using hashing for fast key-value pair operations.
- Uses a hash function to convert keys into indices.
- Provides quick access to elements and handes collisions for proper storage.

Heaps - Binary-tree based structure that ensures efficient retrieval of the minimum or maximum element.

- Binary tree-based structure for efficient retrieval of minimum or maximum element.
- Maintains a partial order property, such as the min-heap or max-heap property.
- Supports fast insertion and deletion of elements while preserving the heap property.

In the study of data structures, we explore various methods of organizing, storing, and manipulating data.

Arrays are sequential collections of elements, allowing efficient access and modification. Linked Lists from a chain of nodes, facilitating efficient insertion and removal. Stacks follow the Last-In-First-Out principle and are useful tasks like function calls and undo opertions. Queues follow the First-In-First-Out principle and are suitable for process scheduling.

Trees, consisting of nodes connected by edges, provide a hierarchical organization, enabling efficient searching, insertion, and deletion. Graphs are collections of nodes connected by edges and represent complex relationships like social networks or computer networks.

Hash Tables emply hashing for efficient insertion, retrieval, and deletion of key-value pairs. They use a hash function to convert keys into indices, providing fast access to elements while handling collisions.

Heaps, based on binary trees, allow efficient retrieval of the minimum or maximum element. They maintain a partial order property and support fast insertion and deletion while preserving the heap property.

Understanding these data structures and their characteristics is essential for problem-solving and designing

efficient algorithms in data-oriented scenarios.

Sec. 1.2 - Abstract Data Types

Abstract Data Types (ADT) can be summarized as:

- Abstract Data Types (ADTs) define a set of operations and behavior for manipulating data without specifying the implementation details.
- ADTs provide a logical representation of data and operations, focusing on the "what" rather than the "how" of data manipulation.
- ADTs promote code abstraction and modularity, allowing for reusable and maintable code by encapsulating dtat and providing a clear interference for interaction.

To summarize, Abstract Data Types (ADTs) provide a high-level, logical representation of data and operations, focusing on the "what" rather than the "how", enabling code abstraction and modularity for reusable and maintable programming.

Common ADTs

Below are some examples of common ADTs:

List - A basic data structure that represents an ordered collection of elements, allowing for efficient insertion, deletion, and retrieval operations.

- Lists are a versatile data structure that can store elements of any type and maintain their order, allowing for easy access and modification.
- They offere efficient insertion and deletion operations at both ends, making them suitable for scenarios where elements need to be dynamically added or removed.
- Lists can be implemented using various techniques such as arrays or linked lists, each with its own trade-offs in terms of memory usage and performance.

Dynamic Array - A dynamic array is a resizable data structure that provides the flexibility to dynamically adjust its size to accommodate the changing needs of a program.

- Dynamic arrays are resizable data structures that can grow or shrink in size based on the program's needs, allowing for efficient memory management.
- They provide the benefits of random access like traditional arrays, enabling constant-time access to elements using indices.
- Dynamic arrays allocate contiguous memory blocks, and when the array size exceeds its capacity, a larger memory block is allocated, and elements are copied over, ensuring efficient insertion and deletion operations while maintaining order.

Stack - A stack is a Last-In-First-Out (LIFO) data structure that allows efficient insertion and removal of elements from one end, commonly used in scenarios involving function calls, memory management, and undo operations.

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, where the last element added is the first one to be removed.
- It supports two primary operations, push, which adds an element to the top of the stack, and pop, which removes the top most element from the stack.
- Stacks are commonly used in tasks that require tracking function calls, managing memory, and undo operations, providing efficient insertion and deletion of elements from a single end.

Queue - A queue is a Firs-In-First-Out (FIFO) data structure that enables efficient insertion at one end and removal at the other, commonly used for managing processes, task scheduling, and breadth-first search algorithms.

- Queues follow the First-In-First-Out (FIFO) principle, ensuring that the element inserted first is the first one to be removed.
- They support two primary operations: enqueue, which adds an element to the rear of the queue, and dequeue, which removes the element from the front.
- Queues are frequently utilized for process management, task scheduling, and breadth-first search algorithms, as they maintain the order of elements and provide efficient insertion and removal at both ends.

Deque - A deque (double-ended queue) is a data structure that allows efficient insertion and removal of elements at both ends, providing flexibility in managing data from the front or the rear.

- Deques support insertion and removal of elements at both ends, allowing for efficient operations at the front and rear of the data structure.
- They provide flexibility in managing data by enabling operations like push and pop at both ends, as well as accessing elements from either end.
- Deques are useful in scenarios where elements need to be added or removed from both ends, such as implementing algorithms like breadth-first search, implementing a queue with additional functionalities, or managing a sliding window in algorithms like dynamic programming.

Bag - A bad, also known as a multiset or a collection, is an unordered data structure that allows storing multiple occurrences of elements, providing efficient insertion and retrieval operations.

- Bags allow for the insertion of elements without enforcing any particular order, making them suitable for scenarios where maintaining the order is not necessary.
- Unlike other data structures, bags can store duplicate elements, allowing for multiple occurrences of the same item.
- Bags are commonly used when it is important to count or track the frequency of elements, such as in data analytics text processing, or certain types of machine learning algorithms.

Set - A set is an unordered data structure that stores a collection of unique elements, providing efficient membership testing and set operations.

- Sets contain only unique elements, ensuring that duplicates are automatically removed, making the suitable for tasks that require uniqueness, such as maintaining a distinct list of items.
- Sets provide efficient membership testing, allowing for quick checks to determine if an element is present or absent.
- Sets support common set operations like union, intersection, and difference, enabling efficient manipulation and comparison of multiple sets, often used in tasks like data deduplication, finding common elements, or checking for similarities across multiple datasets.

Priority Queue - A priority queue is an abstract data type that stores elements with associated priorities, allowing efficient retrieval of the highest priority element.

- Priority queues store elements with priorities, where the element with the highest priority can be efficiently retrieved.
- Elements in a priority queue are typically ordered on their priority, allowing for operations such as insertion and removal according to their priority level.

- Priority queues are commonly used in various applications like task scheduling, event-driven simulations, graph algorithms, and data compression, where efficient handling of elements based on their priority is essential for optimizing performance.

Dictionary (Map) - A dicitionary, also known as a map or associative array, is a data structure that stores key-value pairs, providing efficient lookup and retrieval of values based on their associated keys.

- Dicitionaries store key-value pairs, allowing efficient retrieval of values based on their associated keys.
- Keys in a dictionary are unique, enabling fast and direct access to the corresponding values.
- Dicitionaries are commonly used in situations that require fast lookup, such as data indexing, caching, symbol tables, and implementing algorithms like graph traversal or dynamic programming.

Lists provide ordered collections of elements with efficient insertion, deletion, and retrieval operations. They offer flexibility in managing data and are widely used in various applications that require maintaining a specific order. Dynamic arrays, on the other hand, offer resizable storage that adjusts to the needs of the program. They provide random access to elements and efficient memory management by reallocating memory blocks as the array size changes.

Stacks adhere to the Last-In-First-Out (LIFO) principle and are commonly used for tracking function calls, managing memory, and implementing undo operations. They offer efficient insertion and removal of elements from one end. Queues, on the other hand, follow the Firs-In-First-Out (FIFO) principle. They are employed for process management, task scheduling, and breadth-first search algorithms. Queues enable efficient insertion at one end and removal at the other.

Bags are a type of data structure that stores unordered elements. They allow duplicates and enable frequency tracking. Bags are useful in scenarios where maintaining a distinct collection of items is not necessary, but counting occurrences or tracking frequency is essential. Sets, on the other hand, maintain unique elements. They provide efficient membership testing and support common set operations like union, intersection, and difference. Sets are utilized in various applications that require distinct elements and set manipulation.

Priority queues are data structures that store elements with associated priorities. They allow efficient retrieval of the highest priority element. Priority queues are commonly used in tasks like task scheduling, event-driven simulations, and graph algorithms. Finally, dictionaries (maps) store key-value pairs and provide efficient lookup and retrieval based on keys. They are extensively used for data indexing, symbol tables, and implementing algorithms that require fast access to values based on their associated keys.

