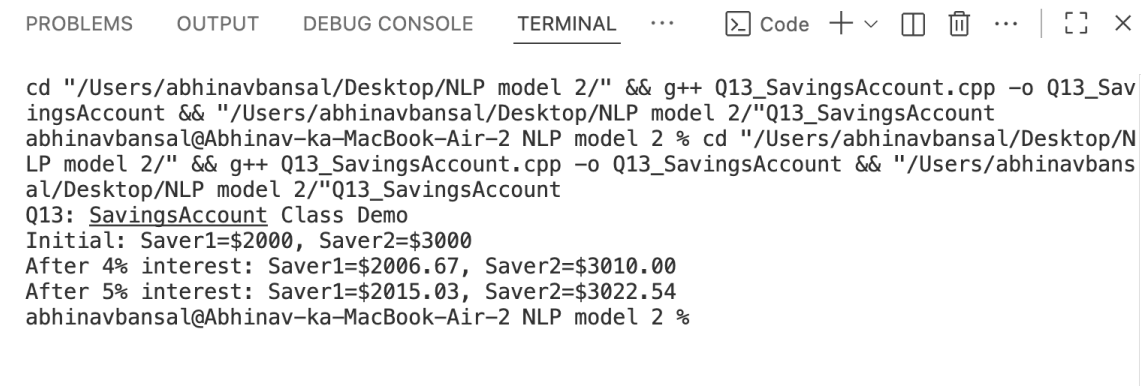
```
/*
Question 13: SavingsAccount Class
Create class SavingsAccount. Use a static variable annualInterestRate to store
the
annual interest rate for all account holders. Each object of the class contains
a private
instance variable savingsBalance indicating the amount the saver currently has
on
deposit. Provide method calculateMonthlyInterest() to calculate the monthly
interest by multiplying the savingsBalance by annualInterestRate divided by
12. This interest should be added to savingsBalance. Provide a static method
modifyInterestRate() that sets the annualInterestRate to a new value. Write a
program to test class SavingsAccount.
*/

#include <iostream>
#include <iomanip>
using namespace std;

class SavingsAccount {
private:
    double savingsBalance;
    static double annualInterestRate;

public:
    // Constructor
    SavingsAccount(double balance) : savingsBalance(balance) {}

    // Method to calculate monthly interest
    void calculateMonthlyInterest() {
        double monthlyInterest = (savingsBalance * annualInterestRate) / 12.0;
        savingsBalance += monthlyInterest;
    }

    // Static method to modify interest rate
    static void modifyInterestRate(double newRate) {
        annualInterestRate = newRate / 100.0; // Convert percentage to decimal
    }

    // Getter for balance
    double getBalance() const {
        return savingsBalance;
    }

    // Method to display account info
    void displayAccount() const {
        cout << fixed << setprecision(2);
        cout << "Current Balance: Rs" << savingsBalance << endl;
    }
};

// Initialize static member
double SavingsAccount::annualInterestRate = 0.0;

int main() {
```

```
    cout << "Q13: SavingsAccount Class Demo" << endl;

    // Create accounts and test functionality
    SavingsAccount saver1(2000.00);
    SavingsAccount saver2(3000.00);

    cout << "Initial: Saver1=$2000, Saver2=$3000" << endl;

    // Test with 4% interest
    SavingsAccount::modifyInterestRate(4.0);
    saver1.calculateMonthlyInterest();
    saver2.calculateMonthlyInterest();
    cout << "After 4% interest: Saver1=$" << fixed << setprecision(2) <<
saver1.getBalance()
         << ", Saver2=$" << saver2.getBalance() << endl;

    // Test with 5% interest
    SavingsAccount::modifyInterestRate(5.0);
    saver1.calculateMonthlyInterest();
    saver2.calculateMonthlyInterest();
    cout << "After 5% interest: Saver1=$" << saver1.getBalance()
         << ", Saver2=$" << saver2.getBalance() << endl;

    return 0;
}
```

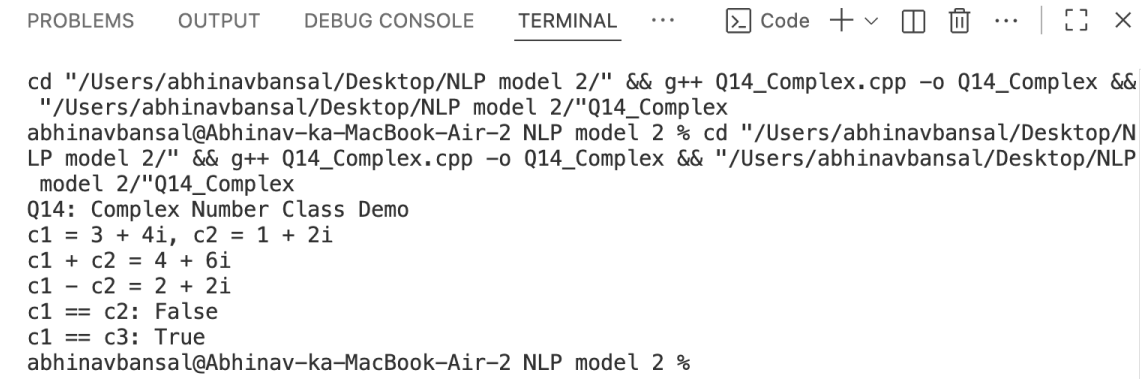
Terminal Output:



```
/*
Question 14: Complex Number Class
Create a class Complex having two int type variable named real & img denoting
real and imaginary part respectively of a complex number. Overload +, -, ==
operator to add, to subtract and to compare two complex numbers being denoted
by
the two complex type objects
*/

#include <iostream>
using namespace std;

class Complex {
private:
    int real;
    int img;

public:
    // Default constructor
    Complex() : real(0), img(0) {}

    // Parameterized constructor
    Complex(int r, int i) : real(r), img(i) {}

    // Copy constructor
    Complex(const Complex& other) : real(other.real), img(other.img) {}

    // Overload + operator
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, img + other.img);
    }

    // Overload - operator
    Complex operator-(const Complex& other) const {
        return Complex(real - other.real, img - other.img);
    }

    // Overload == operator
    bool operator==(const Complex& other) const {
        return (real == other.real && img == other.img);
    }

    // Method to display complex number
    void display() const {
        if (img >= 0) {
            cout << real << " + " << img << "i";
        } else {
            cout << real << " - " << (-img) << "i";
        }
    }

    // Getters
    int getReal() const { return real; }
    int getImg() const { return img; }
```

```
    // Setters
    void setReal(int r) { real = r; }
    void setImg(int i) { img = i; }
};

int main() {
    cout << "Q14: Complex Number Class Demo" << endl;

    // Create complex numbers
    Complex c1(3, 4);
    Complex c2(1, 2);
    Complex c3(3, 4);

    cout << "c1 = "; c1.display(); cout << ", c2 = "; c2.display(); cout <<
endl;

    // Test operations
    Complex sum = c1 + c2;
    Complex diff = c1 - c2;

    cout << "c1 + c2 = "; sum.display(); cout << endl;
    cout << "c1 - c2 = "; diff.display(); cout << endl;
    cout << "c1 == c2: " << (c1 == c2 ? "True" : "False") << endl;
    cout << "c1 == c3: " << (c1 == c3 ? "True" : "False") << endl;

    return 0;
}
```

Terminal Output:



```
/*
Question 15: Unary Operator Overloading
Using the concept of operator overloading. Implement a program to overload the
following:
a. Unary –
b. Unary ++ preincrement, postincrement
c. Unary -- predecrement, postdecrement
*/

#include <iostream>
using namespace std;

class Number {
private:
    int value;

public:
    // Constructor
    Number(int val = 0) : value(val) {}

