Meenakshi Gopalakrishnan
Student ID: 3682020

Assignment 2: Journal

Sources:

For this assignment, I primarily used the Think C++ textbook for reference. In my procedure, I first read through and took notes on chapters 10-14, as indicated in the Study Guide to strengthen my understanding of C++ concepts and complete these assignments. My notes as well as links to other sources I have referenced are listed below.

Think C++ Notes [Chapters 10-14]

Objects and Functions

- Introduction to object-oriented programming in C++.
- Structures correspond to real-world objects or concepts.
- Member functions are functions declared and defined inside a struct definition.

Member Functions

- Member functions are invoked on an object directly.
- Direct access to instance variables within member functions.
- Implicit variable access within member functions improves code readability.
- Non-member functions can be transformed into member functions.
- Use of this pointer for implicit access to instance variables.
- Member functions can access and manipulate the internal state of objects.

Constructors

- Special member functions used for initializing instance variables of objects.
- Constructors have the same name as the class and no return type.
- Invoked automatically when an object is created.
- Constructors initialize objects with default or specific values.

Initialization

- Different ways to initialize structures: using constructors or squiggly-braces.
- Use of constructors becomes mandatory if defined for a structure.

Function Invocation

- Transformation of non-member functions with multiple parameters into member functions.
- Improved encapsulation and more intuitive syntax.

Header files

- Separation of structure definitions and function declarations from implementations.
- Header files contain structure definitions and function declarations.
- Implementation files contain function definitions.
- Separate compilation allows faster compilation of large programs.

Composition

- Composition enables the combination of language features in various arrangements.
- Objects can contain vectors as instance variables, and vectors can contain objects.

Objects and Encapsulation

- Objects encapsulate data and behavior into a single unit.
- Encapsulation allows bundling data (instance variables) and methods (member functions) together.
- Encoding is a way to represent complex data with simpler types.

Vectors of Objects

- Vectors can hold objects of user-defined types.
- A dynamic array that can hold a collection of objects.

Search Algorithms

- Linear Search: Iterating through a vector to find an object. Linear search has a time complexity of O(n).
- Bisection Search: A more efficient search algorithm that requires the vector to be sorted. Bisection search has a time complexity of O(log n).

Abstraction and Parameter Passing

- Abstract Parameter: Treating multiple parameters as a single parameter for function calls.
- Parameter Passing: Passing objects by reference to functions for efficiency and avoiding object duplication.

Enumerated Types

- Enumerated types allow the definition of a set of named constants.
- They provide a way to include a mapping as part of the program and define the set of values that make up the mapping.
- Enumerated types are defined using the enum keyword.
- Each value in the enumerated type is associated with an integer, starting from 0 for the first value.
- Enumerated types make programs clearer and more readable by replacing integer constants with meaningful names.

Switch Statement

- The switch statement provides an alternative to chained conditionals.
- It simplifies code by allowing a variable to be tested for equality against a list of values.
- Syntax: switch (variable) { case value1: ... case value2: ... default: ... }
- Switch statements work with integers, characters, and enumerated types.
- Always include a default case to handle unexpected values.

Encapsulation

- Functional encapsulation: Wrapping up a sequence of instructions in a function to separate its interface from its implementation.
- Data encapsulation: Providing a set of functions that apply to a structure and prevent unrestricted access to its internal representation.

Data Encapsulation

- Hides implementation details from users or programmers who don't need to know them.
- Prevents client programs from accessing the instance variables of an object directly.
- Uses the private keyword to protect parts of a structure definition.

Class

- In C++, a class is a user-defined type with member functions.
- A class is a structure with private instance variables by default.
- ‘private’ keyword is used to make instance variables inaccessible to client programs

Accessor Functions

- Functions that provide access (read or write) to private instance variables.
- Used to maintain data encapsulation and enforce invariants.
- Allow for automatic conversion between different representations.

Preconditions

- Assumptions made about parameters received by a function.

- Preconditions must be true for the function to work properly.
- Functions should check preconditions and handle invalid inputs.
- Postcondition: A condition that is true at the end of a function.

Private Functions

- Member functions used internally by a class but not meant to be invoked by client programs.
- Declared as private to protect them from direct invocation.
- Helps in maintaining data encapsulation and enforcing invariants.

Other sources referenced: https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/

Journal Entry 1: Animal Sounds [Problem 1]

Program Purpose:

The objective of this program is to create an interactive application that demonstrates different animal sounds. By allowing users to select from a predefined list of animals (pig, sheep, duck, cow), the program creates objects representing these animals and shows their respective sounds. This program uses object-oriented programming concepts such as class hierarchies, inheritance, and polymorphism, providing a practical demonstration of how these concepts are applied in real-world software development.