Sec. 1.3 - Applications of ADTs

Abstract Data Types (ADTs) find applications across various domains, offering versatile solutions to address computational challenges. One common application of ADTs is in data storage and retrieval. ADTs like lists, arrays, and dictionaries (maps) provide flexible structures that enable efficient organization and access to data with different requirements for ordering, uniqueness, or key-value associations. These ADTs are used in databases, file systems, and data-driven applications to store and retrieve information in a structured and optimized manner.

ADTs play a crucial role in algorithm design. They are fundamental in solving computational problems efficiently. Stacks and queues, for example, are essential for managing program flow and data manipulation. They are used in areas such as compiler design, expression evaluation, and depth-first or breadth-first

traversals. Priority queues are particularly useful in optimization algorithms and event-driven simulations, where elements with associated priorities need to be processed in a specific order.

Memory management in programming languages relies on ADTs for efficient memory allocation and deallocation. Dynamic arrays, for instance, are used to dynamically allocate and resize memory blocks as needed. Stacks are instrumental in tracking function calls and managing runtime memory, ensuring efficient resource utilization. ADTs help manage memory effectively, preventing issues like memory leaks or excessive memory fragmentation.

ADTs have applications in various fields, including simulation modeling, task scheduling, and graph manipulation. Simulation models often rely on ADTs for modeling and analyzing complex systems. Queues are used for process scheduling, while bags and sets assist in statistical analysis, data sampling, and randomness generation. In task scheduling, ADTs like queues help manage process execution and prioritize tasks based on their priority levels or time constraints. In graph manipulation and network analysis, ADTs such as dictionaries (maps) provide efficient storage and retrieval of graph elements and properties, while priority queues can aid in graph algorithms like Dijkstra's algorithm for finding the shortest path.

Sections

Week 2

Object Orientation in C++ & ADT

3.0.1 Activities

The following are the activities that are planned for Week 2 of this course.

- Read the zyBook chapter(s) assigned and complete the reading quiz(s) by next Tuesday (usually Monday but it's a holiday).
- Read the C++ refresher or access other resources to improve your skills.
- Watch the videos on C++ Classes and Abstract Data Types.
- Watch the videos on Object-Oriented Thinking and Debugging your Assignments.
- Implement the examples In week videos for yourself on Jupytherhub machine.
- Access the GitHub Classroom to get your Assignment-1 repository (assignment due next Tuesday).

3.0.2 Lectures

Here are the lectures that can be found for this week:

- C++ Classes Basics
  - Source Files
- Abstract Data Type (ADT)
- Notes for Assignment 1 - Vector10
- Objected Oriented Thinking

- Object Lifestyle
- My Code is Not Working

3.0.3 Programming Assignment

The programming assignment for Week 2 - Vector10

3.0.4 Notes

The chapters of this week is Chapter 2 - Objects and Classes and Chapter 3 - Introduction to Algorithms.

Sec. 2.1 - Objects: Introduction

Objects

Objects are fundamental concepts in object-oriented programming (OOP) that represent real-world entities or abstract concepts. They encapsulate both data (attributes) and behavior (methods), allowing for modular and reusable code, enhanced code oganization, and modeling of complex systems. Objects promote the principles of encapsulation, inheritence, and polymorphism, facilitating efficient and modular software development.

Objects Example

Here is a simple example of objects in C++:

```
class Person {
private:
    std::string name;
    int age;

public:
    Person(const std::string& name, int age) : name(name), age(age) {}

    void displayInfo() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Person person("John", 25);
    person.displayInfo();
    return 0;
}
```

In this example, the Car class represents a car with a make, model, and year. An object of type Car named car is created in the main function, and its displayInfo method is called to print the car's make, model, and year.

Abstraction

Abstraction is a core concept in computer programming, helping to simplify complex systems by focusing on important aspects and hiding unnecessary details. It involves representing real-world objects or systems in a generalized way using classes and objects. Through abstraction, we create abstract classes that define a common interface and behavior for related objects while hiding implementation specifics. This allows us to manage system complexity, improve code organization, and promote reusability. Abstraction is crucial for creating modular, scalable, and maintainable software systems, allowing us to work at higher levels of abstraction without getting caught up in implementation intricacies.

Abstraction Example

An example of abstraction can be seen below:

```
#include <iostream>

// Abstract class
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function

    void printName() {
        std::cout << "Shape" << std::endl;
    }
};

// Concrete class
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

// Concrete class
class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

int main() {
    // Creating objects of concrete classes
    Circle circle;
    Rectangle rectangle;

    // Using the abstract class pointer to achieve abstraction
    Shape* shapePtr = nullptr;

    // Polymorphic behavior
    shapePtr = &circle;
    shapePtr->draw();  // Calls draw() of Circle class
```

```
        shapePtr = &rectangle;
        shapePtr->draw();  // Calls draw() of Rectangle class

        shapePtr->printName();  // Calls printName() of Shape class

        return 0;
    }
```

This code demonstrates the concept of abstraction and polymorphism using an example of shapes. It defines an abstract class called "Shape" with a pure virtual function "draw()" and a non-virtual function "printName()". Two concrete classes, "Circle" and "Rectangle", inherit from the abstract class and provide their own implementations of the "draw()" function.

In the main function, objects of the concrete classes are created. An abstract class pointer, "shapePtr", is used to achieve abstraction. The pointer is assigned the address of the "Circle" object, and the "draw()" function is called, resulting in the message "Drawing a circle." Similarly, the pointer is assigned the address of the "Rectangle" object, and the "draw()" function is called, resulting in the message "Drawing a rectangle." This demonstrates polymorphic behavior, where the appropriate "draw()" function is called based on the object type.

Additionally, the "printName()" function of the abstract class is called using the abstract class pointer. This function is not overridden in the concrete classes, so the implementation in the abstract class is invoked, printing the message "Shape".

Overall, this code illustrates the use of abstract classes, pure virtual functions, inheritance, and polymorphism to achieve abstraction and enable the flexible handling of different objects through a common interface. It showcases the power of using abstract classes and polymorphism to create modular and extensible code for working with related objects in a data structures context.

Sec. 2.2 - Using a Class

Public Member Functions

Public member functions in object-oriented programming allow objects to interact with each other and provide functionality to the outside world. They define the behavior and operations that objects of a class can perform, encapsulating the logic and operations related to the class. Public member functions serve as an interface through which users can interact with objects, accessing and utilizing the functionality provided without exposing the internal implementation details. They promote code reusability, encapsulation, and maintainability, ensuring controlled access to object behavior and enabling modular design in object-oriented programming.

Public Member Functions Example

Below is an example of Public Member Functions in C++:

```
#include <iostream>
```

```
class Rectangle {
private:
    int width;
    int height;

public:
    void setDimensions(int w, int h) {
        width = w;
        height = h;
    }

    int calculateArea() {
        return width * height;
    }

    void printInfo() {
        std::cout << "Width: " << width << ", Height: " << height << std::endl;
    }
};

int main() {
    Rectangle rect;

    rect.setDimensions(5, 3);
    int area = rect.calculateArea();
    std::cout << "Area: " << area << std::endl;

    rect.printInfo();

    return 0;
}
```

This code example demonstrates the concept of public member functions in C++. It defines a Rectangle class with private member variables for width and height. The class provides three public member functions: setDimensions(), calculateArea(), and printInfo().

The setDimensions() function allows users to set the width and height of the rectangle by passing the values as parameters. The calculateArea() function performs the calculation of the rectangle's area by multiplying the width and height and returns the result. Finally, the printInfo() function prints the width and height of the rectangle to the console.

In the main() function, an object of the Rectangle class is created, and the public member functions are utilized to set the dimensions of the rectangle, calculate its area, and print the information. This example showcases how public member functions provide an interface for interacting with objects, allowing users to manipulate data, perform computations, and retrieve information in a controlled manner, promoting encapsulation and modular design in C++ programming.