    // Copy constructor
    Number(const Number& other) : value(other.value) {}

    // Overload unary minus operator (-)
    Number operator-() const {
        return Number(-value);
    }

    // Overload pre-increment operator (++obj)
    Number& operator++() {
        ++value;
        return *this;
    }

    // Overload post-increment operator (obj++)
    Number operator++(int) {
        Number temp(*this);
        ++value;
        return temp;
    }

    // Overload pre-decrement operator (--obj)
    Number& operator--() {
        --value;
        return *this;
    }

    // Overload post-decrement operator (obj--)
    Number operator--(int) {
        Number temp(*this);
        --value;
        return temp;
    }

    // Display method
```

```
    void display() const {
        cout << value;
    }

    // Getter
    int getValue() const {
        return value;
    }

    // Setter
    void setValue(int val) {
        value = val;
    }
};

int main() {
    cout << "Q15: Unary Operators Demo" << endl;

    Number n1(5);
    Number n2(10);

    cout << "n1 = " << n1.getValue() << ", n2 = " << n2.getValue() << endl;

    // Test unary operators
    Number n3 = -n1;
    cout << "-n1 = " << n3.getValue() << endl;

    cout << "++n1 = " << (++n1).getValue() << ", n1++ = " << (n1++).getValue()
<< ", n1 = " << n1.getValue() << endl;
    cout << "--n2 = " << (--n2).getValue() << ", n2-- = " << (n2--).getValue()
<< ", n2 = " << n2.getValue() << endl;

    return 0;
}
```

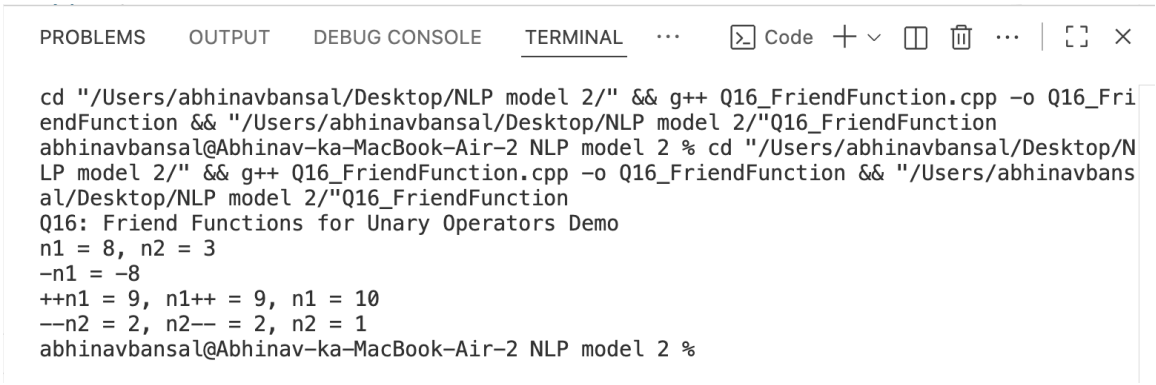
Terminal Output:


```
cd "/Users/abhinavbansal/Desktop/NLP model 2/" && g++ Q15_UnaryOperators.cpp -o Q15_
ryOperators && "/Users/abhinavbansal/Desktop/NLP model 2/"Q15_UnaryOperators
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 % cd "/Users/abhinavbansal/Deskt
LP model 2/" && g++ Q15_UnaryOperators.cpp -o Q15_UnaryOperators && "/Users/abhinav
al/Desktop/NLP model 2/"Q15_UnaryOperators
Q15: Unary Operators Demo
n1 = 5, n2 = 10
-n1 = -5
++n1 = 6, n1++ = 6, n1 = 7
--n2 = 9, n2-- = 9, n2 = 8
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 %
```

```
/*
Question 16: Unary Operator Overloading with Friend Functions
Using the concept of operator overloading. Implement a program to overload the
following:
With the help of friend function
a. Unary –
b. Unary ++ preincrement, postincrement
c. Unary -- predecrement, postdecrement
*/

#include <iostream>
using namespace std;

class Number {
private:
    int value;

public:
    // Constructor
    Number(int val = 0) : value(val) {}

    // Copy constructor
    Number(const Number& other) : value(other.value) {}

    // Friend function declarations for unary operator overloading
    friend Number operator-(const Number& num);
    friend Number& operator++(Number& num);        // Pre-increment
    friend Number operator++(Number& num, int);    // Post-increment
    friend Number& operator--(Number& num);        // Pre-decrement
    friend Number operator--(Number& num, int);    // Post-decrement

    // Display method
    void display() const {
        cout << value;
    }

    // Getter
    int getValue() const {
        return value;
    }

    // Setter
    void setValue(int val) {
        value = val;
    }
};

// Friend function definitions

// Overload unary minus operator (-) using friend function
Number operator-(const Number& num) {
    return Number(-num.value);
}

// Overload pre-increment operator (++obj) using friend function
```

```
Number& operator++(Number& num) {
    ++num.value;
    return num;
}

// Overload post-increment operator (obj++) using friend function
Number operator++(Number& num, int) {
    Number temp(num);
    ++num.value;
    return temp;
}

// Overload pre-decrement operator (--obj) using friend function
Number& operator--(Number& num) {
    --num.value;
    return num;
}

// Overload post-decrement operator (obj--) using friend function
Number operator--(Number& num, int) {
    Number temp(num);
    --num.value;
    return temp;
}

int main() {
    cout << "Q16: Friend Functions for Unary Operators Demo" << endl;

    Number n1(8);
    Number n2(3);

    cout << "n1 = " << n1.getValue() << ", n2 = " << n2.getValue() << endl;

    // Test friend function operators
    Number n3 = -n1;
    cout << "-n1 = " << n3.getValue() << endl;

    cout << "++n1 = " << (++n1).getValue() << ", n1++ = " << (n1++).getValue()
<< ", n1 = " << n1.getValue() << endl;
    cout << "--n2 = " << (--n2).getValue() << ", n2-- = " << (n2--).getValue()
<< ", n2 = " << n2.getValue() << endl;

    return 0;
}
```

Terminal Output:



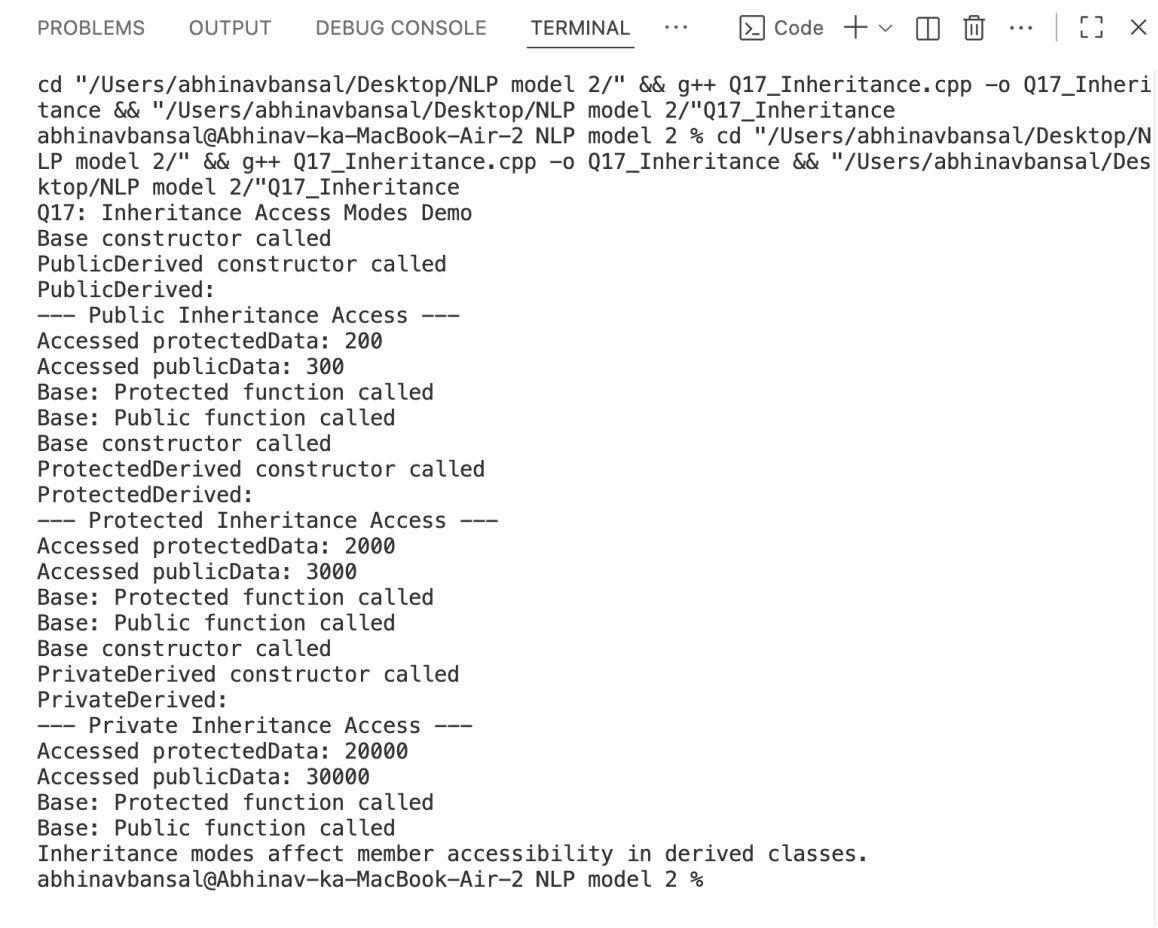
```
/*
Question 17: Inheritance Access Modes
Create a Base class that consists of private, protected and public data members
and
member functions. Try using different access modifiers for inheriting Base
class to
the Derived class and create a table that summarizes the above three modes
(when
derived in public, protected and private modes) and shows the access specifier
of
the members of base class in the Derived class
*/

#include <iostream>
