Question #1:

a. Do you agree with the statement: “When a function is declared a friend by a class, it becomes a member of that class”? Justify your answer.

Ans : No. When a function is declared a friend by a class, it does not become a member of that class. Instead, it gains special access privileges to the private and protected members of that class.

b. Can a constant member function be overloaded with a non-constant version?

Ans : Yes. When you overload a constant member function with a non-constant version, you are essentially providing two different versions of the function: one that can be called on constant objects (const member function) and another that can be called on non-constant objects (non-const member function).

c. Can the diamond problem be mitigated explicitly by disambiguating member function calls in the derived class?

Ans : Yes, the diamond problem in C++ can be mitigated explicitly by disambiguating member function calls in the derived class. One way to resolve this ambiguity is by explicitly specifying which version of the member function to call using the scope resolution operator (::). By specifying the base class name followed by the scope resolution operator, you can clarify to the compiler which version of the member function you intend to use.

d. What will be the order of constructors and destructors in the code snippet given below:

```
class A { };

class B { };

class C : public B, public A { };

class D : public C { };

main() { D d1; }
```

Constructor: B, A, C, D

Destructor: D, C, B, A

e. Is there any problem in a code snippet given below? If yes, how can we resolve it?

```
class A { public: int x;};

class B : private A { };

class C : public B { public: C() {x = 10; } };

main() { C c1;}
```

```
class A { public: int x; };
class B : private A {
public:
   int getX() { return x; }
   void setX(int val) { x = val; }
};
class C : public B {
public:
   C() { setX(10); }
};
```

Other solution may exist.

Question #2:

```
class Employee{
        protected:
                string name;
                int empID;
                double salary;

                public:
                        Employee(string name, int empID, double salary){
                                this -> name = name;
                                this -> empID = empID;
                                this -> salary = salary;
                        }
};

class SecurityAnalyst:virtual public Employee{
        public:
                string specialization;
                bool isCertified;

                SecurityAnalyst(string n, int ID, double s, string specialization, bool isCertified):
Employee(n,ID, s){
                        setSpecialization(specialization);
                        this -> isCertified = isCertified;
```

```
		}

		void setSpecialization(string s){
			if(s != "Network" || s != "Incident Response" || s != "Threat
Intelligence"){
				cout << "Enter specialization again";
				cin >> s;
				specialization = s
			}
			else
			specialization = s;
		}

		void HasCertifications(){
			int noofCerts;
			string certs;
			if(isCertified){
				cout << "Enter no of certs";
				cin >> noofCerts;
				if (noofCerts == 1){
					cout << "Enter your certification";
					cin >> certs;
					cout << certs;
				}
				else if (noofCerts > 1){
					string certifications[noofCerts];
					int i;
					for(i = 0; i<noofCerts;i++){
						cin >> certifications[i];
					}

					for(i = 0; i<noofCerts;i++){
						cout << "Displaying";
						cout << certifications[i] << endl;
					}
				}
			}
			else
			cout << "Not Certified";

		}
};
class SecEngineer : virtual public Employee{
	public:
```

```
        bool isExpert;
        bool isAssigned;
        string* list;

        SecEngineer(string n, int ID, double s, bool isExpert, bool isAssigned):
Employee(n,ID,s){
                this -> isExpert = isExpert;
                isAssigned = false;
                list = new string {"Project 1"};
        }

        bool AssignProject(){
                if(isExpert){
                        isAssigned = true;
                        return isAssigned;
                }
                else{
                        isAssigned = false;
                        return isAssigned;
                }
        }

        void AddProjects(){
                if(AssignProject()){
                        int noofProjects,i;
                        cin >> noofProjects;
                        string* list1 = new string[noofProjects];
                                for(i = 0; i< 1;i++){
                                list1[i] = list[i];
                }

                        for(i = 1; i< noofProjects;i++){
                                cout << "Add Projects";
                                cin >> list1[i];
                }
        }
                else
                cout << "No projects to add";
        }
};
class Lead: public SecurityAnalyst, public SecEngineer{
```

```
        public:
                Lead(string a,int ID, double s, string specialization, bool isCertified,bool isExpert,
bool isAssigned):SecurityAnalyst(a,ID,s,specialization,isCertified),
                SecEngineer(a,ID,s,isExpert,isAssigned),Employee(a,ID,s)
                {
                }

                void show(){
                        HasCertifications();
                        AssignProject();
                        AddProjects();
                }
};
```