Sec. 2.3 - Defining a Class

Private Data Members

In object-oriented programming (OOP), private data members are a fundamental concept that allows for encapsulation and data hiding. Private data members are variables declared within a class that can only be accessed or modified by member functions within the same class. By designating data members as private, they are shielded from direct access by code outside the class, ensuring that the internal state and implementation details of an object are protected. This encapsulation promotes data integrity, enhances code maintainability, and prevents external code from inadvertently modifying or corrupting the object's data. Private data members facilitate information hiding and abstraction, allowing objects to maintain their integrity while providing controlled access to their functionality through public member functions.

Private Data Members Example

Here is an example of private data members in C++:

```
#include <iostream>

class BankAccount {
private:
    std::string accountNumber;
    double balance;

public:
    void deposit(double amount) {
        balance += amount;
    }

    void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            std::cout << "Insufficient balance." << std::endl;
        }
    }

    void displayBalance() {
        std::cout << "Account balance: " << balance << std::endl;
    }
};

int main() {
    BankAccount myAccount;

    myAccount.deposit(1000.0);
    myAccount.displayBalance();

    myAccount.withdraw(500.0);
    myAccount.displayBalance();

    myAccount.withdraw(800.0);
```

```
    myAccount.displayBalance();

    return 0;
}
```

In the main() function, an object of the BankAccount class is created. Public member functions are used to deposit an amount, display the balance, withdraw amounts, and display the updated balance.

By making the accountNumber and balance private, they cannot be directly accessed or modified from outside the class. This ensures the encapsulation and data hiding of sensitive information. Users can interact with the bank account object through the public member functions, maintaining data integrity and preventing unauthorized access to or modification of the private data members.

This example illustrates how private data members in C++ provide encapsulation and data hiding. By hiding the internal implementation details, the class enforces controlled access to the data and protects it from unauthorized manipulation. Private data members facilitate proper data management and security within an object, ensuring that only the intended interface, defined by public member functions, is used to interact with and modify the object's state.

Sec. 2.4 - Inline Member Functions

Inline Member Functions

An inline member function in C++ is a function that is defined within a class declaration and is marked with the inline keyword. When a member function is declared as inline, it suggests to the compiler that the function should be expanded at the point of its call instead of being invoked through a function call. This expansion replaces the function call with the actual code of the function, eliminating the overhead of the function call itself. Inline member functions are typically used for small and frequently used functions to improve performance by reducing the function call overhead. They provide a mechanism for code optimization and are especially useful when the function body is simple, making it more efficient to replace the function call with the actual code.

Inline Member Function Example

Below is an example of inline member functions:

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    void setDimensions(int w, int h) {
        width = w;
        height = h;
    }
```

```
    // Inline member function
    inline int calculateArea() {
        return width * height;
    }
};

int main() {
    Rectangle rect;

    rect.setDimensions(5, 3);
    int area = rect.calculateArea();

    std::cout << "Area: " << area << std::endl;

    return 0;
}
```

In this example, we have a Rectangle class with private data members width and height. The class provides two member functions: setDimensions() and calculateArea(). The setDimensions() function sets the width and height of the rectangle, while the calculateArea() function calculates the area of the rectangle by multiplying its width and height.

The calculateArea() function is declared as an inline member function by using the inline keyword before the function declaration. This suggests to the compiler that the function should be expanded at the point of its call. In this case, when calculateArea() is called, the compiler replaces the function call with the actual code of the function, eliminating the overhead of the function call.

In the main() function, an object of the Rectangle class is created. The setDimensions() function is called to set the width and height of the rectangle. Then, the calculateArea() function is invoked to calculate the area of the rectangle, and the result is printed to the console.

This example demonstrates how inline member functions can be used to optimize code performance by reducing the overhead of function calls. By marking the calculateArea() function as inline, the compiler expands the function call at the point of invocation, avoiding the function call overhead and providing direct access to the function's code. Inline member functions are particularly useful for small and frequently used functions, where the expansion at the call site can lead to performance improvements.

Sec. 2.5 - Mutators, Accessors, & Private Helpers

Mutators & Accessors

Mutators and accessors are two types of member functions commonly used in object-oriented programming to manipulate and retrieve the values of private data members of a class. Mutators, also known as setter functions or modifiers, are used to modify the values of private data members by accepting parameters and updating the internal state of the object. They provide a controlled way to change the values of the object's attributes while enforcing any necessary validation or business rules. Accessors, also known as getter functions or inspectors, are used to retrieve the values of private data members without allowing direct access to them. They return the values of private data members, allowing users

to access the object's attributes in a read-only manner. Mutators and accessors play a crucial role in encapsulation, providing an interface to manipulate and retrieve the object's state while maintaining data integrity, encapsulation, and abstraction. They allow for controlled interaction with the object's data and facilitate modular design and code maintainability by separating the implementation details from the external interface of the class.

Mutators & Accessors Example

Below is an example of mutators & accessors in C++:

```
#include <iostream>

class Circle {
private:
    double radius;

public:
    // Mutator
    void setRadius(double r) {
        if (r >= 0) {
            radius = r;
        }
    }

    // Accessor
    double getRadius() const {
        return radius;
    }

    double calculateArea() const {
        return 3.14 * radius * radius;
    }
};

int main() {
    Circle myCircle;

    myCircle.setRadius(5.0);
    double radius = myCircle.getRadius();
    double area = myCircle.calculateArea();

    std::cout << "Radius: " << radius << std::endl;
    std::cout << "Area: " << area << std::endl;

    return 0;
}
```

In this example, we have a Circle class with a private data member radius. The class provides two member functions: setRadius() and getRadius().

The setRadius() function is a mutator that allows users to set the value of the radius data member. It accepts a parameter r and updates the radius only if the value is non-negative.

The getRadius() function is an accessor that returns the value of the radius data member. It allows users to retrieve the value of radius without directly accessing the private data member.

In the main() function, an object of the Circle class is created. The setRadius() mutator is called to set the radius of the circle to 5.0. The getRadius() accessor is then used to retrieve the value of the radius, and the calculateArea() function is invoked to calculate the area of the circle. Finally, the radius and area are printed to the console.

This example demonstrates how mutators and accessors provide a controlled interface for manipulating and retrieving the values of private data members. The mutator setRadius() allows users to set the radius of the circle, while the accessor getRadius() allows them to retrieve the radius. By encapsulating the private data member and providing these member functions, the class ensures data integrity and abstraction. Users can interact with the object through the mutators and accessors without direct access to the private data member, promoting encapsulation and modular design in C++ programming.

Sec. 2.6 - Initialization & Constructors

Data Member Initialization

Data member initialization in C++ allows you to assign initial values to the data members of a class when objects are created. It provides a convenient way to ensure that data members have valid initial values and avoids the need for separate initialization steps. Data member initialization can be done using two approaches: member initialization list and default member initializer. Member initialization list initializes data members directly in the constructor's initialization list, while default member initializer assigns values to data members directly in the class declaration. By initializing data members during object creation, you can ensure that the object starts in a consistent state and avoid potential bugs or undefined behavior caused by uninitialized data. Data member initialization enhances code readability, simplifies object construction, and promotes good programming practices in C++.

Data Member Initialization Example

Here is an example of data member initialization in C++.

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Constructor with member initialization list
    Rectangle(int w, int h) : width(w), height(h) {
        // Additional constructor code, if needed
    }
```

```
        // Default member initializer
        int area = width * height;

        void printArea() {
            std::cout << "Area: " << area << std::endl;
        }
    };

    int main() {
        Rectangle rect(5, 3);
        rect.printArea();

        return 0;
    }
```

In this example, we have a Rectangle class with private data members width and height. There are two ways to initialize these data members.

Firstly, in the constructor declaration, we use a member initialization list to initialize the width and height data members directly. The constructor takes two parameters w and h, and the member initialization list assigns these values to the corresponding data members.

Secondly, we can use default member initializer directly in the class declaration. In this case, we initialize the area data member using a default member initializer, which calculates the area as the product of width and height.

In the main() function, we create an object of the Rectangle class named rect with width 5 and height 3. The constructor initializes the width and height data members using the member initialization list. The printArea() function is called, which displays the calculated area of the rectangle.