using namespace std;

// Base class with all three access specifiers
class Base {
private:
    int privateData;
    void privateFunction() {
        cout << "Base: Private function called" << endl;
    }

protected:
    int protectedData;
    void protectedFunction() {
        cout << "Base: Protected function called" << endl;
    }

public:
    int publicData;

    // Constructor
    Base() : privateData(10), protectedData(20), publicData(30) {
        cout << "Base constructor called" << endl;
    }

    void publicFunction() {
        cout << "Base: Public function called" << endl;
    }

    // Public function to access private members (for demonstration)
    void accessPrivateMembers() {
        cout << "Base: Accessing private data = " << privateData << endl;
        privateFunction();
    }

    void displayBaseData() {
        cout << "Base Data - Private: " << privateData
             << ", Protected: " << protectedData
             << ", Public: " << publicData << endl;
    }
```

```
};

// Public Inheritance
class PublicDerived : public Base {
public:
    PublicDerived() {
        cout << "PublicDerived constructor called" << endl;
    }

    void accessMembers() {
        cout << "\n--- Public Inheritance Access ---" << endl;

        // privateData is NOT accessible - would cause compilation error
        // privateData = 100; // ERROR!

        // protectedData is accessible as protected
        protectedData = 200;
        cout << "Accessed protectedData: " << protectedData << endl;

        // publicData is accessible as public
        publicData = 300;
        cout << "Accessed publicData: " << publicData << endl;

        // Protected and public functions are accessible
        protectedFunction();
        publicFunction();
    }
};

// Protected Inheritance
class ProtectedDerived : protected Base {
public:
    ProtectedDerived() {
        cout << "ProtectedDerived constructor called" << endl;
    }

    void accessMembers() {
        cout << "\n--- Protected Inheritance Access ---" << endl;

        // privateData is NOT accessible
        // privateData = 100; // ERROR!

        // protectedData is accessible as protected
        protectedData = 2000;
        cout << "Accessed protectedData: " << protectedData << endl;

        // publicData is accessible as protected (not public anymore)
        publicData = 3000;
        cout << "Accessed publicData: " << publicData << endl;

        // Protected and public functions are accessible as protected
        protectedFunction();
        publicFunction();
    }
```

```
    // Public function to demonstrate access from outside
    void demonstrateAccess() {
        accessMembers();
    }
};

// Private Inheritance
class PrivateDerived : private Base {
public:
    PrivateDerived() {
        cout << "PrivateDerived constructor called" << endl;
    }

    void accessMembers() {
        cout << "\n--- Private Inheritance Access ---" << endl;

        // privateData is NOT accessible
        // privateData = 100; // ERROR!

        // protectedData is accessible as private
        protectedData = 20000;
        cout << "Accessed protectedData: " << protectedData << endl;

        // publicData is accessible as private
        publicData = 30000;
        cout << "Accessed publicData: " << publicData << endl;

        // Protected and public functions are accessible as private
        protectedFunction();
        publicFunction();
    }

    // Public function to demonstrate access from outside
    void demonstrateAccess() {
        accessMembers();
    }
};

void printAccessTable() {
    cout << "\n" << string(80, '=') << endl;
    cout << "ACCESS SPECIFIER TABLE FOR INHERITANCE" << endl;
    cout << string(80, '=') << endl;
    cout << "| Base Class    | Public        | Protected     | Private       |"
<< endl;
    cout << "| Access        | Inheritance   | Inheritance   | Inheritance   |"
<< endl;
    cout << "|---------------|---------------|---------------|---------------|"
<< endl;
    cout << "| Private       | Not Accessible| Not Accessible| Not Accessible|"
<< endl;
    cout << "| Protected     | Protected     | Protected     | Private       |"
<< endl;
    cout << "| Public        | Public        | Protected     | Private       |"
<< endl;
    cout << string(80, '=') << endl;
```

```
}

int main() {
    cout << "Q17: Inheritance Access Modes Demo" << endl;

    // Test public inheritance
    PublicDerived pd;
    cout << "PublicDerived: ";
    pd.accessMembers();

    // Test protected inheritance
    ProtectedDerived prd;
    cout << "ProtectedDerived: ";
    prd.demonstrateAccess();

    // Test private inheritance
    PrivateDerived pvd;
    cout << "PrivateDerived: ";
    pvd.demonstrateAccess();

    cout << "Inheritance modes affect member accessibility in derived classes."
<< endl;

    return 0;
}
```

Terminal Output:



```
/*
Question 18: Virtual Functions
Create a class hierarchy with virtual functions to demonstrate polymorphism.
Show the difference between virtual and non-virtual function calls.