Question #3:

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

using namespace std;

class Pastry {
  friend double calculateTotalCostWithTax(const Pastry& pastry);
protected:
  string name;
  string ingredients;
  double productionCost;

public:
  Pastry(const string& pastryName, const string& pastryIngredients, double cost)
    : name(pastryName), ingredients(pastryIngredients), productionCost(cost) {}

  double calculateTotalCost() const {
    return productionCost * 1.10; // 10% markup for labor and expenses
  }

  virtual double calculateRetailPrice() const {
    return calculateTotalCost(); // No tax by default
  }
```

```
    virtual double calculateProfit() const {
        double retailPrice = calculateRetailPrice();
        double profit = (retailPrice - calculateTotalCost()) * 0.70; // Shop retains 70% profit
        return profit;
    }
};

class SweetPastry : public Pastry {
private:
    double salesTaxRate; // Tax rate for sweet pastries

public:
    SweetPastry(const string& pastryName, const string& pastryIngredients, double cost, double taxRate)
        : Pastry(pastryName, pastryIngredients, cost), salesTaxRate(taxRate) {}

    double calculateRetailPrice() const override {
        double totalCost = calculateTotalCost();
        double retailPrice = totalCost * (1 + salesTaxRate);
        return retailPrice;
    }
};

class SavoryPastry : public Pastry {
private:
    double salesTaxRate; // Tax rate for savory pastries

public:
    SavoryPastry(const string& pastryName, const string& pastryIngredients, double cost, double taxRate)
        : Pastry(pastryName, pastryIngredients, cost), salesTaxRate(taxRate) {}

    double calculateRetailPrice() const override {
        double totalCost = calculateTotalCost();
        double retailPrice = totalCost * (1 + salesTaxRate);
        return retailPrice;
    }
};

// Friend function to calculate total cost including taxes
double calculateTotalCostWithTax(const Pastry& pastry);

// Friend class to generate a report of total sales and profits
class PastryReport {
private:
```

```
    static double totalSales;
    static double totalProfit;

public:
    static void addSale(double salesAmount, double profitAmount) {
        totalSales += salesAmount;
        totalProfit += profitAmount;
    }

    static void generateReport() {
        cout << "Total Sales: " << totalSales << endl;
        cout << "Total Profit: " << totalProfit << endl;
    }

    friend class PastryShop; // Allow PastryShop to access private static members
};

double PastryReport::totalSales = 0.0;
double PastryReport::totalProfit = 0.0;


// Friend function definition to calculate total cost including taxes
double calculateTotalCostWithTax(const Pastry& pastry) {
    return pastry.calculateTotalCost() * (1 + pastry.calculateRetailPrice());
}

// PastryShop class to manage multiple pastries
class PastryShop {
private:
    Pastry* pastries[2]; // Assuming only 2 pastries for simplicity

public:
    PastryShop() : pastries{nullptr, nullptr} {}

    void addPastry(Pastry* pastry, int index) {
        if (index >= 0 && index < 2) {
            pastries[index] = pastry;
        }
    }

    void calculateTotalProfit() {
        for (int i = 0; i < 2; ++i) {
            if (pastries[i]) {
                double profit = pastries[i]->calculateProfit();
```

```
            double salesAmount = calculateTotalCostWithTax(*pastries[i]);
            PastryReport::addSale(salesAmount, profit);
          }
        }
    }

    void generateSalesReport() {
      PastryReport::generateReport();
    }
};
```