This example demonstrates how data member initialization can be done using member initialization list in the constructor or default member initializer in the class declaration. It ensures that the data members have valid initial values when objects are created, simplifies object construction, and promotes code readability. Data member initialization is a useful feature in C++ that helps ensure the consistency and integrity of objects' initial states.

Constructors

Constructors in object-oriented programming (OOP) are special member functions that are responsible for initializing objects of a class. They are called automatically when an object is created and allow you to set the initial state of the object. Constructors have the same name as the class and can have parameters to receive values required for initialization. They can perform various tasks, such as allocating memory, initializing data members, setting default values, and executing other necessary initialization logic. Constructors play a crucial role in object creation and ensure that objects start in a valid and consistent state. They promote encapsulation, as they provide a controlled way to initialize objects and enforce any necessary validation or business rules during the creation process. Constructors contribute to code readability, reusability, and maintainability by encapsulating the object initialization logic within the class itself.

Constructors Example

Here is an example of constructors in C++:

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Default constructor
    Rectangle() {
        width = 0;
        height = 0;
    }

    // Parameterized constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    void printDimensions() {
        std::cout << "Width: " << width << ", Height: " << height << std::endl;
    }
};

int main() {
    // Creating objects using constructors
    Rectangle rect1; // Default constructor called
    Rectangle rect2(5, 3); // Parameterized constructor called

    // Printing dimensions
    rect1.printDimensions(); // Output: Width: 0, Height: 0
    rect2.printDimensions(); // Output: Width: 5, Height: 3

    return 0;
}
```

In this example, we have a Rectangle class with private data members width and height. The class provides two constructors: a default constructor and a parameterized constructor.

The default constructor initializes the width and height to 0. It is called automatically when an object is created without any arguments, as in the case of rect1.

The parameterized constructor takes two arguments w and h and initializes the width and height using the provided values. It is called when an object is created with specific values, as in the case

of rect2.

In the main() function, we create two objects of the Rectangle class, rect1 and rect2, using the constructors. We then call the printDimensions() function to display the dimensions of the rectangles.

This example demonstrates how constructors are used to initialize objects of a class. The default constructor allows objects to be created with default values, while the parameterized constructor allows objects to be created with custom values. Constructors enable proper initialization of objects, ensuring they start in a valid state. They provide flexibility and encapsulation in object creation, enhancing code readability and maintainability.

Sec. 2.7 - Classes and Vectors / Classes

Vectors

The 'std::vector' class in C++ is a dynamic array container that provides a flexible and convenient way to store and manipulate a sequence of elements. It allows for dynamic resizing, efficient element access, insertion, and deletion at both ends, and provides various member functions to perform common operations on the elements. Vectors are templated, which means they can store elements of any type, providing great flexibility. They offer automatic memory management, handling memory allocation and deallocation internally. Vectors are widely used in C++ programming due to their versatility, efficiency, and ease of use, making them a fundamental data structure for managing collections of elements.

Vectors Example

Here is an example of the 'vectors' class in C++:

```
#include <iostream>
#include <vector>

int main() {
    // Create a vector of integers
    std::vector<int> numbers;

    // Add elements to the vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Access elements using indexing
    std::cout << "First element: " << numbers[0] << std::endl;
    std::cout << "Second element: " << numbers[1] << std::endl;
    std::cout << "Third element: " << numbers[2] << std::endl;

    // Iterate over the vector using a loop
    std::cout << "All elements: ";
    for (int i = 0; i < numbers.size(); i++) {
        std::cout << numbers[i] << " ";
    }
    std::cout << std::endl;
```

```
        // Remove the last element
        numbers.pop_back();

        // Check the size of the vector
        std::cout << "Size of vector: " << numbers.size() << std::endl;

        return 0;
    }
```

In this example, we include the necessary header files for using std::vector. We create a vector named numbers that stores integers.

We use the push_back() function to add elements to the vector. In this case, we add the integers 10, 20, and 30 to the vector. We access elements of the vector using indexing, such as numbers[0] to access the first element. We iterate over the vector using a loop and print all the elements. We use the pop_back() function to remove the last element from the vector. Finally, we check the size of the vector using the size() function.

This example demonstrates the basic usage of std::vector in C++. It shows how to create a vector, add elements, access elements using indexing, iterate over the vector, remove elements, and check the size. The std::vector class provides a convenient and flexible way to work with dynamic arrays, making it a powerful data structure in C++ for managing collections of elements.

Sec. 2.8 - Separate Files for Classes

Two Files Per Class

Separate files for classes in C++ programs provide a modular approach to organizing code. Each class is defined in its own header file, containing the class declaration, and the member function implementations are placed in a corresponding source file. This practice enhances code organization, readability, and reusability. It simplifies navigation, allowing developers to quickly locate and modify code related to specific classes. Separating classes into individual files also promotes code reuse by facilitating their inclusion in other projects. Additionally, it aids in managing dependencies, prevents name conflicts, and simplifies maintenance and debugging. Overall, utilizing separate files for classes in C++ programs improves code structure and facilitates the development and management of complex projects.

Sec. 2.9 - Choosing Classes to Create

Decomposing Into Classes

When creating classes in Object-Oriented Programming (OOP), it is important to follow certain guidelines to ensure a well-designed and effective class structure. Start by identifying the attributes and behaviors that define the class's purpose and responsibilities. Encapsulate the data by declaring private data members and provide public access through member functions. Design intuitive and descriptive names for the class and its members. Establish clear and meaningful relationships between classes, using inheritance and composition when appropriate. Implement appropriate constructors, destructors, and assignment operators to manage the lifecycle of objects. Strive for cohesive and focused classes with single responsibilities. Apply principles like encapsulation, abstraction, inheritance, and polymorphism to achieve modularity, code reusability, and maintainability. Document the class with clear comments and

adhere to coding style conventions for consistency. Regularly review and refine the class design as needed to ensure a well-structured and efficient implementation.

Sec. 2.10 - Unit Testing (Classes)

Testbenches

In Object-Oriented Programming (OOP), a test bench refers to a dedicated component or code module designed to test and validate the functionality of other classes or modules in a system. It serves as an environment for conducting systematic and comprehensive testing of software components. A test bench provides a controlled setting to simulate different scenarios and input conditions, allowing developers to verify the correctness and robustness of their code. It typically includes test cases, input data, and expected output values, along with mechanisms to execute the tests and compare the actual results against the expected ones. By using test benches, developers can identify and rectify issues early in the development process, ensuring the quality and reliability of the software. Test benches play a crucial role in achieving effective testing and debugging practices, enabling thorough assessment and validation of object-oriented systems.

Regression Testing

In Object-Oriented Programming (OOP), regression testing refers to the process of retesting previously tested code to ensure that any modifications or enhancements to the system do not introduce new defects or regressions. It involves rerunning existing test cases on the modified code to verify that the changes made to the system have not adversely affected its existing functionality. Regression testing is crucial for maintaining the stability and reliability of software systems, especially in complex object-oriented projects where changes in one module can have unintended consequences on other interconnected modules. By performing regression testing, developers can identify and fix any regressions or unintended side effects caused by code modifications, ensuring that the system continues to function as expected and previous functionalities are not compromised. Regression testing is an integral part of the software development lifecycle, providing confidence in the system's integrity and minimizing the risk of introducing new defects during the development and maintenance phases.

Erroneous Unit Tests

Erroneous unit tests refer to test cases or test code that are flawed or incorrect, resulting in inaccurate or misleading test results. These tests may have various issues, such as incorrect assumptions, flawed logic, inadequate coverage, or improper assertions. Erroneous unit tests can lead to false positives or false negatives, where passing tests falsely indicate correct functionality or failing tests erroneously indicate defects. Such tests can be problematic as they can give a false sense of security or create confusion during the development process. It is important to identify and rectify erroneous unit tests promptly to ensure the reliability and effectiveness of the testing process. Conducting regular code reviews, employing static analysis tools, and encouraging collaboration and knowledge sharing within the development team can help in identifying and addressing erroneous unit tests, resulting in more accurate and reliable testing outcomes.

Unit Test Example

Here is an example of unit testing classes in C++:

```
// File: MyClass.h
#ifndef MYCLASS_H
```

```
#define MYCLASS_H

class MyClass {
private:
    int value;

public:
    MyClass(int val);

    int getValue() const;
    void setValue(int val);
};

#endif

// File: MyClass.cpp
#include "MyClass.h"

MyClass::MyClass(int val) : value(val) {}

int MyClass::getValue() const {
    return value;
}

void MyClass::setValue(int val) {
    value = val;
}

// File: MyClassTest.cpp
#include <gtest/gtest.h>
#include "MyClass.h"

TEST(MyClassTest, ConstructorSetsInitialValue) {
    MyClass obj(42);
    EXPECT_EQ(obj.getValue(), 42);
}

TEST(MyClassTest, SettingNewValueUpdatesValue) {
    MyClass obj(0);
    obj.setValue(100);
    EXPECT_EQ(obj.getValue(), 100);
}

int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

In this example, we have a class called MyClass with a private member variable value and public member functions getValue() and setValue(). We write unit tests for this class using the Google

Test framework.

In the MyClassTest.cpp file, we define two test cases using the TEST macro provided by Google Test. Each test case focuses on testing a specific aspect of the MyClass class. For example, one test case checks if the constructor sets the initial value correctly, and another test case verifies that setting a new value updates the value correctly.

The main function initializes the Google Test framework using testing::InitGoogleTest and runs all the defined tests using RUN_ALL_TESTS(). To compile and run the tests, you would need to include the Google Test framework and compile the test files along with it.

This example demonstrates how you can use unit testing to verify the behavior and correctness of your class implementation. Each test case focuses on a specific aspect of the class, ensuring that it behaves as expected in different scenarios. By running these tests, you can identify and address any issues or regressions in your class implementation, leading to more reliable and robust code.

Sec. 2.11 - Constructor Overloading

Basics

Constructor overloading in Object Oriented Programming (OOP) refers to the ability to define multiple constructors for a class, each with different set of parameters. This allows objects to be created with different initial states or configurations, providing flexibility and customization during object instantiation. By overloading constructors, developers can conveniently initialize objects with different combinations of values or provide default values for certain parameters. Constructor overloading enables the creation of objects that meet specific requirements or use cases, making the class more versatile and adaptable to different scenarios. It promotes code reuse and enhances the usability of the class by accommodating various ways of object initialization.

Constructor Overloading Example

Below is an example of constructor overloading in C++:

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    // Default constructor
    MyClass() {
        value = 0;
    }

    // Constructor with one parameter
    MyClass(int val) {
        value = val;
    }
```

```
        // Constructor with two parameters
        MyClass(int val1, int val2) {
            value = val1 + val2;
        }

        int getValue() const {
            return value;
        }
    };

    int main() {
        MyClass obj1;               // Calls the default constructor
        MyClass obj2(42);           // Calls the constructor with one parameter
        MyClass obj3(10, 20);       // Calls the constructor with two parameters

        std::cout << obj1.getValue() << std::endl;    // Output: 0
        std::cout << obj2.getValue() << std::endl;    // Output: 42
        std::cout << obj3.getValue() << std::endl;    // Output: 30

        return 0;
    }
```

In the above example, the MyClass class demonstrates constructor overloading. It has three constructors: a default constructor, a constructor with one parameter, and a constructor with two parameters. Each constructor initializes the value member variable based on the provided arguments or default values.

In the main function, we create three objects of the MyClass class using different constructor calls. The first object obj1 is created using the default constructor, which sets the value to 0. The second object obj2 is created by invoking the constructor with one parameter, setting the value to 42. The third object obj3 is created using the constructor with two parameters, where the value is the sum of the two provided values (10 and 20).

By overloading the constructors, we can instantiate objects with different initial states or configurations depending on the parameters provided. This enhances the flexibility and usability of the class, allowing developers to create objects with specific values or default values conveniently. Constructor overloading promotes code reuse and simplifies the process of object creation, making the class more versatile and adaptable to different use cases.

Sec. 2.12 - Constructor Initializer List

Constructor initializer lists in Object-Oriented Programming (OOP) provide a way to initialize class member variables directly in the constructor declaration, rather than assigning values to them within the body of the constructor. By using the initializer list syntax, constructors can efficiently initialize member variables, especially for cases involving const variables or reference variables that need to be initialized upon object creation. Constructor initializer lists offer several benefits, including improved performance, the ability to initialize const and reference variables, and the initialization of base class subobjects. They enhance code readability and maintainability by clearly expressing the initialization process and ensuring that member variables are properly initialized before the constructor body executes. Overall, constructor

initializer lists are a powerful feature in C++ that enable efficient and proper initialization of class member variables during object construction.

Constructor Initializer List Example

Below is an example of constructor initializer list in C++:

```
#include <iostream>

class MyClass {
private:
    int value;
    const int constantValue;
    int& refValue;

public:
    MyClass(int val, int& ref) : value(val), constantValue(42), refValue(ref) {
        // Constructor body
    }

    int getValue() const {
        return value;
    }

    int getConstantValue() const {
        return constantValue;
    }

    int& getRefValue() const {
        return refValue;
    }
};

int main() {
    int ref = 100;
    MyClass obj(42, ref);

    std::cout << obj.getValue() << std::endl;          // Output: 42
    std::cout << obj.getConstantValue() << std::endl;  // Output: 42
    std::cout << obj.getRefValue() << std::endl;       // Output: 100

    return 0;
}
```

In the above example, the MyClass class demonstrates the use of constructor initializer lists. It has three member variables: value, constantValue, and refValue, representing an integer, a constant integer, and a reference to an integer, respectively.

In the MyClassTest.cpp file, we define two test cases using the TEST macro provided by Google

Test. Each test case focuses on testing a specific aspect of the MyClass class. For example, one test case checks if the constructor sets the initial value correctly, and another test case verifies that setting a new value updates the value correctly.

By using the constructor initializer list, we can efficiently and directly initialize these member variables, including the initialization of const and reference variables, which cannot be assigned values inside the constructor body.

The main function creates an object of the MyClass class, passing in the values 42 and ref as arguments. We can then access the member variables using the appropriate getter functions to verify their values.

Constructor initializer lists enhance code readability by explicitly and efficiently initializing member variables during object construction. They ensure proper initialization of const and reference variables, making the code more robust and maintainable. By using initializer lists, we can initialize member variables directly, avoiding unnecessary assignment statements within the constructor body.

Sec. 2.13 - The 'this' Implicit Parameter

Implicit Parameter

In C++, the 'this' implicit parameter is a pointer that is automatically passed to member functions of a class. It refers to the object on which the member function is being called. The 'this' pointer allows access to the member variables and member functions of the object within its own scope, distinguishing them from local variables or function parameters with the same name. It is particularly useful in scenarios where there is a need to differentiate between the object's member variables and function parameters that have the same names. The 'this' pointer enables efficient and unambiguous access to the object's data and behavior, promoting encapsulation and facilitating object-oriented programming principles.

Using 'this' In Class Member Functions and Constructors

In C++, the 'this' pointer is used in class member functions and constructors to refer to the object on which the function is being invoked. Within member functions, 'this' allows direct access to the member variables and member functions of the current object, differentiating them from local variables or function parameters. It is particularly useful when there is a need to disambiguate between class members and local variables with the same name. 'this' can also be used in constructors to initialize member variables, especially in cases where the parameter names clash with the member variable names. By using 'this' in member functions and constructors, developers can ensure accurate and unambiguous access to the object's data and behavior, promoting clarity, readability, and maintainability of the code.

'this' Implicit Parameter Example

Here is an example of the use of 'this' implicit parameter in C++:

```
#include <iostream>

class MyClass {
private:
    int value;
```

```
    public:
        MyClass(int value) {
            this->value = value;
        }

        void printValue() {
            std::cout << "Value: " << this->value << std::endl;
        }
    };

    int main() {
        MyClass obj(42);
        obj.printValue();  // Output: Value: 42

        return 0;
    }
```

In the above example, we have a class called MyClass with a private member variable value and a constructor that takes an integer parameter. Inside the constructor, we use the 'this' pointer to differentiate between the parameter value and the member variable value. By using this->value, we explicitly refer to the member variable and assign the value of the parameter to it.

The printValue() member function of MyClass also uses the 'this' pointer. Within the function, we use this->value to access the member variable and print its value. In the main() function, we create an object of MyClass called obj and pass the value 42 to the constructor. We then call the printValue() member function on the obj object, which outputs the value of the value member variable.

By using the 'this' pointer, we can differentiate between local variables and member variables within the class scope, ensuring the correct variable is accessed or modified. It promotes clarity and avoids naming conflicts between function parameters and member variables.

Sec. 2.14 - Operator Overloading

Overview

Operator overloading in Object-Oriented Programming (OOP) allows the customization of the behavior of predefined operators for user-defined classes. It enables objects of a class to exhibit intuitive and meaningful behavior when used with operators such as +, -, *, /, ==, and so on. By overloading operators, developers can define how objects of a class interact with operators, making code more expressive and natural. Operator overloading enables the use of familiar syntax and semantics for user-defined types, enhancing code readability and maintainability. It allows objects to participate in operations that are consistent with their intended purpose, leading to more concise and intuitive code.

Overloading Same Operator

Overloading the same operator more than once in a single class in C++ allows different behaviors to be defined for the same operator depending on the types of the operands. This feature is known as operator overloading with different argument types. By providing multiple implementations of an operator,

each with distinct parameter types, the class can handle different scenarios and provide appropriate behavior for each case. This enables flexibility in how the class interacts with the operator, accommodating various operand combinations and ensuring consistent and meaningful operations. Overloading the same operator multiple times in a class allows for versatile and specialized behavior, enhancing the usability and adaptability of the class within different contexts.

Operator Overloading Example

Below is an example of operator overloading in C++:

```
#include <iostream>

class Vector2D {
private:
    double x, y;

public:
    Vector2D(double x = 0.0, double y = 0.0) : x(x), y(y) {}

    Vector2D operator+(const Vector2D& other) const {
        return Vector2D(x + other.x, y + other.y);
    }

    Vector2D operator-(const Vector2D& other) const {
        return Vector2D(x - other.x, y - other.y);
    }

    Vector2D operator*(double scalar) const {
        return Vector2D(x * scalar, y * scalar);
    }
};

int main() {
    Vector2D v1(2.0, 3.0);
    Vector2D v2(1.0, 2.0);

    Vector2D sum = v1 + v2;           // Operator+ overload
    Vector2D difference = v1 - v2;     // Operator- overload
    Vector2D scaled = v1 * 2.5;        // Operator* overload

    std::cout << "Sum: (" << sum.x << ", " << sum.y << ")" << std::endl;
    std::cout << "Difference: (" << difference.x << ", " << difference.y << ")"
    << std::endl;
    std::cout << "Scaled: (" << scaled.x << ", " << scaled.y << ")"
    << std::endl;

    return 0;
}
```

In the above example, we have a class called Vector2D representing a 2D vector. The class overloads the +, -, and * operators to perform vector addition, subtraction, and scalar multiplication, respectively.

By providing multiple implementations of the same operator, each with different parameter types (Vector2D and double in this case), the class can handle different scenarios. The operator+ overload performs element-wise addition of the coordinates, the operator- overload performs element-wise subtraction, and the operator* overload performs scalar multiplication.

In the main() function, we create two Vector2D objects, v1 and v2. We then use the overloaded operators to perform vector addition, subtraction, and scalar multiplication. The results are stored in sum, difference, and scaled variables, respectively.

The program outputs the results, demonstrating how the overloaded operators provide intuitive and meaningful behavior for the Vector2D class. Overloading the same operator multiple times in the class allows for flexible and specialized operations, enhancing the usability and expressiveness of the class.

Sec. 2.15 - Overloading Comparison Operators

Overloading comparison operators in Object-Oriented Programming (OOP) allows custom behavior to be defined for comparing objects of user-defined classes. By overloading operators such as ==, !=, <, >, <=, and >=, developers can specify how objects should be compared based on their internal data or specific criteria. This enables objects of a class to be compared in a way that is meaningful and appropriate for the class's concept and purpose. Overloading comparison operators allows for more natural and intuitive code, as objects can be compared using familiar syntax and semantics. It enhances the readability and clarity of code by providing consistent and logical comparisons for user-defined types, making it easier to reason about the behavior of objects in comparison operations.

Comparison Operator Overloading Example

Below is an example of comparison operator overloading in C++:

```
#include <iostream>

class Fraction {
private:
    int numerator;
    int denominator;

public:
    Fraction(int numerator = 0, int denominator = 1)
    : numerator(numerator), denominator(denominator) {}

    bool operator==(const Fraction& other) const {
        return (numerator == other.numerator)
        && (denominator == other.denominator);
    }

    bool operator!=(const Fraction& other) const {
```

```
        return !(*this == other);
    }

    bool operator<(const Fraction& other) const {
        return (numerator * other.denominator)
        < (other.numerator * denominator);
    }

    bool operator>(const Fraction& other) const {
        return (numerator * other.denominator)
        > (other.numerator * denominator);
    }
};

int main() {
    Fraction f1(3, 4);
    Fraction f2(2, 3);
    Fraction f3(3, 4);

    if (f1 == f2) {
        std::cout << "f1 and f2 are equal." << std::endl;
    } else {
        std::cout << "f1 and f2 are not equal." << std::endl;
    }

    if (f1 != f3) {
        std::cout << "f1 and f3 are not equal." << std::endl;
    } else {
        std::cout << "f1 and f3 are equal." << std::endl;
    }

    if (f2 < f1) {
        std::cout << "f2 is less than f1." << std::endl;
    } else {
        std::cout << "f2 is not less than f1." << std::endl;
    }

    if (f1 > f2) {
        std::cout << "f1 is greater than f2." << std::endl;
    } else {
        std::cout << "f1 is not greater than f2." << std::endl;
    }

    return 0;
}
```

In the above example, we have a Fraction class representing a fraction with a numerator and denominator. We overload the comparison operators ==, !=, <, and > to compare fractions.

The operator== compares two fractions for equality, checking if both the numerator and

denominator are the same. The operator!= is implemented in terms of operator==, negating the result. The operator< compares fractions based on their relative values, using cross multiplication to compare the numerators and denominators. Similarly, the operator> is implemented based on operator<, but with the operands swapped.

In the main() function, we create three Fraction objects, f1, f2, and f3, and perform comparison operations using the overloaded operators. We check for equality, inequality, less than, and greater than relationships between fractions and print the corresponding messages.

The program outputs the results of the comparisons, demonstrating the custom behavior defined by overloading the comparison operators. By overloading these operators, we can compare fractions using intuitive syntax and obtain meaningful results based on their numerical values.

Sec. 2.16 - Vector ADT

The Vector Abstract Data Type (ADT) is a versatile and efficient dynamic array-like structure that allows for the flexible storage and manipulation of elements. It offers constant-time access by index, efficient appending and removal of elements, and automatic resizing when needed. Vectors are widely used in programming for their ability to adapt to changing collection sizes, making them suitable for a variety of applications. They provide a contiguous block of memory, allowing for efficient traversal and sequential access. With their ability to store elements of any type, vectors serve as a fundamental data structure in algorithms, data structures, and applications that require dynamic and efficient element storage. Understanding the capabilities and operations of the Vector ADT is crucial for effectively managing and manipulating collections of elements in programming tasks.

Vector ADT Example

Here is an example of a vector ADT in C++:

```
#include <iostream>
#include <vector>

int main() {
    // Creating a vector to store integers
    std::vector<int> numbers;

    // Adding elements to the vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Accessing elements by index
    std::cout << "First element: " << numbers[0] << std::endl;
    std::cout << "Second element: " << numbers[1] << std::endl;
    std::cout << "Third element: " << numbers[2] << std::endl;