*/

#include <iostream>
#include <vector>
#include <memory>
using namespace std;

// Base class with virtual and non-virtual functions
class Animal {
protected:
    string name;

public:
    Animal(const string& n) : name(n) {
        cout << "Animal constructor: " << name << endl;
    }

    // Virtual destructor (important for proper cleanup)
    virtual ~Animal() {
        cout << "Animal destructor: " << name << endl;
    }

    // Virtual function - will be overridden
    virtual void makeSound() const {
        cout << name << " makes a generic animal sound" << endl;
    }

    // Virtual function - will be overridden
    virtual void move() const {
        cout << name << " moves in some way" << endl;
    }

    // Non-virtual function - will NOT be overridden properly
    void eat() const {
        cout << name << " eats food" << endl;
    }

    // Pure virtual function - makes this an abstract class
    virtual void describe() const = 0;

    // Virtual function with default implementation
    virtual void sleep() const {
        cout << name << " sleeps peacefully" << endl;
    }

    string getName() const { return name; }
};

// Derived class - Dog
```

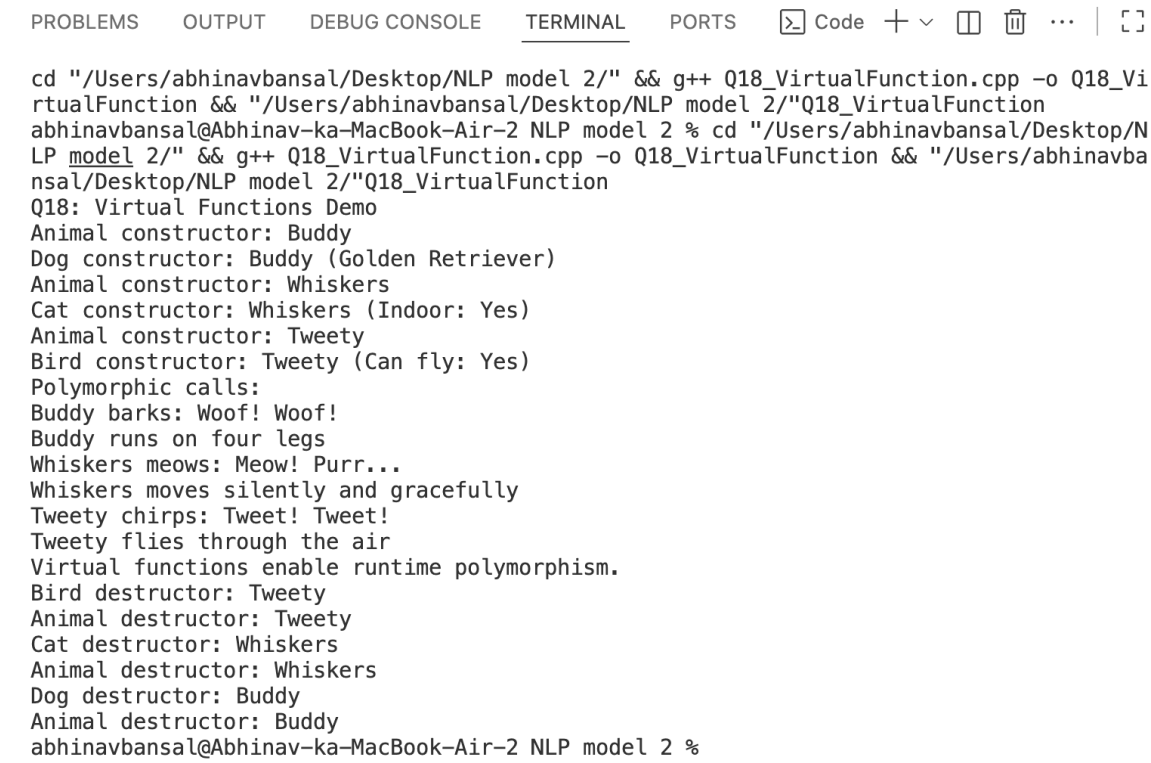
```
class Dog : public Animal {
private:
    string breed;

public:
    Dog(const string& n, const string& b) : Animal(n), breed(b) {
        cout << "Dog constructor: " << name << " (" << breed << ")" << endl;
    }

    ~Dog() {
        cout << "Dog destructor: " << name << endl;
    }

    // Override virtual function
    void makeSound() const override {
        cout << name << " barks: Woof! Woof!" << endl;
    }

    // Override virtual function
    void move() const override {
        cout << name << " runs on four legs" << endl;
    }

    // Override non-virtual function (this won't work as expected with
polymorphism)
    void eat() const {
        cout << name << " eats dog food and treats" << endl;
    }

    // Implement pure virtual function
    void describe() const override {
        cout << name << " is a " << breed << " dog" << endl;
    }

    // Dog-specific function
    void wagTail() const {
        cout << name << " wags tail happily" << endl;
    }
};

// Derived class - Cat
class Cat : public Animal {
private:
    bool isIndoor;

public:
    Cat(const string& n, bool indoor) : Animal(n), isIndoor(indoor) {
        cout << "Cat constructor: " << name << " (Indoor: " << (indoor ? "Yes"
: "No") << ")" << endl;
    }

    ~Cat() {
        cout << "Cat destructor: " << name << endl;
    }
```

```
    // Override virtual function
    void makeSound() const override {
        cout << name << " meows: Meow! Purr..." << endl;
    }

    // Override virtual function
    void move() const override {
        cout << name << " moves silently and gracefully" << endl;
    }

    // Override non-virtual function
    void eat() const {
        cout << name << " eats cat food and fish" << endl;
    }

    // Implement pure virtual function
    void describe() const override {
        cout << name << " is a " << (isIndoor ? "indoor" : "outdoor") << " cat"
<< endl;
    }

    // Override virtual function with different behavior
    void sleep() const override {
        cout << name << " sleeps 16 hours a day in a sunny spot" << endl;
    }

    // Cat-specific function
    void climb() const {
        cout << name << " climbs up high places" << endl;
    }
};

// Derived class - Bird
class Bird : public Animal {
private:
    bool canFly;

public:
    Bird(const string& n, bool fly) : Animal(n), canFly(fly) {
        cout << "Bird constructor: " << name << " (Can fly: " << (fly ? "Yes" :
"No") << ")" << endl;
    }

    ~Bird() {
        cout << "Bird destructor: " << name << endl;
    }

    // Override virtual function
    void makeSound() const override {
        cout << name << " chirps: Tweet! Tweet!" << endl;
    }

    // Override virtual function
    void move() const override {
        if (canFly) {
```

```
            cout << name << " flies through the air" << endl;
        } else {
            cout << name << " walks and hops on the ground" << endl;
        }
    }

    // Implement pure virtual function
    void describe() const override {
        cout << name << " is a " << (canFly ? "flying" : "flightless") << "
bird" << endl;
    }

    // Bird-specific function
    void buildNest() const {
        cout << name << " builds a nest" << endl;
    }
};

// Function to demonstrate polymorphism
void demonstratePolymorphism(const Animal& animal) {
    cout << "\n--- Polymorphic behavior ---" << endl;
    animal.describe();      // Virtual - calls derived class version
    animal.makeSound();     // Virtual - calls derived class version
    animal.move();          // Virtual - calls derived class version
    animal.eat();           // Non-virtual - calls base class version
    animal.sleep();         // Virtual - calls appropriate version
}

// Function to demonstrate polymorphism with pointers
void demonstratePolymorphismWithPointers(Animal* animal) {
    cout << "\n--- Polymorphic behavior with pointers ---" << endl;
    animal->describe();     // Virtual - calls derived class version
    animal->makeSound();    // Virtual - calls derived class version
    animal->move();         // Virtual - calls derived class version
    animal->eat();          // Non-virtual - calls base class version
    animal->sleep();        // Virtual - calls appropriate version
}

int main() {
    cout << "Q18: Virtual Functions Demo" << endl;

    // Create objects
    Dog dog("Buddy", "Golden Retriever");
    Cat cat("Whiskers", true);
    Bird bird("Tweety", true);

    // Polymorphism demonstration
    Animal* animals[] = {&dog, &cat, &bird};

    cout << "Polymorphic calls:" << endl;
    for (int i = 0; i < 3; i++) {
        animals[i]->makeSound();  // Virtual function call
        animals[i]->move();       // Virtual function call
    }
```

```
    cout << "Virtual functions enable runtime polymorphism." << endl;

    return 0;
}
```

Terminal Output:



```
/*
Question 19: Multiple Inheritance
Create a Person class with basic information. Create Student and Faculty
classes
that inherit from Person. Then create a TeachingAssistant class that inherits
from both Student and Faculty to demonstrate multiple inheritance.
