SC1008 Tutorial 4 – Inheritance, Template and STL

1. **(Inheritance and Redefining a Base Class Function)** Create a base class `Person` with attributes for `name` and `age`. Then, create a derived class `Student` that adds a `studentID` attribute.

   Header/Source Separation: Define the class interfaces in header files (.h) with include guards (or `#pragma once`) and implement the member functions in separate source files (.cpp).

   The details for members of the two classes are as follows:
   Base Class `Person`
   - Attributes:
     - `string name`
     - `int age`
   - Constructor: `Person(string n, int a)`
   - Function: `void displayInfo() const`

   Derived Class `Student` (inherits from `Person`)
   - Attribute: `studentID`
   - Constructor: `Student(string n, int a, int id)`
   - Function: `void displayInfo() const` (override the base class function)

Below is the starting code with missing implementation for you to fill in:

Person.h

```
#ifndef PERSON_H
#define PERSON_H

#include <string>
using namespace std;

class Person {
protected:
    // TODO: Define the member variables (name and age)
public:
    // Constructor declaration
    Person(string n, int a);

    // Function to display person details
    void displayInfo() const;
};

#endif // PERSON_H
```

Person.cpp

```cpp
#include "Person.h"
#include <iostream>
using namespace std;

// TODO: Implement Person class constructor and function here
Person::Person(string n, int a) {
    // TODO: Initialize member variables
}

void Person::displayInfo() const {
    // TODO: Display person's details
}
```

Student.h

```cpp
#ifndef STUDENT_H
#define STUDENT_H

#include "Person.h"
#include <string>
using namespace std;

class Student : public Person {
private:
    // TODO: Define the additional attribute (studentID), studentID
is an integer


public:
    // Constructor declaration
    Student(string n, int a, int id);

    // Function to display student details (redefine base class
function)
    void displayInfo() const;
};

#endif // STUDENT_H
```

Student.cpp

```cpp
#include "Student.h"
#include <iostream>
using namespace std;
```

```
// TODO: Implement Student class constructor and initialize studentID
Student::Student(string n, int a, int id)
    : Person(n, a) {

}

void Student::displayInfo() const {
    // TODO: Output student information
}
```

Main.cpp

```
#include "Student.h"
#include <iostream>
using namespace std;

int main() {
    Student s1("Alice", 20, 12345);
    s1.displayInfo();
    cout<<endl;

    // A base class pointer points to the derived class object
    Person* p = &s1;
    p->displayInfo();

    return 0;
}
```

Sample output:

```
Name: Alice, Age: 20
Student ID: 12345

Name: Alice, Age: 20
```

2. (Multilevel Inheritance and Overriding a Base Class Function) Extend the Student class in Question 1 to create a `GraduateStudent` class, which has an additional attribute `researchTopic`. The details of class `GraduateStudent` (inherits from `Student`) are as follows:

- New attribute: `string researchTopic`
- Constructor: `GraduateStudent(string n, int a, int id, string topic)`
- Function: `void displayInfo()` (overrides the base class function)

Your tasks are as follows:

1) Implement the `Person` and `Student` class within the given starting code file, i.e., *implement their member functions within the class definition*, given that the two classes are simple.
2) Update the `displayInfo()` function of the `Person` and `Student` class as a virtual function, so that the class `GraduateStudent` can correctly override this function.
3) Finish the definition of the class `GraduateStudent` to ensure it can be used and run correctly by following the sample output.

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

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


// TODO: Update your implementation for Student Class and Person Class in
Question 1 below
//       Declare displayInfo() as virtual




// Derived class: GraduateStudent
class GraduateStudent : public Student {
private:
    // TODO: Define the additional attribute (researchTopic)

public:
    // TODO: Implement the Constructor
    GraduateStudent(string n, int a, int id, string topic);


    // TODO: Implement displayInfo() (Note: it is virtual function in Student)


    virtual void displayInfo() const;
};


int main() {
    GraduateStudent gs1("Alice", 25, 56789, "Machine Learning");
    gs1.displayInfo();
    cout<<endl;

    Student* stu = &gs1;
    stu->displayInfo();
    cout<<endl;

    Person* per = &gs1;
    per->displayInfo();
```

```
    return 0;
}
```

Sample output:

```
Name: Alice, Age: 25, Student ID: 56789, Research Topic: Machine Learning

Name: Alice, Age: 25, Student ID: 56789, Research Topic: Machine Learning

Name: Alice, Age: 25, Student ID: 56789, Research Topic: Machine Learning
```

3. (Template Class) Imagine you are building a software module for processing students' exam results. In different courses, the exam results can be represented in different formats—for instance, letter grades (e.g., "A", "B", "C"), integer scores (e.g., 80, 85), floating-point numbers (e.g., 80.5, 91.7), or as boolean values indicating pass/fail. To avoid writing separate classes for each type of result, you are asked to implement a generic container template, `ExamResult<T>`. This class will store a single exam result of any data type and track whether a result has been stored. Create a template class `ExamResult<T>` that has the following members

- **Private Members**
  - `T* result`: the pointer pointing to the array storing exam result (e.g., T could be an `int`, `std::string`, etc.).
  - `int size`: the size of the array pointed to by the pointer result.
- **Public Members**
  - A default constructor without any parameters initializing the private member variables as their default values (i.e., set `size` as 0, and `result` as a NULL pointer)
  - `setExamResult(T* array, int len)` initializing private member variables (i.e., sets `result` as `array`, `size` as `len`)
  - `updateResultAtOneLoc(int i, const T &newResult)`: Update the exam result at i-th location. If the index i exceeds the size of the array size, output an error message "Out of the size limit!"
  - `printExamResult()` print out all the exam result. If it is empty, print out "No exam results!".
  - The destructor clears the stored exam result (clear the dynamically-allocated memory `result`, and set size as 0).

Below is the starting code with missing implementations for you to implement.

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

// Template class for storing exam results of different data types
template <typename T>
class ExamResult {
private:
```

```
    T* result; // Pointer to dynamically allocated array of exam results
    int size;  // Number of exam results

public:
    // Default constructor
    ExamResult() : result(nullptr), size(0) {}

    void setExamResult(T* array, int len) {
        // TO-DO: Set the exam result as the input array
        //

    }


    void updateResultAtOneLoc(int i, const T &newResult) {
        // TO-DO: Update the exam result at i-th location to new result
        //
        //


    }


    void printExamResult() const {
        // TO-DO: Print all exam results
        //

    }


    ~ExamResult() {
        // TO-DO: Destructor to free allocated memory
        //

    }
};


int main() {
    // Test with integer scores
    int intScores[] = {80, 90, 75, 85};
    ExamResult<int> intExam;
    intExam.setExamResult(intScores, 4);
    intExam.printExamResult();
    intExam.updateResultAtOneLoc(2, 95);
    intExam.printExamResult();
    cout<<endl;

    int intScores2[] = {100, 99};
    intExam.setExamResult(intScores2, 2);
    intExam.printExamResult();
    cout<<endl;
```

```
    // Test with letter grades
    string letterGrades[] = {"A", "B", "C", "D"};
    ExamResult<string> stringExam;
    stringExam.setExamResult(letterGrades, 4);
    stringExam.printExamResult();
    stringExam.updateResultAtOneLoc(3, "A+");
    stringExam.printExamResult();
    cout<<endl;

    // Test with boolean pass/fail results
    cout << boolalpha; // Enables printing "true" and "false"
    bool passFail[] = {true, false, true};
    ExamResult<bool> boolExam;
    boolExam.setExamResult(passFail, 3);
    boolExam.printExamResult();
    boolExam.updateResultAtOneLoc(0, false);
    boolExam.printExamResult();

    return 0;
}
```

Expected output:

```
80 90 75 85
80 90 95 85

100 99

A B C D
A B C A+

true false true
false false true
```

4. (STL vector) A small café wants to track its daily sales for one week. Each day's sales figure is an integer. You are asked to use an STL vector container, i.e., `std::vector<int> dailySales` to track the daily sales. Please perform the following operations in the `main()` function:

- Add Sales: Insert seven daily sale values into the vector.
- Print Sales: Traverse and print all sales values using an iterator.
- Calculate Average: Compute and display the average of the sales figures.
- Sort Sales: Sort the sales in ascending order using std::sort and then print the sorted list using an iterator.

Here is the sample code for you to get started:

```
#include <iostream>
```

```cpp
#include <vector>
#include <algorithm> // for std::sort
#include <numeric>   // for std::accumulate

int main() {
    // Declare a vector to store daily sales.
    std::vector<int> dailySales;

    // TO-DO: Add seven daily sales values to the vector:
    //         120, 200, 150, 80, 90, 220, 100
    //
    //

    // TO-DO: Print all sales values by using an iterator
    //
    //


    // TO-DO: Calculate the average of the sales values and print it
    //
    //



    // TO-DO: Sort the vector in ascending order using std::sort.
    //


    // TO-DO: Print all the sorted sales values by using an iterator
    //
    //


    return 0;
}
```

Expected output:

```
Daily Sales: 120 200 150 80 90 220 100
Average Sales: 137.143
Sorted Sales: 80 90 100 120 150 200 220
```