SC1008 Lab 4 of C++ – Inheritance, Templates and STL

1. **(Multiple Inheritance) [10 marks]** Create two base classes—`Sports` and `Academics`—and a derived class `StudentAthlete` that inherits from both. The `Sports` class will store sports-related information, and the `Academics` class will store academic information. The derived `StudentAthlete` class will combine these with additional details about the student, such as their name, age, and studentID. The members you should define in each class is as follows:
   - Class `Sports`
     - Attribute: `string sport`
     - Constructor: `Sports(string s)`
     - Destructor: `~Sports()`
     - Display sport info function: `void displaySports() const`
   - Class `Academics`
     - Attribute: `float gpa`
     - Constructor: `Academics(float g)`
     - Destructor: `~Academics()`
     - Display academic info function: `void displayAcademics() const`
   - Class `StudentAthlete` inherits from the above two classes
     - Attributes:
       ```
       string name;
       int age;
       int studentID;
       ```
     - Constructor: `StudentAthlete(string s, float g, string n, int a, int id)`
     - Destructor: `~StudentAthlete()`
     - Display information function: `displayInfo()`

   You are asked to use the following starting code, fill the missing parts, and print out the sample output.

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

// Base class: Sports
class Sports {
protected:
    string sport; // sport name
public:
    // TO-DO 1: Implement the constructor and descructor
    //

    // Display sports information
    void displaySports() const {
        cout << "Sport: " << sport << endl;
    }
};
```

```
// Base class: Academics
class Academics {
protected:
    float gpa; // GPA
public:
    // TO-DO 2: Implement the constructor and destructor
    //

    // Display academic information
    void displayAcademics() const {
        cout << "GPA: " << gpa << endl;
    }
};

// TO-DO 3: Implement the Derived class: StudentAthlete, which inherits from both
Academics and Sports


int main() {
    //Create a StudentAthlete instance
    StudentAthlete stu("Basketball", 3.8, "Bob", 21, 1001);

    // Display all the information
    cout<<endl;
    stu.displayInfo();
    cout<<endl;

    // Test code to demonstrate multiple inheritances
    stu.displaySports();
    stu.displayAcademics();
    cout<<endl;

    return 0;
}
```

Sample output:

```
Sports constructor!
Academics constructor!
StudentAthlete constructor!

Name: Bob, Age: 21, Student ID: 1001
Sport: Basketball
GPA: 3.8

Sport: Basketball
GPA: 3.8

StudentAthlete destructor!
```

```
Academics destructor!
Sports destructor!
```

2. **(std::list) [10 marks]** You are helping a small clinic to implement its patient queue management system. Patients arrive and get added to the end of the queue. When the doctor is ready, the patient at the front of the queue is served and removed from the patient queue. You are asked to use `std::list<std::string>` to implement this queue, and implement a class called `PatientQueue` that has the following members:

    - **Private Members:**
        - `std::list<std::string> queue`: a list of patient names.
    - **Public Members:**
        - A constructor and destructor.
        - A function `addPatient(const std::string &name)` that adds a new patient's name to the back of the list.
        - A function `servePatient()` that removes the patient from the front of the list and returns their name. If the list is empty, return a reminder message "Empty queue!".
        - A function `isEmpty()` that returns true if the list is empty, false otherwise.
        - A function `displayQueue()` that prints out all patients in the linked list in order.

Below is the starting code with missing implementation that you can fill in:

```
#include <iostream>
#include <list>
#include <string>

class PatientQueue {
private:
    std::list<std::string> queue; // Stores patient names
public:
    // Constructor: Initializes an empty patient queue
    PatientQueue() : queue() {}
    ~PatientQueue(){}

    // TO-DO 1: Implement the function addPatient to add a new patient's name to the end of the queue
    //
```

```
    // TO-DO 2: Implement the function servePatient: removes and returns the patient at the
front of the queue
    // If the queue is empty, returns a reminder message
    //

    // TO-DO 3: Implement the function isEmpty
    //

    // TO-DO 4: Implement the function displayQueue to print out all patients in order
     //

};

int main() {
    PatientQueue clinicQueue;

    // Test 1: Add patients
    clinicQueue.addPatient("Alice");
    clinicQueue.addPatient("Bob");
    clinicQueue.addPatient("Charlie");

    std::cout << "Current queue: ";
    clinicQueue.displayQueue();
    std::cout<<std::endl;

    // Test 2: Serve a patient
    std::cout << "Serving patient: " << clinicQueue.servePatient() << std::endl;
    std::cout << "Queue after serving: ";
    clinicQueue.displayQueue();
    std::cout<<std::endl;

    // Test 3: Serve more until the queue is empty
    std::cout << "Serving patient: " << clinicQueue.servePatient() << std::endl;
    std::cout << "Serving patient: " << clinicQueue.servePatient() << std::endl;
    std::cout<<std::endl;

    // Test 4: Check if it is empty
    if (clinicQueue.isEmpty()) {
        std::cout << "No patient in the queue now" << std::endl;
    }
    std::cout << "Serving patient: " << clinicQueue.servePatient() << std::endl;

    return 0;
}
```

Expected output:

```
Current queue: Alice Bob Charlie

Serving patient: Alice
Queue after serving: Bob Charlie

Serving patient: Bob
Serving patient: Charlie

No patient in the queue now
Serving patient: Empty queue!
```