*/

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

// Base class - Person
class Person {
protected:
    string name;
    int age;
    string address;
    string phoneNumber;

public:
    // Constructor
    Person(const string& n, int a, const string& addr, const string& phone)
        : name(n), age(a), address(addr), phoneNumber(phone) {
        cout << "Person constructor: " << name << endl;
    }

    // Virtual destructor
    virtual ~Person() {
        cout << "Person destructor: " << name << endl;
    }

    // Virtual functions
    virtual void displayInfo() const {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Address: " << address << endl;
        cout << "Phone: " << phoneNumber << endl;
    }

    virtual void introduce() const {
        cout << "Hi, I'm " << name << ", " << age << " years old." << endl;
    }

    // Getters
    string getName() const { return name; }
    int getAge() const { return age; }
    string getAddress() const { return address; }
    string getPhoneNumber() const { return phoneNumber; }

    // Setters
    void setAddress(const string& addr) { address = addr; }
    void setPhoneNumber(const string& phone) { phoneNumber = phone; }
};
```

```
// Derived class - Student
class Student : virtual public Person {
protected:
    string studentId;
    string major;
    double gpa;
    int semester;
    vector<string> courses;

public:
    // Constructor
    Student(const string& n, int a, const string& addr, const string& phone,
            const string& id, const string& maj, double g, int sem)
        : Person(n, a, addr, phone), studentId(id), major(maj), gpa(g),
semester(sem) {
        cout << "Student constructor: " << name << " (ID: " << studentId << ")"
<< endl;
    }

    // Virtual destructor
    virtual ~Student() {
        cout << "Student destructor: " << name << endl;
    }

    // Override virtual functions
    void displayInfo() const override {
        cout << "=== STUDENT INFORMATION ===" << endl;
        Person::displayInfo();
        cout << "Student ID: " << studentId << endl;
        cout << "Major: " << major << endl;
        cout << "GPA: " << gpa << endl;
        cout << "Semester: " << semester << endl;
        cout << "Enrolled Courses: ";
        if (courses.empty()) {
            cout << "None" << endl;
        } else {
            for (size_t i = 0; i < courses.size(); i++) {
                cout << courses[i];
                if (i < courses.size() - 1) cout << ", ";
            }
            cout << endl;
        }
    }

    void introduce() const override {
        cout << "Hi, I'm " << name << ", a " << major << " student with GPA "
<< gpa << endl;
    }

    // Student-specific methods
    void enrollCourse(const string& course) {
        courses.push_back(course);
        cout << name << " enrolled in " << course << endl;
    }
```

```
    void dropCourse(const string& course) {
        auto it = find(courses.begin(), courses.end(), course);
        if (it != courses.end()) {
            courses.erase(it);
            cout << name << " dropped " << course << endl;
        } else {
            cout << name << " is not enrolled in " << course << endl;
        }
    }

    void study() const {
        cout << name << " is studying for " << major << " courses" << endl;
    }

    void takeExam(const string& subject) const {
        cout << name << " is taking an exam in " << subject << endl;
    }

    // Getters
    string getStudentId() const { return studentId; }
    string getMajor() const { return major; }
    double getGPA() const { return gpa; }
    int getSemester() const { return semester; }

    // Setters
    void setGPA(double g) { gpa = g; }
    void setSemester(int sem) { semester = sem; }
};

// Derived class - Faculty
class Faculty : virtual public Person {
protected:
    string employeeId;
    string department;
    string position;
    double salary;
    vector<string> coursesTeaching;

public:
    // Constructor
    Faculty(const string& n, int a, const string& addr, const string& phone,
            const string& empId, const string& dept, const string& pos, double
sal)
        : Person(n, a, addr, phone), employeeId(empId), department(dept),
          position(pos), salary(sal) {
        cout << "Faculty constructor: " << name << " (ID: " << employeeId <<
")" << endl;
    }

    // Virtual destructor
    virtual ~Faculty() {
        cout << "Faculty destructor: " << name << endl;
    }
```

```
    // Override virtual functions
    void displayInfo() const override {
        cout << "=== FACULTY INFORMATION ===" << endl;
        Person::displayInfo();
        cout << "Employee ID: " << employeeId << endl;
        cout << "Department: " << department << endl;
        cout << "Position: " << position << endl;
        cout << "Salary: $" << salary << endl;
        cout << "Teaching Courses: ";
        if (coursesTeaching.empty()) {
            cout << "None" << endl;
        } else {
            for (size_t i = 0; i < coursesTeaching.size(); i++) {
                cout << coursesTeaching[i];
                if (i < coursesTeaching.size() - 1) cout << ", ";
            }
            cout << endl;
        }
    }

    void introduce() const override {
        cout << "Hi, I'm " << name << ", a " << position << " in " <<
department << " department" << endl;
    }

    // Faculty-specific methods
    void assignCourse(const string& course) {
        coursesTeaching.push_back(course);
        cout << name << " is now teaching " << course << endl;
    }

    void removeCourse(const string& course) {
        auto it = find(coursesTeaching.begin(), coursesTeaching.end(), course);
        if (it != coursesTeaching.end()) {
            coursesTeaching.erase(it);
            cout << name << " is no longer teaching " << course << endl;
        } else {
            cout << name << " is not teaching " << course << endl;
        }
    }

    void teach() const {
        cout << name << " is teaching classes in " << department << "
department" << endl;
    }

    void conductResearch() const {
        cout << name << " is conducting research in " << department << endl;
    }

    void gradeExams() const {
        cout << name << " is grading exams" << endl;
    }

    // Getters
```

```
    string getEmployeeId() const { return employeeId; }
    string getDepartment() const { return department; }
    string getPosition() const { return position; }
    double getSalary() const { return salary; }

    // Setters
    void setPosition(const string& pos) { position = pos; }
    void setSalary(double sal) { salary = sal; }
};

// Multiple Inheritance - TeachingAssistant inherits from both Student and
Faculty
class TeachingAssistant : public Student, public Faculty {
private:
    string supervisorName;
    int hoursPerWeek;
    double stipend;

public:
    // Constructor - must call both base class constructors
    TeachingAssistant(const string& n, int a, const string& addr, const string&
phone,
                      const string& studentId, const string& major, double gpa,
int sem,
                      const string& empId, const string& dept, const string&
pos,
                      const string& supervisor, int hours, double stip)
        : Person(n, a, addr, phone),  // Virtual base class constructor
          Student(n, a, addr, phone, studentId, major, gpa, sem),
          Faculty(n, a, addr, phone, empId, dept, pos, 0.0),  // Salary is 0
for TA
          supervisorName(supervisor), hoursPerWeek(hours), stipend(stip) {
        cout << "TeachingAssistant constructor: " << name << " (TA)" << endl;
    }

    // Destructor
    ~TeachingAssistant() {
        cout << "TeachingAssistant destructor: " << name << endl;
    }

    // Override displayInfo to show both student and faculty information
    void displayInfo() const override {
        cout << "=== TEACHING ASSISTANT INFORMATION ===" << endl;
        Person::displayInfo();
        cout << "--- Student Details ---" << endl;
        cout << "Student ID: " << studentId << endl;
        cout << "Major: " << major << endl;
        cout << "GPA: " << gpa << endl;
        cout << "Semester: " << semester << endl;
        cout << "--- Faculty Details ---" << endl;
        cout << "Employee ID: " << employeeId << endl;
        cout << "Department: " << department << endl;
        cout << "Position: " << position << endl;
        cout << "--- TA Specific Details ---" << endl;
        cout << "Supervisor: " << supervisorName << endl;
```

```
        cout << "Hours per Week: " << hoursPerWeek << endl;
        cout << "Stipend: $" << stipend << endl;
    }

    void introduce() const override {
        cout << "Hi, I'm " << name << ", a Teaching Assistant studying " <<
major
             << " and helping in " << department << " department" << endl;
    }

    // TA-specific methods
    void assistInLab() const {
        cout << name << " is assisting students in the lab" << endl;
    }

    void holdOfficeHours() const {
        cout << name << " is holding office hours for " << hoursPerWeek << "
hours per week" << endl;
    }

    void gradeAssignments() const {
        cout << name << " is grading assignments under supervision of " <<
supervisorName << endl;
    }

    // Methods that use both Student and Faculty functionality
    void balanceResponsibilities() const {
        cout << name << " is balancing student and teaching responsibilities:"
<< endl;
        cout << "  - Studying for " << major << " degree" << endl;
        cout << "  - Teaching/assisting in " << department << " department" <<
endl;
        cout << "  - Working " << hoursPerWeek << " hours per week" << endl;
    }

    // Getters
    string getSupervisor() const { return supervisorName; }
    int getHoursPerWeek() const { return hoursPerWeek; }
    double getStipend() const { return stipend; }

    // Setters
    void setSupervisor(const string& supervisor) { supervisorName = supervisor;
}
    void setHoursPerWeek(int hours) { hoursPerWeek = hours; }
    void setStipend(double stip) { stipend = stip; }
};

int main() {
    cout << "Q19: Multiple Inheritance Demo" << endl;

    // Create objects
    Student student("Alice", 20, "123 Main St", "555-0101", "S001", "Computer
Science", 3.8, 6);
    Faculty faculty("Dr. Smith", 45, "456 Faculty Ave", "555-0102", "F001",
"Computer Science", "Professor", 75000);
```

```
    TeachingAssistant ta("Bob", 25, "789 Campus Rd", "555-0103", "S002",
"Mathematics", 3.9, 8, "F002", "Mathematics", "Assistant Professor", "Dr.