    // Iterating over the vector
    std::cout << "Elements in the vector: ";
    for (int i = 0; i < numbers.size(); i++) {
        std::cout << numbers[i] << " ";
```

```
    }
    std::cout << std::endl;

    // Removing an element from the vector
    numbers.pop_back();

    // Querying the size and capacity of the vector
    std::cout << "Size of the vector: " << numbers.size() << std::endl;
    std::cout << "Capacity of the vector: " << numbers.capacity() << std::endl;

    return 0;
}
```

In this example, we include the <vector> header to use the Vector ADT provided by the C++ Standard Library. We create a vector called numbers to store integers. We use the push_back() function to add elements to the vector, and the [] operator to access elements by index. We iterate over the vector using a loop and print the elements. Then, we remove an element using the pop_back() function. Finally, we query the size of the vector using the size() function and the capacity using the capacity() function.

When you run this program, it will output the elements of the vector, the size, and the capacity. This example demonstrates the basic usage of the Vector ADT in C++ for dynamic storage and manipulation of elements.

Sec. 2.17 - Namespaces

Namespaces in C++ provide a way to group related code elements and prevent naming conflicts. They act as a container for identifiers such as variables, functions, and classes, allowing them to be organized and accessed in a structured manner. By enclosing code within a namespace, we can avoid naming collisions between entities with the same name but defined in different contexts. Namespaces enhance code modularity, readability, and maintainability by providing a hierarchical structure to the codebase. They enable developers to create separate logical units and manage the scope of identifiers more effectively. With namespaces, it becomes easier to differentiate and reference code elements, making the codebase more manageable and reducing the risk of naming conflicts when integrating different libraries or modules.

Namespaces Example

Here is an example of namespaces in C++:

```
#include <iostream>

// First namespace
namespace First {
    void greet() {
        std::cout << "Hello from First namespace!" << std::endl;
    }
}
```

```
// Second namespace
namespace Second {
    void greet() {
        std::cout << "Hello from Second namespace!" << std::endl;
    }
}

int main() {
    First::greet();   // Calling greet() from the First namespace
    Second::greet();  // Calling greet() from the Second namespace

    return 0;
}
```

In this example, we define two namespaces: First and Second. Each namespace has its own greet() function that outputs a greeting message. In the main() function, we explicitly specify the namespace when calling the greet() function to differentiate between the two implementations.

This example demonstrates how namespaces in C++ allow us to organize code elements into separate logical units. By enclosing code within namespaces, we can prevent naming conflicts and explicitly specify which version of a function or variable to use. Namespaces help improve code readability and maintainability, especially in larger projects where different libraries or modules may have overlapping identifiers.

Sec. 2.18 - Static Data Members & Functions

Static data members and functions in C++ are associated with the class itself rather than specific instances of the class. A static data member is shared among all objects of the class and has a single instance regardless of the number of objects created. Similarly, a static member function is not bound to any specific object and can be called directly using the class name. Static members are useful for storing and accessing shared data or performing operations that are independent of individual objects. They can be accessed without creating an instance of the class and are commonly used for maintaining counts, global variables, utility functions, or class-wide properties. Static members provide a way to encapsulate data or functionality that is not tied to a specific object but belongs to the class as a whole.

Static Data Members & Functions Example

Below is an example of static data members & functions in C++:

```
#include <iostream>

class MyClass {
public:
    static int count;  // Static data member

    static void incrementCount() {  // Static member function
        count++;
```

```
    }

    void displayCount() {
        std::cout << "Count: " << count << std::endl;
    }
};

int MyClass::count = 0;  // Initializing static data member

int main() {
    MyClass::incrementCount();  // Calling static member function
    MyClass obj1;
    obj1.displayCount();  // Output: Count: 1

    MyClass::incrementCount();
    MyClass obj2;
    obj2.displayCount();  // Output: Count: 2

    MyClass::count = 10;  // Modifying static data member directly

    MyClass obj3;
    obj3.displayCount();  // Output: Count: 10

    return 0;
}
```

In this example, we have a class called MyClass with a static data member count and a static member function incrementCount(). The count variable is shared among all objects of the class and is initialized to 0. The incrementCount() function increments the count by one. In the main() function, we call the static member function incrementCount() using the class name MyClass::incrementCount(). We also create multiple objects of MyClass and call the member function displayCount() to display the current value of count. We can directly access and modify the static data member count using the class name as shown. The output demonstrates how the static data member is shared among all objects and how the static member function can be used to manipulate it.

Sec. 3.1 - Introduction to Algorithms

Algorithms

In object-oriented programming (OOP), an algorithm refers to a set of step-by-step instructions or procedures designed to solve a specific problem or perform a particular task. It is a logical sequence of operations that can be implemented in code to achieve a desired outcome. In OOP, algorithms are often encapsulated within methods or functions of classes, enabling reusability and modularity. Algorithms in OOP can involve various operations such as data manipulation, conditional statements, loops, and function calls. They play a crucial role in implementing the logic and functionality of programs by providing a systematic approach to solving problems and achieving specific objectives. Well-designed algorithms are efficient, correct, and maintainable, contributing to the overall effectiveness and quality of the software.

Algorithm Efficiency

Algorithm efficiency in object-oriented programming (OOP) refers to the measure of how well an algorithm utilizes computational resources such as time and memory. It involves analyzing the performance characteristics of an algorithm and understanding its scalability as the input size increases. Efficiency is crucial in OOP as it directly impacts the program's overall performance and resource utilization. By designing and implementing efficient algorithms, developers can optimize the execution time and memory usage of their programs, leading to faster and more responsive software. Techniques like algorithmic complexity analysis, Big O notation, and data structure selection are employed to evaluate and improve algorithm efficiency. Striving for efficient algorithms is essential for developing high-performance applications that can handle large-scale data and complex computations effectively.

Big $\mathcal{O}$ Notation

Below are the different Big $\mathcal{O}$ Notations for algorithms with a simple explanation of each:

Big $\mathcal{O}$ Notation	Explanation
$\mathcal{O}(1)$	Constant time complexity - The algorithm's execution time is constant regardless of the input size.
$\mathcal{O}(\log(n))$	Logarithmic time complexity - The algorithm's execution time increases logarithmically with the input size.
$\mathcal{O}(n)$	Linear time complexity - The algorithm's execution time increases linearly with the input size.
$\mathcal{O}(n \log(n))$	Linearithmic time complexity - The algorithm's execution time grows in proportion to the product of the input size and its logarithm.
$\mathcal{O}(n^2)$	Quadratic time complexity - The algorithm's execution time increases quadratically with the input size.
$\mathcal{O}(2^n)$	Exponential time complexity - The algorithm's execution time grows exponentially with the input size.
$\mathcal{O}(n!)$	Factorial time complexity - The algorithm's execution time increases factorially with the input size.

These notations provide a way to express the scalability and efficiency of algorithms, allowing developers to compare and analyze different algorithms based on their time complexity and make informed decisions when designing and optimizing their programs.

To further demonstrate what an algorithm is, we take a look at a couple of examples.

Algorithms Example

Below are some examples of algorithms in C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {4, 2, 7, 5, 1, 3, 6};

    // O(1) - Accessing an element in a vector using index
```

```
int element = numbers[2];  // Accessing the third element

// O(log n) - Binary search algorithm
std::sort(numbers.begin(), numbers.end());
bool found = std::binary_search(numbers.begin(), numbers.end(), 5);

// O(n) - Linear search algorithm
bool exists = std::find(numbers.begin(), numbers.end(), 8) != numbers.end();

// O(n log n) - Sorting algorithm (e.g., Quick Sort)
std::sort(numbers.begin(), numbers.end());

// O(n^2) - Bubble sort algorithm
for (int i = 0; i < numbers.size() - 1; i++) {
    for (int j = 0; j < numbers.size() - i - 1; j++) {
        if (numbers[j] > numbers[j + 1]) {
            std::swap(numbers[j], numbers[j + 1]);
        }
    }
}

// O(2^n) - Recursive Fibonacci sequence calculation
int fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// O(n!) - Permutation generation using recursion
void generatePermutations(std::vector<int>& arr, int start, int end) {
    if (start == end) {
        for (int num : arr) {
            std::cout << num << " ";
        }
        std::cout << std::endl;
    } else {
        for (int i = start; i <= end; i++) {
            std::swap(arr[start], arr[i]);
            generatePermutations(arr, start + 1, end);
            std::swap(arr[start], arr[i]);
        }
    }
}

// Example usage of the functions
std::cout << "Element: " << element << std::endl;
std::cout << "Binary search found: " << found << std::endl;
std::cout << "Linear search exists: " << exists << std::endl;

std::cout << "Sorted numbers: ";
for (int num : numbers) {
    std::cout << num << " ";
```

```
        }
        std::cout << std::endl;

        int fibResult = fibonacci(5);
        std::cout << "Fibonacci(5): " << fibResult << std::endl;

        std::vector<int> permutationArr = {1, 2, 3};
        generatePermutations(permutationArr, 0, permutationArr.size() - 1);

        return 0;
    }
```

