Student: Michael Chen
ID: MC2023089

Question 1: Implement a simple Calculator class with add, subtract, multiply, and divide operations.

Answer:

```
#include <iostream>
using namespace std;

// Calculator class that performs basic arithmetic operations
class Calculator {
   // Private member variable to store the current result
   private:
      double currentResult;

   // Public methods for calculator operations
   public:
      // Constructor initializes the result to 0
      Calculator() {
         currentResult = 0.0;
      }

      // Method to add a number to the current result
      void add(double operand) {
         currentResult += operand;
      }

      // Method to subtract a number from the current result
      void subtract(double operand) {
         currentResult -= operand;
      }

      // Method to multiply the current result by a number
      void multiply(double operand) {
         currentResult *= operand;
      }

      // Method to divide the current result by a number
      // Checks for division by zero to avoid errors
      void divide(double operand) {
         if (operand == 0) {
            std::cout << "Error: Division by zero is not allowed" << std::endl;
            return;
         }
         currentResult /= operand;
      }
```

```
    // Method to get the current result
    double getResult() const {
        return currentResult;
    }

    // Method to reset the calculator by setting the result to 0
    void clear() {
        currentResult = 0.0;
    }
};

int main() {
    // Create a calculator instance
    Calculator calculator;

    // Demonstrate calculator operations
    std::cout << "Calculator Operations:" << std::endl;
    std::cout << "Starting value: " << calculator.getResult() << std::endl;

    // Perform addition
    calculator.add(25.5);
    std::cout << "After adding 25.5: " << calculator.getResult() << std::endl;

    // Perform subtraction
    calculator.subtract(10.25);
    std::cout << "After subtracting 10.25: " << calculator.getResult() << std::endl;

    // Perform multiplication
    calculator.multiply(2);
    std::cout << "After multiplying by 2: " << calculator.getResult() << std::endl;

    // Perform division
    calculator.divide(5);
    std::cout << "After dividing by 5: " << calculator.getResult() << std::endl;

    // Attempt division by zero
    calculator.divide(0);
    std::cout << "After attempted division by zero: " << calculator.getResult() << std::endl;

    // Clear the calculator
    calculator.clear();
    std::cout << "After clearing: " << calculator.getResult() << std::endl;

    return 0;
}
```

Question 2: Implement a Circle class with methods to calculate area and circumference.
Answer:

```
#include <iostream>
#include <cmath>
using namespace std;

class Circle {
    private:
        double radius; // The radius of the circle
        const double PI = 3.14159265358979323846; // Value of PI

    public:
        // Constructor to initialize the circle with a radius
        Circle(double r) : radius(r) {}

        // Calculate and return the area of the circle
        double calculateArea() {
            return PI * radius * radius;
        }

        // Calculate and return the circumference of the circle
        double calculateCircumference() {
            return 2 * PI * radius;
        }

        // Get the current radius
        double getRadius() const {
            return radius;
        }

        // Update the radius with a new value
        void setRadius(double newRadius) {
            if (newRadius >= 0) {
                radius = newRadius;
            } else {
                cout << "Error: Radius cannot be negative" << endl;
            }
        }
};

int main() {
    // Create circle objects with different radii
    Circle smallCircle(3.0);
    Circle largeCircle(8.5);
```

```
    // Display information about the small circle
    cout << "Small Circle (Radius = " << smallCircle.getRadius() << ")" << endl;
    cout << "Area: " << smallCircle.calculateArea() << " square units" << endl;
    cout << "Circumference: " << smallCircle.calculateCircumference() << " units" << endl;
    cout << endl;

    // Display information about the large circle
    cout << "Large Circle (Radius = " << largeCircle.getRadius() << ")" << endl;
    cout << "Area: " << largeCircle.calculateArea() << " square units" << endl;
    cout << "Circumference: " << largeCircle.calculateCircumference() << " units" << endl;
    cout << endl;

    // Modify the radius of the small circle and display updated information
    smallCircle.setRadius(5.0);
    cout << "Small Circle after modification (Radius = " << smallCircle.getRadius() << ")" <<
endl;
    cout << "Area: " << smallCircle.calculateArea() << " square units" << endl;
    cout << "Circumference: " << smallCircle.calculateCircumference() << " units" << endl;
    cout << endl;

    // Attempt to set a negative radius
    cout << "Attempting to set a negative radius:" << endl;
    smallCircle.setRadius(-2.0);
    cout << "Current radius: " << smallCircle.getRadius() << endl;

    return 0;
}
```