Smith", 20, 1500);

    cout << "Student: "; student.displayInfo();
    cout << "Faculty: "; faculty.displayInfo();
    cout << "TA: "; ta.displayInfo();

    // Polymorphism test
    Person* people[] = {&student, &faculty, &ta};
    cout << "Polymorphic calls:" << endl;
    for (int i = 0; i < 3; i++) {
        people[i]->introduce();
    }

    return 0;
}
```

Terminal Output:

```
TA: === TEACHING ASSISTANT INFORMATION ===
Name: Bob
Age: 25
Address: 789 Campus Rd
Phone: 555-0103
--- Student Details ---
Student ID: S002
Major: Mathematics
GPA: 3.9
Semester: 8
--- Faculty Details ---
Employee ID: F002
Department: Mathematics
Position: Assistant Professor
--- TA Specific Details ---
Supervisor: Dr. Smith
Hours per Week: 20
Stipend: $1500
Polymorphic calls:
Hi, I'm Alice, a Computer Science student with GPA 3.8
Hi, I'm Dr. Smith, a Professor in Computer Science department
Hi, I'm Bob, a Teaching Assistant studying Mathematics and helping in Mathematics depa
tment
TeachingAssistant destructor: Bob
Faculty destructor: Bob
Student destructor: Bob
Person destructor: Bob
Faculty destructor: Dr. Smith
Person destructor: Dr. Smith
Student destructor: Alice
Person destructor: Alice
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 %
```

```
/*
Question 20: Diamond Problem
Demonstrate the diamond problem in multiple inheritance and show how to solve
it
using virtual inheritance. Create a hierarchy where a class inherits from two
classes that both inherit from a common base class.
*/

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

// ================================================================================
// PART 1: DEMONSTRATING THE DIAMOND PROBLEM (WITHOUT VIRTUAL INHERITANCE)
// ================================================================================

namespace WithoutVirtualInheritance {

    // Base class
    class Device {
    protected:
        string brand;
        double price;

    public:
        Device(const string& b, double p) : brand(b), price(p) {
            cout << "Device constructor: " << brand << " ($" << price << ")" <<
endl;
        }

        ~Device() {
            cout << "Device destructor: " << brand << endl;
        }

        void displayDevice() const {
            cout << "Device: " << brand << " - $" << price << endl;
        }

        string getBrand() const { return brand; }
        double getPrice() const { return price; }
    };

    // First derived class
    class Phone : public Device {
    protected:
        string phoneNumber;

    public:
        Phone(const string& b, double p, const string& num)
            : Device(b, p), phoneNumber(num) {
            cout << "Phone constructor: " << phoneNumber << endl;
        }

        ~Phone() {
            cout << "Phone destructor: " << phoneNumber << endl;
```

```
        }

        void makeCall(const string& number) const {
            cout << "Calling " << number << " from " << phoneNumber << endl;
        }

        void displayPhone() const {
            displayDevice();
            cout << "Phone Number: " << phoneNumber << endl;
        }
    };

    // Second derived class
    class Camera : public Device {
    protected:
        int megapixels;

    public:
        Camera(const string& b, double p, int mp)
            : Device(b, p), megapixels(mp) {
            cout << "Camera constructor: " << megapixels << "MP" << endl;
        }

        ~Camera() {
            cout << "Camera destructor: " << megapixels << "MP" << endl;
        }

        void takePicture() const {
            cout << "Taking picture with " << megapixels << "MP camera" <<
endl;
        }

        void displayCamera() const {
            displayDevice();
            cout << "Megapixels: " << megapixels << "MP" << endl;
        }
    };

    // Diamond problem: SmartPhone inherits from both Phone and Camera
    // This creates TWO instances of Device in SmartPhone
    class SmartPhone : public Phone, public Camera {
    private:
        string operatingSystem;

    public:
        SmartPhone(const string& b, double p, const string& num, int mp, const
string& os)
            : Phone(b, p, num), Camera(b, p, mp), operatingSystem(os) {
            cout << "SmartPhone constructor: " << os << endl;
        }

        ~SmartPhone() {
            cout << "SmartPhone destructor: " << operatingSystem << endl;
        }
```

```
        void displaySmartPhone() const {
            cout << "=== SMARTPHONE INFO (Diamond Problem) ===" << endl;

            // PROBLEM: Ambiguous calls - which Device should we use?
            // displayDevice();  // ERROR: Ambiguous!

