ELEN3009 – Software Development II
Class Test 2022 Memo

Question 1

a) Here's a graphical representation of the commit history based on the sequence of Git commands in Listing 1:

```
Master:   o --- o --- o
                   \
Feature:            o --- o --- o
HEAD is on the 'master' branch.
```

- Commits are represented by circles labelled with commit messages.
- The 'master' branch initially contains three commits.
- The 'feature' branch starts from the last commit on 'master' and contains three additional commits.
- 'HEAD' is currently on the 'master' branch.

b) **Fast-Forward Merge**: A fast-forward merge is a type of merge operation in Git where the branch being merged has all its commits directly accessible from the branch being merged into. In other words, there are no new commits on the target branch since the branch you are merging started. In a fast-forward merge, Git simply moves the branch pointer forward to include the commits from the other branch, creating a linear history.

c) No, a fast-forward merge will not take place if you input "git merge feature" after the sequence in Listing 1. This is because commits have been made on the 'feature' branch after it was created, and these commits are not present in the 'master' branch. Fast-forward merges occur when there are no new commits on the target branch since the branch being merged started. In this case, a regular merge (non-fast-forward) would be performed to incorporate the changes from 'feature' into 'master'.

Question 2

Listing 2 defines a **Person** class with a constructor, copy constructor, and destructor. Each of these functions prints a message when called.

Listing 3 contains a **main** function that calls the **newPerson** function, which returns a **Person** object.

Here's the expected output and an explanation for each line:

1. "Constructing a person" - This message is printed when a **Person** object is created inside the **newPerson** function because of the line **return Person{"Bobby"};**.

2. "Destructing a person" - This message is printed when the **Person** object created in the **newPerson** function goes out of scope at the end of the function. The object's destructor is called.
3. No message - The **Person** object created in the **newPerson** function is returned but not stored in a variable in the **main** function. Therefore, it goes out of scope immediately, and its destructor is called, but no message is printed because it's out of the scope of **main**.

So, the output of the main program will be:

```
Constructing a person

Destructing a person
```

The "Constructing a person" message is printed when the **Person** object is created, and the "Destructing a person" message is printed when the object's destructor is called as it goes out of scope.

Question 3

a) You can solve this problem without using if statements, switch statements, or loops by leveraging the power of the Standard Template Library (STL) and C++ algorithms. One way to achieve this is by using the **remove_if** and **erase** functions. Here's the **remove_vowels** function:

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;
// Function to check if a character is a vowel (lowercase or uppercase)
bool is_vowel(char c) {
    c = tolower(c); // Convert to lowercase for case insensitivity
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}

// Function to remove vowels from a string
string remove_vowels(string str) {
    str.erase(remove_if(str.begin(), str.end(), is_vowel), str.end());
    return str;
}

int main() {
    string sentence = "This sentence contains a number of vowels.";
    string result = remove_vowels(sentence);

    cout << "Original sentence: " << sentence << endl;
    cout << "Result after removing vowels: " << result << endl;

    return 0;
}
```

In this code:

1. The **is_vowel** function checks if a character is a vowel, considering both lowercase and uppercase vowels. It uses **std::tolower** for case insensitivity.
2. The **remove_vowels** function takes a string as input, uses **std::remove_if** to remove all characters that are vowels, and then erases the removed characters from the string.
3. In the **main** function, we demonstrate how to use **remove_vowels** to remove vowels from a sentence.

You can use this **remove_vowels** function with the test cases you provided in Listing 4 to verify its correctness.

b) Avoiding the manual implementation of branching statements (like **if** statements) and loops (like **for** or **while** loops) in your code can have several benefits:

1. **Improved Readability**: Using higher-level abstractions provided by the language or libraries makes your code more readable. Higher-level constructs often express your intent more clearly, which can be beneficial for you and other developers who read and maintain the code.
2. **Reduced Complexity**: Writing your own branching and looping logic can make code more complex and error-prone. Using built-in constructs simplifies your code and reduces the likelihood of logic errors.
3. **Increased Maintainability**: Code with fewer custom loops and branches is typically easier to maintain. When you use language constructs, you rely on tested and standardized behavior, reducing the need for extensive testing and debugging.
4. **Code Reusability**: Standard constructs can be easily reused in different parts of your codebase or in other projects. Writing custom loops and branches for each situation can lead to duplicated code and maintenance challenges.
5. **Performance Optimization**: Compiler and library implementers have optimized standard constructs to run efficiently on various platforms. Using these constructs allows your code to take advantage of these optimizations.
6. **Cross-Platform Compatibility**: Standard constructs are designed to work consistently across different platforms and compilers. Writing custom branching and looping logic may introduce platform-specific issues.
7. **Compatibility with Code Analysis Tools**: Many code analysis and debugging tools are designed to work with standard constructs. Custom logic may not be as compatible with these tools.
8. **Reduced Bugs**: Standard constructs are widely used and well-tested, reducing the likelihood of introducing subtle bugs in your code.
9. **Easier Debugging**: Code that uses standard constructs is typically easier to debug because it follows familiar patterns and idioms. Debugging custom logic can be more challenging.

10. **Conciseness**: Standard constructs often allow you to express complex logic concisely, which can lead to shorter and more maintainable code.

Overall, while there may be situations where custom branching and looping logic are necessary, relying on standard constructs provided by the language and libraries is a best practice in most cases. It leads to more maintainable, readable, and bug-free code and allows you to take advantage of the optimizations and compatibility features provided by the development environment.