Reflection on Assignment:

Developing the Animal Sounds Program involved creating a base class for animals and derived child classes for specific animals, such as Pig, Sheep, Duck and Cow. Through developing this program, I gained hands-on experience with concepts like class instantiation, method overriding, and dynamic polymorphism. This assignment offered hands-on experience in designing and implementing a modular, object-oriented solution.

Activities Undertaken:

Understanding Requirements:

- Carefully reviewed assignment specifications to understand program behavior.
- Identified the need for a base class (Animal) and derived child classes (Pig, Sheep, Duck, Cow) to represent different animals and their sounds.
- Recognized input/method requirements and implementing extensive input validation.

Design Decisions:

- Opted for a hierarchical class structure with a base class (Animal) and child classes for each animal.
- Designed Animal class with a virtual method for producing animal sounds, allowing for code reuse.
- Created derived classes for each animal, overriding the sound method to produce respective sounds.
- Implemented dynamic memory allocation for creating objects and input reception through terminal.

Implementation:

- Implemented base class Animal() with child classes with sounds specific to each animal (Pig, Sheep, Cow, Duck) or quit
- Implemented clear and descriptive class names with comments for enhanced code readability.
- Paid special attention to input validation to handle invalid user entries and provide informative error messages.
- Used dynamic memory allocation to create objects dynamically, ensuring efficient resource utilization and preventing memory leaks.

Testing and Debugging:

- Conducted testing across various scenarios, including valid and invalid user inputs.
- Verified program correctness and robustness, debugging any encountered issues.
- Ensured smooth program execution and a efficient user experience.

Difficulties Encountered:

Implementing Sound Methods:

- Ensuring that each derived class overrides the sound method correctly was challenging.
- Required careful attention to detail to make sure each animal showed its respective sound respectively.

Input Handling:

- Encountered difficulties in effectively handling user input, especially in validating input against a predefined list of animal types.
- Overcame challenges through iterative refinement of input processing logic and error handling strategies.

Learning Outcomes Fulfilled:

Consolidation of Core Concepts:

- Reinforced understanding of object-oriented programming concepts, including class hierarchies, inheritance, and polymorphism.
- Applied these concepts effectively to design and implement a solution to the assignment requirements.

Enhanced Debugging Skills:

- Improved problem-solving skills through iterative design, implementation, testing, and debugging processes.
- Developed the ability to identify and rectify logical errors and edge cases through rigorous testing and debugging processes.

Sources Used:

Reference for C++ overriding:
https://en.cppreference.com/w/cpp/language/this
https://en.cppreference.com/w/cpp/language/access

Reflection on Programming Process:

- The development of the Animal Sounds Program deepened my understanding of object-oriented design and the importance of class hierarchies.
- Emphasized the significance of user interaction and input validation
- Gained valuable insights into modular, object-oriented programming and developed a more structured approach to software development.

Journal Entry 2: Book Information Program [Problem 2]

Program Purpose:

The objective of this program is to define a Book class with private attributes such as title, author, publisher, ISBN, year, and edition. It provides public getter and setter methods to access and modify these attributes. This program helped me gain experience with the implementation of a class with encapsulated data and methods to manipulate that data.

Reflection on Assignment:

Developing the Book Information Program provided a practical exercise in implementing a class with encapsulated data and methods. It required designing a class to represent book information and implementing methods to access and modify this information. Through this assignment, I gained a deeper understanding of class design, encapsulation, and the importance of data abstraction.

Activities Undertaken:

Understanding Requirements:

- Thoroughly reviewed the assignment specifications to understand the purpose and functionality of the program.
- Identified the need to create a Book class with private attributes and public getter and setter methods to access and modify these attributes.

Design Decisions:

- Opted for a class-based approach to encapsulate book information and provide methods for interaction with this information.
- Designed the Book class with private attributes such as title, author, publisher, ISBN, year, and edition, along with corresponding getter and setter methods.

Implementation:

- Implemented the Book class with appropriate getter and setter methods to retrieve and update book information.
- Ensured adherence to best coding practices, including clear and descriptive method names and meaningful comments for code readability.

Testing and Debugging:

- Tested the Book class by creating several Book objects with different attributes and displaying their information.
- Verified that getter methods correctly retrieved book attributes, and setter methods successfully updated book information.
- Debugged any encountered issues promptly, emphasizing error detection and graceful handling.

Difficulties Encountered:

Data Encapsulation:

- Ensuring proper encapsulation of data and methods presented challenges, particularly in designing an effective class structure.
- Overcame this challenge by carefully designing the Book class with private attributes and public getter and setter methods.

Learning Outcomes Fulfilled:

This problem reinforced my understanding of core programming concepts, including class design, encapsulation, and data abstraction. Through this assignment, I gained a deeper understanding of class-based programming and the importance of data encapsulation in software development.