            // We must specify which path to take:
            cout << "Via Phone path: ";
            Phone::displayDevice();
            cout << "Via Camera path: ";
            Camera::displayDevice();

            cout << "Phone Number: " << phoneNumber << endl;
            cout << "Camera: " << megapixels << "MP" << endl;
            cout << "OS: " << operatingSystem << endl;
        }

        void demonstrateProblem() const {
            cout << "\n--- Diamond Problem Demonstration ---" << endl;

            // These work fine
            makeCall("555-1234");
            takePicture();

            // But accessing Device members is ambiguous
            cout << "Brand via Phone: " << Phone::getBrand() << endl;
            cout << "Brand via Camera: " << Camera::getBrand() << endl;
            cout << "Price via Phone: $" << Phone::getPrice() << endl;
            cout << "Price via Camera: $" << Camera::getPrice() << endl;

            cout << "Problem: Two separate Device objects exist!" << endl;
        }
    };
}

// ============================================================================
// PART 2: SOLVING THE DIAMOND PROBLEM (WITH VIRTUAL INHERITANCE)
// ============================================================================

namespace WithVirtualInheritance {

    // Base class (same as before)
    class Device {
    protected:
        string brand;
        double price;

    public:
        Device(const string& b, double p) : brand(b), price(p) {
            cout << "Device constructor (virtual): " << brand << " ($" << price
<< ")" << endl;
        }

        virtual ~Device() {
            cout << "Device destructor (virtual): " << brand << endl;
```

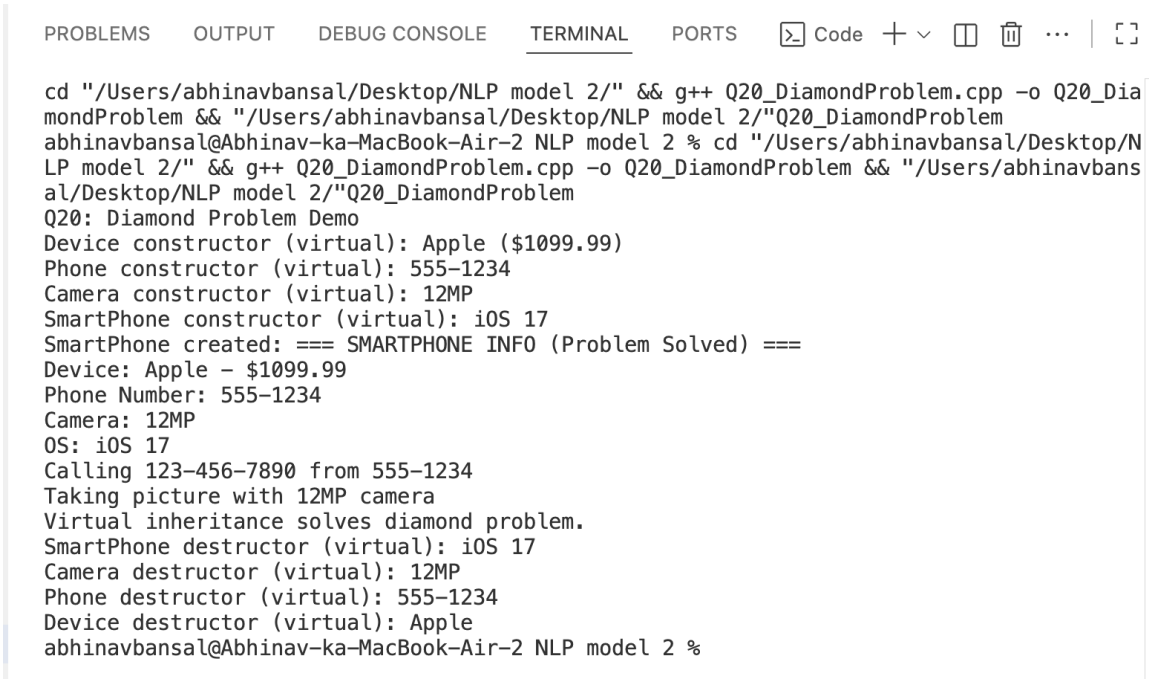
```
        }

        virtual void displayDevice() const {
            cout << "Device: " << brand << " - $" << price << endl;
        }

        string getBrand() const { return brand; }
        double getPrice() const { return price; }

        void setBrand(const string& b) { brand = b; }
        void setPrice(double p) { price = p; }
    };

    // First derived class with VIRTUAL inheritance
    class Phone : virtual public Device {
    protected:
        string phoneNumber;

    public:
        Phone(const string& b, double p, const string& num)
            : Device(b, p), phoneNumber(num) {
            cout << "Phone constructor (virtual): " << phoneNumber << endl;
        }

        virtual ~Phone() {
            cout << "Phone destructor (virtual): " << phoneNumber << endl;
        }

        void makeCall(const string& number) const {
            cout << "Calling " << number << " from " << phoneNumber << endl;
        }

        void displayPhone() const {
            displayDevice();
            cout << "Phone Number: " << phoneNumber << endl;
        }

        string getPhoneNumber() const { return phoneNumber; }
    };

    // Second derived class with VIRTUAL inheritance
    class Camera : virtual public Device {
    protected:
        int megapixels;

    public:
        Camera(const string& b, double p, int mp)
            : Device(b, p), megapixels(mp) {
            cout << "Camera constructor (virtual): " << megapixels << "MP" <<
endl;
        }

        virtual ~Camera() {
            cout << "Camera destructor (virtual): " << megapixels << "MP" <<
endl;
```

```
        }

        void takePicture() const {
            cout << "Taking picture with " << megapixels << "MP camera" <<
endl;
        }

        void displayCamera() const {
            displayDevice();
            cout << "Megapixels: " << megapixels << "MP" << endl;
        }

        int getMegapixels() const { return megapixels; }
    };

    // Solution: SmartPhone with virtual inheritance
    // Now there's only ONE instance of Device
    class SmartPhone : public Phone, public Camera {
    private:
        string operatingSystem;

    public:
        // IMPORTANT: Must explicitly call Device constructor
        SmartPhone(const string& b, double p, const string& num, int mp, const
string& os)
            : Device(b, p),  // Explicit call to virtual base constructor
              Phone(b, p, num), Camera(b, p, mp), operatingSystem(os) {
            cout << "SmartPhone constructor (virtual): " << os << endl;
        }

        virtual ~SmartPhone() {
            cout << "SmartPhone destructor (virtual): " << operatingSystem <<
endl;
        }

        void displaySmartPhone() const {
            cout << "=== SMARTPHONE INFO (Problem Solved) ===" << endl;

            // NO AMBIGUITY: Only one Device object exists
            displayDevice();  // Works perfectly!

            cout << "Phone Number: " << phoneNumber << endl;
            cout << "Camera: " << megapixels << "MP" << endl;
            cout << "OS: " << operatingSystem << endl;
        }

        void demonstrateSolution() const {
            cout << "\n--- Diamond Problem Solution ---" << endl;

            // All work without ambiguity
            makeCall("555-5678");
            takePicture();

            // No ambiguity in accessing Device members
            cout << "Brand: " << getBrand() << endl;
```

```
            cout << "Price: $" << getPrice() << endl;

            cout << "Solution: Only one Device object exists!" << endl;
        }

        void takeSelfieDuringCall() const {
            cout << "Multi-tasking: Taking selfie while on a call!" << endl;
            cout << "Using phone features and camera features simultaneously"
<< endl;
        }

        string getOS() const { return operatingSystem; }
    };
}

// ================================================================================
// DEMONSTRATION AND COMPARISON
// ================================================================================

void demonstrateDiamondProblem() {
    cout << "\n" << string(70, '=') << endl;
    cout << "DIAMOND PROBLEM DEMONSTRATION" << endl;
    cout << string(70, '=') << endl;

    cout << "\n1. WITHOUT VIRTUAL INHERITANCE (Problem):" << endl;
    cout << string(50, '-') << endl;

    {
        WithoutVirtualInheritance::SmartPhone phone1("Apple", 999.99,
"555-0001", 12, "iOS");
        phone1.displaySmartPhone();
        phone1.demonstrateProblem();

        cout << "\nMemory layout: SmartPhone contains TWO Device objects" <<
endl;
        cout << "Size of SmartPhone: " << sizeof(phone1) << " bytes" << endl;
    }

    cout << "\n" << string(50, '-') << endl;
    cout << "\n2. WITH VIRTUAL INHERITANCE (Solution):" << endl;
    cout << string(50, '-') << endl;

    {
        WithVirtualInheritance::SmartPhone phone2("Samsung", 899.99,
"555-0002", 48, "Android");
        phone2.displaySmartPhone();
        phone2.demonstrateSolution();
        phone2.takeSelfieDuringCall();

        cout << "\nMemory layout: SmartPhone contains ONE Device object" <<
endl;
        cout << "Size of SmartPhone: " << sizeof(phone2) << " bytes" << endl;
    }
}
```

```
void explainDiamondProblem() {
    cout << "\n" << string(70, '=') << endl;
    cout << "DIAMOND PROBLEM EXPLANATION" << endl;
    cout << string(70, '=') << endl;

    cout << "\nDiamond Problem Hierarchy:" << endl;
    cout << "       Device" << endl;
    cout << "      /      \\" << endl;
    cout << "   Phone    Camera" << endl;
    cout << "      \\      /" << endl;
    cout << "    SmartPhone" << endl;

    cout << "\nWithout Virtual Inheritance:" << endl;
    cout << "- SmartPhone contains TWO Device objects" << endl;
    cout << "- Ambiguous access to Device members" << endl;
    cout << "- Larger memory footprint" << endl;
    cout << "- Must use scope resolution (Phone::getBrand())" << endl;

    cout << "\nWith Virtual Inheritance:" << endl;
    cout << "- SmartPhone contains ONE Device object" << endl;
    cout << "- No ambiguity in member access" << endl;
    cout << "- Smaller memory footprint" << endl;
    cout << "- Direct access to Device members" << endl;
    cout << "- Must explicitly call virtual base constructor" << endl;

    cout << "\nKey Points:" << endl;
    cout << "1. Use 'virtual' keyword in inheritance: class Phone : virtual
public Device" << endl;
    cout << "2. Virtual inheritance creates shared base class instance" <<
endl;
    cout << "3. Most derived class must call virtual base constructor" << endl;
    cout << "4. Slight performance overhead due to virtual table lookups" <<
endl;
    cout << "5. Essential for complex inheritance hierarchies" << endl;
}

int main() {
    cout << "Q20: Diamond Problem Demo" << endl;

    // Create SmartPhone object with virtual inheritance
    WithVirtualInheritance::SmartPhone myPhone("Apple", 1099.99, "555-1234",
12, "iOS 17");

    cout << "SmartPhone created: ";
    myPhone.displaySmartPhone();

    // Test functionality
    myPhone.makeCall("123-456-7890");
    myPhone.takePicture();

    cout << "Virtual inheritance solves diamond problem." << endl;

    return 0;
}
```

Terminal Output:



```
/*
Question 21: Shape Hierarchy with Virtual Functions
Create an abstract Shape class with pure virtual functions for calculating area
and perimeter. Derive classes like Circle, Rectangle, Triangle, and demonstrate
polymorphism by storing different shapes in an array and calling their methods.