This code demonstrates the use of different algorithms corresponding to various Big $\mathcal{O}$ notations. It includes examples such as accessing an element in a vector with $\mathcal{O}(1)$, binary search with $\mathcal{O}(\log(n))$, linear search with $\mathcal{O}(n)$, sorting algorithms with $\mathcal{O}(n\log(n))$ and $\mathcal{O}(n^2)$, recursive Fibonacci sequence calculation with $\mathcal{O}(2^n)$, and permutation generation using recursion with $\mathcal{O}(n!)$. The output of the code showcases the results of each algorithm. This example provides a practical illustration of how different algorithms perform in terms of time complexity and highlights their corresponding efficiency characteristics.

Sec. 3.2 - Relation Between Data Structures and Algorithms

Algorithms for Data Structures

Algorithms for data structures refer to the set of procedures or methods designed to operate on specific data structures efficiently. These algorithms encompass a wide range of operations, including insertion, deletion, searching, sorting, and traversal, among others. The goal is to devise algorithms that leverage the underlying properties and organization of the data structure to optimize time and space complexity. For example, data structures like arrays, linked lists, stacks, queues, trees, and graphs each have their own set of algorithms tailored to their unique characteristics and usage scenarios. Efficient algorithms for data structures are essential for achieving optimal performance and scalability in various applications, enabling efficient data manipulation and retrieval operations. By employing appropriate algorithms for specific data structures, developers can harness the full potential of these structures and unlock efficient solutions for a wide range of computational problems.

Algorithms Using Data Structures

Algorithms using data structures refer to the utilization of specific data structures in combination with well-designed procedures to solve computational problems efficiently. These algorithms leverage the properties and functionality of data structures to store, organize, and manipulate data in a way that optimizes performance and resource utilization. By selecting the appropriate data structure for a given problem and implementing efficient algorithms, it is possible to achieve faster execution times, reduced memory consumption, and improved overall efficiency. Algorithms using data structures encompass a broad range of applications, including searching, sorting, graph traversal, pathfinding, data compression, and more. The synergy between algorithms and data structures is fundamental in computer science, enabling the development of powerful and efficient solutions to complex problems across various domains.

Data Structures Algorithm Example

Here is an example of an algorithm that is using a data structure in C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {5, 2, 7, 1, 3};

    // Sorting the numbers using the std::sort algorithm
    std::sort(numbers.begin(), numbers.end());

    // Searching for a specific number using the std::binary_search algorithm
    int target = 7;
    bool found = std::binary_search(numbers.begin(), numbers.end(), target);

    // Displaying the result
    if (found) {
        std::cout << "The number " << target
        << " is found in the vector." << std::endl;
    } else {
        std::cout << "The number " << target
        << " is not found in the vector." << std::endl;
    }

    return 0;
}
```

In this example, a std::vector is used as the data structure to store a collection of numbers. The std::sort algorithm is employed to sort the numbers in ascending order. Then, the std::binary_search algorithm is utilized to search for a specific number (target) within the sorted vector. The result is displayed based on whether the number is found or not. This example showcases the combination of algorithms (std::sort and std::binary_search) with the data structure (std::vector) to efficiently manipulate and search data, providing a concise and practical illustration of algorithms using data structures in C++.

Sec. 3.3 - Algorithm Efficiency

Algorithm Efficiency

Algorithm efficiency refers to the measure of how well an algorithm performs in terms of time and space usage. It is crucial to assess and analyze the efficiency of algorithms as it directly impacts the overall performance and scalability of a program. Efficiency is commonly evaluated by considering the time complexity, which measures how the algorithm's execution time grows with the input size, and the space complexity, which determines the amount of memory required by the algorithm. The goal is to design and select algorithms that exhibit favorable efficiency characteristics, such as lower time and space complexities, to ensure optimal performance and resource utilization. By employing efficient

algorithms, developers can significantly improve program efficiency, reduce computational costs, and enable the handling of larger datasets and more complex problem instances. Evaluating and optimizing algorithm efficiency is a fundamental aspect of algorithm design and analysis, enabling the development of faster and more scalable solutions in various domains.

Runtime Complexity, Best Case, & Worst Case

Runtime complexity refers to the measure of how the performance of an algorithm scales with the size of the input. It provides insights into the efficiency of an algorithm in terms of time and space usage. The best case runtime complexity represents the lowest possible amount of time an algorithm can take to complete, usually occurring when the input is in the most favorable configuration. On the other hand, the worst case runtime complexity represents the maximum amount of time an algorithm can take to complete, typically occurring when the input is in the least favorable configuration. Analyzing the best and worst case scenarios helps in understanding the upper and lower bounds of an algorithm's performance. By considering both the best and worst case runtime complexities, developers can make informed decisions about the algorithm's efficiency and choose the most suitable algorithm for a given problem, balancing trade-offs between time and space requirements.

Space Complexity

Space complexity refers to the measure of the amount of memory or storage space required by an algorithm to solve a problem. It assesses how the space usage of an algorithm grows with the size of the input. The space complexity of an algorithm is influenced by factors such as the data structures used, the number of variables and their sizes, and any auxiliary space required during the execution. It is commonly expressed in terms of the maximum space used by the algorithm relative to the input size. Analyzing the space complexity helps in understanding the memory requirements of an algorithm and enables the estimation of how much space will be consumed during its execution. By considering the space complexity, developers can optimize memory utilization, minimize unnecessary storage allocation, and ensure the algorithm can handle larger inputs without exhausting available memory resources.

Algorithm Efficiency Example

Below is an example of algorithm efficieny in C++:

```
#include <iostream>
#include <vector>

// Function to find the maximum element in a vector
int findMax(const std::vector<int>& nums) {
    int max = nums[0];
    for (int i = 1; i < nums.size(); ++i) {
        if (nums[i] > max) {
            max = nums[i];
        }
    }
    return max;
}

int main() {
    std::vector<int> numbers = {5, 2, 8, 3, 1};
```

```
        // Find the maximum element in the vector
        int maxNum = findMax(numbers);
        std::cout << "Maximum number: " << maxNum << std::endl;

        return 0;
    }
```

In this example, the findMax function takes a vector of integers as input and returns the maximum element in the vector. It uses a simple linear search algorithm to iterate through the vector and update the maximum element as it encounters larger values. The runtime complexity of this algorithm is $\mathcal{O}(n)$, where n is the size of the input vector. In the best case, when the maximum element is located at the beginning of the vector, the algorithm will terminate early, resulting in a lower execution time. In the worst case, when the maximum element is located at the end of the vector or when all elements are the same, the algorithm will perform the maximum number of comparisons, leading to a higher execution time. As for space complexity, this algorithm requires a constant amount of additional space to store the maximum element and loop variables, resulting in $\mathcal{O}(1)$ space complexity.

By analyzing the runtime complexity, best case, worst case, and space complexity of this example, we can understand the performance characteristics of the algorithm and make informed decisions about its efficiency and suitability for different input scenarios.