Reflection on Programming Process:

The development of the Book Information Program was an enriching experience that deepened my understanding of object-oriented programming principles. It highlighted the importance of data encapsulation and abstraction in designing modular and maintainable code. Through this assignment, I gained valuable insights into class-based programming and developed a structured approach to software development.

Journal Entry 3: Elevator Simulation [Problem 3]

Program Purpose:

The objective of this program is to simulate an elevator moving between floors in a building. It utilizes the Elevator class to represent the elevator, with attributes such as currentFloor and totalFloors, and methods moveToFloor, finalize, and getCurrentFloor to simulate elevator operations. By developing this program, I was able to apply core programming concepts and develop proficiency in designing and implementing class-based solutions.

Reflection on Assignment:

By designing and implementing the Elevator class, I gained a deeper understanding of class design, data encapsulation, and method implementation. By implementing the program, I gained a deeper understanding of class-based programming, including class design, implementation, and interaction. Furthermore, testing the Elevator class with various scenarios helped me refine my debugging skills and systematic testing approaches. I encountered and successfully resolved logical errors and edge cases, which significantly contributed to my learning experience.

Activities Undertaken:

Understanding Requirements:

- Thoroughly reviewed the assignment specifications to understand the purpose and functionality of the program.
- Identified the need to create an Elevator class with appropriate attributes and methods to simulate elevator operations.

Implementation:

- Implemented the Elevator class with appropriate methods to move the elevator between floors and return it to the first floor, ensuring that the elevator operated as expected, adhering to the defined requirements.
- Ensured adherence to best coding practices, including clear and descriptive method names and meaningful comments for code readability.

Testing and Debugging:

- Tested the Elevator class by simulating various scenarios of the elevator moving between floors, to verify that the elevator operated correctly under different scenarios.
- Debugged any encountered issues promptly, focusing on precision handling, formatting inconsistencies, and boundary conditions.

Difficulties Encountered:

Simulating Elevator Operations:

- Ensuring accurate simulation of elevator operations presented challenges in designing an effective class structure.
- Mitigated by carefully designing the Elevator class with appropriate methods to move the elevator between floors and return it to the first floor.

Learning Outcomes Fulfilled:

Class Design and Implementation:

- Enhanced skills in class design and implementation by developing the Elevator class to simulate elevator operations.
- Learned to encapsulate functionality within classes to ensure modularity and code reusability.

Efficient Problem-Solving Strategies:

- Employed effective problem-solving strategies to simulate elevator operation scenarios, ensuring comprehensive testing of the Elevator class functionality.

Reflection on Programming Process:

By utilizing constructors, member functions, and access specifiers effectively, I created an effective class structure. Moreover, I developed effective debugging and testing strategies to ensure the correctness and reliability of the simulation program. Through systematic testing under various scenarios, I identified and resolved potential issues, enhancing the overall robustness of the code. This experience also refined my problem-solving skills, as I learned to approach programming challenges with a structured and systematic mindset, breaking down complex problems into manageable components and implementing efficient solutions.

Journal Entry 4: Rodent Behaviour [Problem 4]

Program Purpose:

The objective of this program is to simulate the behavior of different rodents, including Mouse, Gerbil, Hamster, and Guinea Pig. It implements a base class Rodent with virtual functions to simulate eating, sleeping, grooming, and moving behaviors. By developing this program, I deepened my understanding of class inheritance, polymorphism, and virtual functions.

Reflection on Assignment:

By designing and implementing the Rodent class hierarchy and deriving specific rodent classes, I gained practical experience in class inheritance and polymorphism. Testing each rodent's behavior allowed me to ensure that the program behaved as expected and that the derived classes properly implemented the virtual functions.

Activities Undertaken:

Understanding Requirements:

- Thoroughly reviewed the assignment specifications to understand the purpose and functionality of the program.
- Designed a class hierarchy with a base class Rodent and derived classes Mouse, Gerbil, Hamster, and GuineaPig.

Implementation:

- Implemented virtual functions in the Rodent base class to simulate common rodent behaviors: eating, sleeping, grooming, and moving.
- Derived specific rodent classes from the Rodent base class and overrode the virtual functions to implement unique behaviors for each type of rodent.

Testing and Debugging:

- Tested each rodent class to ensure that the virtual functions were overridden correctly and that the behavior of each rodent was simulated accurately.
- Debugged any encountered issues promptly, focusing on ensuring that each rodent's behavior matched its real-world counterpart.

Difficulties Encountered:

Implementing Polymorphism:

- Ensuring that each derived class correctly implemented the virtual functions and that the behavior of each rodent was accurately simulated presented challenges.
- Mitigated by carefully designing the class hierarchy and thoroughly testing each rodent's behavior.

Learning Outcomes Fulfilled:

Understanding Class Inheritance:

- Expanded my comprehension of class inheritance and polymorphism in C++ through the design and implementation of a class hierarchy to simulate rodent behavior.
- Acquired proficiency in effectively using virtual functions to achieve required behavior and in overriding functions in child classes to provide specific implementations.

