SC1008 Tutorial 4 – Solution

Note: **The parts highlighted in red+bold** are the key check points for us to explain to students.

1. **(Inheritance and Redefining a Base Class Function)**

Person.h

```h
#ifndef PERSON_H
#define PERSON_H

#include <string>
using namespace std;

class Person {
protected:
    // Define the member variables (name and age)
    string name;
    int 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>

Person::Person(std::string n, int a)
    : name(n), age(a) {
    // Instead of using the initialization list, we can also initialize the
member variables within the function block
}

void Person::displayInfo() const {
    std::cout << "Name: " << name << ", Age: " << age << std::endl;
}
```

Student.h

```h
#ifndef STUDENT_H
#define STUDENT_H
```

```
#include "Person.h"

class Student : public Person {
private:
    // Define the additional attribute (studentID)
    int studentID;

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

    // Function to display student details
    // Note: displayInfo() is NOT a virtual function in the base class, so we are
    // redefining the base class function, NOT function overriding)
    void displayInfo() const;
};

#endif // STUDENT_H
```

Student.cpp

```
#include "Student.h"
#include <iostream>

Student::Student(string n, int a, int id)
    : Person(n, a), studentID(id) {
    // Additional initialization if needed.

        //We can also initialize studentID within the constructor function body,
i.e.,
        // studentID = id;
}

void Student::displayInfo() const {
    // Option 1: Call base class displayInfo() then output studentID
    Person::displayInfo();
    std::cout << "Student ID: " << studentID << std::endl;

    // Option 2: Alternatively, display everything together:
    // std::cout << "Name: " << name << ", Age: " << age << ", Student ID: " <<
studentID << std::endl;
}
```

Main.cpp

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

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

    // Base class pointer points to the derived class object
    Person* p = &s1;

    // Call displayInfo() using the base class pointer.
    // Here displayInfo() is NOT declared as virtual in Person (Redefining the base class function, not function overriding), the Persion version will be called.
    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)

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

// Base class: Person
class Person {
protected:
    string name;
    int age;
public:
    Person(string n, int a) : name(n), age(a) {}

    // Declare displayInfo() as virtual so it can be overridden.
    virtual void displayInfo() const {
        cout << "Name: " << name << ", Age: " << age;
    }
};

// Derived class: Student (inherits from Person)
class Student : public Person {
protected:
    int studentID;
public:
    Student(string n, int a, int id) : Person(n, a), studentID(id) {}
```

```
    // Mark displayInfo() as virtual; it overrides Person::displayInfo()
    virtual void displayInfo() const override {
        Person::displayInfo();
        cout << ", Student ID: " << studentID;
    }
};

// Derived class: GraduateStudent (inherits from Student)
class GraduateStudent : public Student {
private:
    string researchTopic; // Additional attribute for GraduateStudent
public:
    // Constructor
    GraduateStudent(string n, int a, int id, string topic)
        : Student(n, a, id), researchTopic(topic) {}

    // Override displayInfo() to include researchTopic information.
    virtual void displayInfo() const override {
        Student::displayInfo();
        cout << ", Research Topic: " << researchTopic << endl;
    }
};

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;
}
```

Commented [AWY1]: We achieve polymorphism here

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)

```cpp
#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) {}

    // Set exam results
    void setExamResult(T* array, int len) {
        // Free any existing memory before assigning new values
        if(result != nullptr)
            delete[] result;

        size = len;
        result = new T[size]; // Allocate new memory
        for (int i = 0; i < size; ++i) {
            result[i] = array[i];
        }
    }

    // Update a specific exam result
    void updateResultAtOneLoc(int i, const T &newResult) {
        if (i < 0 || i >= size) {
            cout << "Out of the size limit!" << endl;
            return;
        }
        result[i] = newResult;
    }

    // Print all exam results
    void printExamResult() const {
        if (size == 0) {
            cout << "No exam results!" << endl;
            return;
        }
        for (int i = 0; i < size; ++i) {
            cout << result[i] << " ";
        }
        cout << endl;
    }

    // Destructor to free allocated memory
    ~ExamResult() {
```

Commented [AWY2]: Some students may forget to free the memory

```
        delete[] result;
        result = nullptr;
        size = 0;
    }
};


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)

```
#include <iostream>
#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;

    // Add seven daily sales values to the vector:
    //         120, 200, 150, 80, 90, 220, 100
    dailySales.push_back(120);
    dailySales.push_back(200);
    dailySales.push_back(150);
    dailySales.push_back(80);
    dailySales.push_back(90);
    dailySales.push_back(220);
    dailySales.push_back(100);

    // Print all sales values by using an iterator
    std::cout << "Daily Sales: ";
    for (std::vector<int>::iterator it = dailySales.begin(); it !=
dailySales.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // Calculate the average of the sales values and print it.
    int sum = std::accumulate(dailySales.begin(), dailySales.end(), 0);
    double average = static_cast<double>(sum) / dailySales.size();
    std::cout << "Average Sales: " << average << std::endl;

    // Sort the vector in ascending order using std::sort.
    std::sort(dailySales.begin(), dailySales.end());

    // Traverse the sorted vector using an iterator and print the sorted sales
values.
    std::cout << "Sorted Sales: ";
    for (std::vector<int>::iterator it = dailySales.begin(); it !=
dailySales.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
```

Commented [AWY3]: STL container iterator

Commented [AWY4]: C++ style type conversion, which is similar to “(double) sum” in C

```
    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
```