Question 3: Implement a Stack data structure with push, pop, isEmpty, and peek operations.
Answer:

```
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

template <typename T>
class Stack {
    private:
        vector<T> elements;

    public:
        // Push an element onto the stack
        void push(const T& value) {
            elements.push_back(value);
        }
```

```
    // Remove and return the top element
    T pop() {
        if (isEmpty()) {
            throw runtime_error("Cannot pop from an empty stack");
        }

        T top = elements.back();
        elements.pop_back();
        return top;
    }

    // Check if the stack is empty
    bool isEmpty() const {
        return elements.empty();
    }

    // View the top element without removing it
    T peek() const {
        if (isEmpty()) {
            throw runtime_error("Cannot peek an empty stack");
        }

        return elements.back();
    }

    // Get the current size of the stack
    size_t size() const {
        return elements.size();
    }
};

int main() {
    try {
        // Create a stack of integers
        Stack<int> intStack;

        // Print initial state
        cout << "Stack Operations Demonstration:" << endl;
        cout << "Is the stack empty? " << (intStack.isEmpty() ? "Yes" : "No") << endl;

        // Push elements onto the stack
        cout << "Pushing elements: 5, 10, 15, 20, 25" << endl;
        intStack.push(5);
        intStack.push(10);
        intStack.push(15);
        intStack.push(20);
        intStack.push(25);
```

```
        // Show current state
        cout << "Stack size after pushes: " << intStack.size() << endl;
        cout << "Top element: " << intStack.peek() << endl;

        // Pop and display elements
        cout << "Popping and displaying elements: ";
        while (!intStack.isEmpty()) {
            cout << intStack.pop() << " ";
        }
        cout << endl;

        // Check if stack is empty after all pops
        cout << "Is the stack empty after pops? " << (intStack.isEmpty() ? "Yes" : "No") << endl;

        // Try to peek on an empty stack
        cout << "Attempting to peek on an empty stack..." << endl;
        intStack.peek(); // This should throw an exception
    }
    catch (const exception& e) {
        cout << "Exception caught: " << e.what() << endl;
    }

    return 0;
}
```

Question 4: Implement a binary search function that searches for a target value in a sorted array.
Solution:

```
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        // Calculate middle index using unsigned arithmetic to avoid overflow
        int mid = left + (right - left) / 2;

        // Check if target is present at mid
        if (arr[mid] == target) {
            return mid;
        }
```

```
        // If target is greater, ignore left half
        if (arr[mid] < target) {
            left = mid + 1;
        }
        // If target is smaller, ignore right half
        else {
            right = mid - 1;
        }
    }

    // Target is not present in array
    return -1;
}

int main() {
    // Create a sorted vector for testing
    vector<int> numbers = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25};

    cout << "Binary Search Demonstration" << endl;
    cout << "Sorted Array: ";
    for (int num : numbers) {
        cout << num << " ";
    }
    cout << endl;

    // Test cases with expected results
    cout << "Test Cases:" << endl;

    // Case 1: Finding a value that exists in the middle
    int target1 = 13;
    int result1 = binarySearch(numbers, target1);
    cout << "Searching for " << target1 << ": ";
    if (result1 != -1) {
        cout << "Found at index " << result1 << endl;
    } else {
        cout << "Not found" << endl;
    }

    // Case 2: Finding a value that exists at the beginning
    int target2 = 1;
    int result2 = binarySearch(numbers, target2);
    cout << "Searching for " << target2 << ": ";
    if (result2 != -1) {
        cout << "Found at index " << result2 << endl;
    } else {
        cout << "Not found" << endl;
    }
```

```
    // Case 3: Finding a value that exists at the end
    int target3 = 25;
    int result3 = binarySearch(numbers, target3);
    cout << "Searching for " << target3 << ": ";
    if (result3 != -1) {
        cout << "Found at index " << result3 << endl;
    } else {
        cout << "Not found" << endl;
    }

    // Case 4: Finding a value that does not exist
    int target4 = 14;
    int result4 = binarySearch(numbers, target4);
    cout << "Searching for " << target4 << ": ";
    if (result4 != -1) {
        cout << "Found at index " << result4 << endl;
    } else {
        cout << "Not found" << endl;
    }

    return 0;
}
```