ITesting and Debugging Skills:

- Strengthened my abilities in testing and debugging by systematically examining the behavior of each rodent and promptly addressing any issues encountered.
- Developed a structured and systematic approach to testing and debugging, ensuring the program's correct behavior across various scenarios.

Sources Used:

https://www.research.psu.edu/animalresourceprogram/experimental-guidelines/rodent-behavior#:~:text=Rodent%20social%20behavior%20may%20be,in%20each%20of%20these%20categories.

Reflection on Programming Process:

This assignment underscored the significance of systematic problem-solving approaches, effective collaboration, and continuous learning. I deepened my technical expertise and cultivated a mindset of adaptability in the face of programming challenges. Through thorough testing and debugging, I ensured the correctness and reliability of the program, further refining my problem-solving skills and enhancing my overall programming proficiency.

Journal Entry 5: Shapes - Area, Circumference, Bounding Box [Problem 5]

Program Purpose:

The primary objective of this program is to define a comprehensive hierarchy of geometric shapes, including Circle, Rectangle, and Triangle. Each shape within the hierarchy provides functionalities to calculate and display its area, circumference, and bounding box. Additionally, the program incorporates robust error handling mechanisms to manage and address scenarios involving invalid shapes. Through the implementation of this program, I aimed to reinforce my understanding of class inheritance, virtual functions, and error handling.

Reflection on Assignment:

By implementing the Shape hierarchy, I not only solidified my understanding of inheritance but also gained practical experience in utilizing virtual functions to calculate shape properties. By defining a base class (Shape) and derived classes (Circle, Rectangle, Triangle), I learned how to leverage inheritance to create a flexible and extensible codebase. By defining virtual functions for calculating area, circumference, and bounding box in the base class, I was able to provide specific implementations for each shape type. This not only improved code organization but also facilitated code reuse. Additionally, by incorporating validation checks within the derived classes, I learned how to effectively handle and communicate errors.

Activities Undertaken:

Understanding Requirements:

- Analyzed the assignment specifications to grasp the desired functionality and constraints of the prime number finder.
- Identified the necessary components needed to create a hierarchy of geometric shapes
- Defined appropriate methods to calculate essential shape properties such as area, circumference, and bounding box.

Implementation:

- Implemented the program logic according to the outlined design decisions, focusing on clarity, simplicity, and efficiency.
- Designed Shape class as a base class, containing virtual functions for calculating the area, circumference, and bounding box of a generic shape, enabling specific implementations to be provided by derived classes.
- Implemented child classes, including Circle, Rectangle, and Triangle, to provide specific implementations for each shape type.
- Child classes overrode the virtual functions defined in the Shape class, ensuring that the calculations are tailored to the properties of each shape.

Testing and Debugging:

- Incorporated error-handling into child classes to validate the shapes' calculations and provide error messages for invalid scenarios.
- Tested various scenarios outlined in the test plan to verify the accuracy of the calculations and error handling mechanisms
- Debugging was performed to address any encountered issues, focusing on precision handling, formatting inconsistencies, and boundary conditions.

Learning Outcomes Fulfilled:

Deepened Understanding of Class Inheritance and Polymorphism:

- Implemented a hierarchical structure of classes (Circle, Rectangle, Triangle) derived from a base class (Shape), demonstrating proficiency in class inheritance.
- Utilized virtual functions to achieve polymorphic behavior, allowing for customized implementations in derived classes to cater to the unique properties of each shape.
- Defined virtual functions (calculateArea(), calculateCircumference(), boundingBox()) in the base class, which were overridden in the derived classes to calculate specific properties of each shape.

Enhanced Testing and Debugging Proficiency:

- Created a comprehensive test plan covering both valid and invalid inputs for each shape class, including scenarios such as negative radii for circles, non-rectangular shapes for rectangles, and zero-length sides for triangles.
- Analyzed test case outputs to identify and resolve potential issues promptly, enhancing debugging proficiency.
- Tackled issues related to class inheritance, function overriding, and error handling, which deepened understanding of object-oriented programming concepts and honed problem-solving skills.

Reflection on Programming Process:

The process of designing and implementing the Shape hierarchy involved careful consideration of class relationships, method overriding, and error handling mechanisms. This required a systematic approach to problem-solving, where I analyzed the unique properties of each shape and defined appropriate virtual functions in the base class. Furthermore, the iterative nature of the development process allowed me to refine my coding practices and debugging techniques. By systematically testing the program under various scenarios, I identified and resolved logical errors and edge cases, enhancing the robustness and reliability of the code. Overall, this assignment not only reinforced my understanding of object-oriented programming concepts but also provided me with valuable experience in designing and implementing class-based solutions.