ELEN3009 – Software Development II
Class Test 2017 Memo

Question 1

a) Why the doctest framework offers an approximate comparison for floating-point numbers:

- Finite Precision: Floating-point numbers have finite precision due to their binary representation, leading to rounding errors during calculations.
- Small Differences: Tiny differences in least significant bits can cause equality checks (using `==`) to fail unexpectedly.
- `doctest::Approx`: `doctest::Approx` provides a tolerance mechanism to compare within a specified epsilon, accommodating these small differences.

b) Reasons for the deprecation of the C `gets()` function:

- Lack of Bounds Checking: `gets()` lacks buffer size checking, making it prone to buffer overflows.
- Insecurity: The function's insecurity allows input to overwrite memory, posing security risks.
- Safer Alternatives: More secure alternatives like `fgets()` and C++ Standard Library functions offer better control and safety for reading input with defined buffer sizes.

Question 2

a) This program appears to be a guessing game where the player is asked to guess a random number between 1 and 100. The player has up to 5 attempts to guess the correct number, and they receive hints ("Guess smaller" or "Guess bigger") based on the comparison of their guess to the random number. If the player guesses the correct number within 5 attempts, they win; otherwise, they lose.

b) Here are some issues with the code that don't conform with good coding principles and practices:

- Lack of Meaningful Functions: The code is written as a single block within the main function. It's good practice to break down functionality into smaller, meaningful functions for better code organization and readability.
- Magic Numbers: There are magic numbers (e.g., 100, 5) directly used in the code without explanation. These should be defined as named constants to improve code readability and maintainability.
- Inadequate Comments: The code lacks comments explaining its logic and purpose. Comments can greatly help in understanding the code, especially for larger programs or when working in a team.
- Lack of Error Handling: There's no error handling for invalid input, such as non-numeric input. It's a good practice to validate user input and handle potential errors gracefully.
- Hard-to-Follow Logic: The logic for handling guesses and providing hints is embedded within a nested structure, making it somewhat hard to follow. It could benefit from better structure and separation of concerns.

c) Here's a refactored version of the code that addresses the issues and improves its structure by using functions:

```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>

// Constants
const int MAX_ATTEMPTS = 5;
const int MAX_NUMBER = 100;

// Function to generate a random number between 1 and MAX_NUMBER
int generateRandomNumber() {
    return (rand() % MAX_NUMBER) + 1;
}

// Function to provide a hint based on the user's guess
void provideHint(int target, int guess) {
    if (guess > target) {
        std::cout << "Guess smaller." << std::endl;
    }
    else if (guess < target) {
        std::cout << "Guess bigger." << std::endl;
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));

    int targetNumber = generateRandomNumber();
    int totalAttempts = 0;

    std::cout << "Let the game begin... " << std::endl;

    for (int attempt = 1; attempt <= MAX_ATTEMPTS; ++attempt) {
        int userGuess;
        std::cin >> userGuess;
        ++totalAttempts;

        if (userGuess == targetNumber) {
            std::cout << "Congratulations, you win!" << std::endl;
            break;
        }
        else {
            if (totalAttempts == MAX_ATTEMPTS) {
                std::cout << "Sorry, you lose." << std::endl;
            }
            else {
                provideHint(targetNumber, userGuess);
            }
        }
    }

    return 0;
}
```

In this refactored code:

- Constants are defined for maximum attempts and the maximum number.
- Two functions are introduced: **generateRandomNumber** to generate the random target number and **provideHint** to give hints.

- The main logic is organized into functions and follows a clearer structure.
- Comments are added to explain the purpose of functions and the code's overall logic.
- Input validation and error handling are not added here but can be implemented as needed.

Question 3

a) Here's the implementation of the rotate function that rotates the contents of a vector to the left:

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

Using namespace std;

namespace MyUtils {
    template <typename Iterator>
    void rotate(Iterator first, Iterator middle, Iterator last) {
        reverse(first, middle);
        reverse(middle, last);
        reverse(first, last);
    }
}

int main() {
    // Test 1
    vector<int> integers1 = { 1, 2, 3, 4, 5, 6 };
    MyUtils::rotate(integers1.begin(), integers1.begin() + 3, integers1.end());
    vector<int> expected1 = { 4, 5, 6, 1, 2, 3 };
    if (integers1 == expected1) {
        cout << "Test 1 Passed" << std::endl;
    }
    else {
        cout << "Test 1 Failed" << std::endl;
    }

    // Test 2
    vector<int> integers2 = { 1, 2, 3, 4, 5 };
    MyUtils::rotate(integers2.begin(), integers2.begin() + 2, integers2.end());
    vector<int> expected2 = { 3, 4, 5, 1, 2 };
    if (integers2 == expected2) {
        cout << "Test 2 Passed" << std::endl;
    }
    else {
        cout << "Test 2 Failed" << std::endl;
    }

    // Test 3
    vector<int> integers3 = { 1, 2, 3, 4, 5 };
    MyUtils::rotate(integers3.begin(), integers3.begin(), integers3.end());
    vector<int> expected3 = { 1, 2, 3, 4, 5 };
    if (integers3 == expected3) {
        cout << "Test 3 Passed" << std::endl;
    }
    else {
        cout << "Test 3 Failed" << std::endl;
    }

    // Test 4
    vector<int> integers4 = { 1, 2, 3, 4, 5 };
    MyUtils::rotate(integers4.begin(), integers4.end() - 1, integers4.end());
    vector<int> expected4 = { 5, 1, 2, 3, 4 };
    if (integers4 == expected4) {
        cout << "Test 4 Passed" << std::endl;
    }
    else {
        cout << "Test 4 Failed" << std::endl;
    }

    // Test 5
    vector<int> integers5 = { 1, 2, 3, 4, 5 };
    MyUtils::rotate(integers5.begin(), integers5.end() + 2, integers5.end());
    vector<int> expected5 = { 1, 2, 3, 4, 5 };
    if (integers5 == expected5) {
        cout << "Test 5 Passed" << std::endl;
    }
    else {
        cout << "Test 5 Failed" << std::endl;
    }

    return 0;
}
```

b) The behavior of the STL's rotate algorithm with an invalid rotation point being supplied is undefined for a few reasons:

- Performance Overhead: Signaling an error to the calling code for every invalid rotation point would introduce a significant performance overhead, especially when dealing with large data sets. This is because it would require additional checks and conditional statements for each rotation operation.
- Flexibility: The STL aims to provide a flexible and efficient set of algorithms for various use cases. Allowing undefined behavior in this case allows the algorithm to be as efficient as possible for valid inputs, without the need for extra error-checking code.
- Legacy Compatibility: Many STL algorithms are used in performance-critical applications, including those with legacy codebases. Changing the behavior of an algorithm like rotate to signal an error for invalid inputs could break existing code that relies on the current behavior.
- Responsibility Shift: In C++, the responsibility for providing valid inputs to algorithms generally falls on the programmer. It is assumed that the programmer will ensure that the inputs are valid before using the algorithm. This aligns with the philosophy of C++ to not impose overhead on operations that should be rare or avoidable through proper programming practices.

While undefined behavior can lead to unexpected results or program crashes, it's ultimately the programmer's responsibility to ensure that inputs to algorithms are valid. This approach favors performance and flexibility, which are core principles of the C++ Standard Library.