*/

#include <iostream>
#include <vector>
#include <memory>
#include <cmath>
#include <iomanip>
using namespace std;

// Abstract base class - Shape
class Shape {
protected:
    string name;
    string color;

public:
    // Constructor
    Shape(const string& n, const string& c) : name(n), color(c) {
        cout << "Shape constructor: " << name << " (" << color << ")" << endl;
    }

    // Virtual destructor
    virtual ~Shape() {
        cout << "Shape destructor: " << name << endl;
    }

    // Pure virtual functions - make this an abstract class
    virtual double calculateArea() const = 0;
    virtual double calculatePerimeter() const = 0;

    // Virtual function with default implementation
    virtual void displayInfo() const {
        cout << "Shape: " << name << " (Color: " << color << ")" << endl;
        cout << "Area: " << fixed << setprecision(2) << calculateArea() << " sq
units" << endl;
        cout << "Perimeter: " << fixed << setprecision(2) <<
calculatePerimeter() << " units" << endl;
    }

    // Virtual function for drawing (can be overridden)
    virtual void draw() const {
        cout << "Drawing a " << color << " " << name << endl;
    }

    // Non-virtual functions
    string getName() const { return name; }
    string getColor() const { return color; }
    void setColor(const string& c) { color = c; }

    // Virtual function to get shape-specific properties
```

```
    virtual void displayProperties() const = 0;
};

// Derived class - Circle
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r, const string& c = "Red")
        : Shape("Circle", c), radius(r) {
        cout << "Circle constructor: radius = " << radius << endl;
    }

    ~Circle() {
        cout << "Circle destructor: radius = " << radius << endl;
    }

    // Override pure virtual functions
    double calculateArea() const override {
        return M_PI * radius * radius;
    }

    double calculatePerimeter() const override {
        return 2 * M_PI * radius;
    }

    // Override virtual functions
    void displayInfo() const override {
        cout << "=== CIRCLE INFO ===" << endl;
        Shape::displayInfo();
        cout << "Radius: " << radius << " units" << endl;
        cout << "Diameter: " << 2 * radius << " units" << endl;
    }

    void draw() const override {
        cout << "Drawing a " << getColor() << " circle with radius " << radius
<< endl;
        cout << "    ***" << endl;
        cout << "  *     *" << endl;
        cout << " *       *" << endl;
        cout << "  *     *" << endl;
        cout << "    ***" << endl;
    }

    void displayProperties() const override {
        cout << "Circle Properties: Radius = " << radius << endl;
    }

    // Circle-specific methods
    double getDiameter() const { return 2 * radius; }
    double getRadius() const { return radius; }
    void setRadius(double r) { radius = r; }
};
```

```
// Derived class - Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double l, double w, const string& c = "Blue")
        : Shape("Rectangle", c), length(l), width(w) {
        cout << "Rectangle constructor: " << length << " x " << width << endl;
    }

    ~Rectangle() {
        cout << "Rectangle destructor: " << length << " x " << width << endl;
    }

    // Override pure virtual functions
    double calculateArea() const override {
        return length * width;
    }

    double calculatePerimeter() const override {
        return 2 * (length + width);
    }

    // Override virtual functions
    void displayInfo() const override {
        cout << "=== RECTANGLE INFO ===" << endl;
        Shape::displayInfo();
        cout << "Length: " << length << " units" << endl;
        cout << "Width: " << width << " units" << endl;
        cout << "Diagonal: " << sqrt(length*length + width*width) << " units"
<< endl;
    }

    void draw() const override {
        cout << "Drawing a " << getColor() << " rectangle " << length << " x "
<< width << endl;
        cout << "*********" << endl;
        cout << "*       *" << endl;
        cout << "*       *" << endl;
        cout << "*********" << endl;
    }

    void displayProperties() const override {
        cout << "Rectangle Properties: Length = " << length << ", Width = " <<
width << endl;
    }

    // Rectangle-specific methods
    double getLength() const { return length; }
    double getWidth() const { return width; }
    double getDiagonal() const { return sqrt(length*length + width*width); }
    bool isSquare() const { return length == width; }
    void setDimensions(double l, double w) { length = l; width = w; }
```

```
};

// Derived class - Triangle
class Triangle : public Shape {
private:
    double side1, side2, side3;

public:
    Triangle(double s1, double s2, double s3, const string& c = "Green")
        : Shape("Triangle", c), side1(s1), side2(s2), side3(s3) {
        cout << "Triangle constructor: " << side1 << ", " << side2 << ", " <<
side3 << endl;

        // Validate triangle inequality
        if (!isValidTriangle()) {
            cout << "Warning: Invalid triangle dimensions!" << endl;
        }
    }

    ~Triangle() {
        cout << "Triangle destructor: " << side1 << ", " << side2 << ", " <<
side3 << endl;
    }

    // Override pure virtual functions
    double calculateArea() const override {
        // Using Heron's formula
        double s = calculatePerimeter() / 2.0;  // semi-perimeter
        return sqrt(s * (s - side1) * (s - side2) * (s - side3));
    }

    double calculatePerimeter() const override {
        return side1 + side2 + side3;
    }

    // Override virtual functions
    void displayInfo() const override {
        cout << "=== TRIANGLE INFO ===" << endl;
        Shape::displayInfo();
        cout << "Side 1: " << side1 << " units" << endl;
        cout << "Side 2: " << side2 << " units" << endl;
        cout << "Side 3: " << side3 << " units" << endl;
        cout << "Type: " << getTriangleType() << endl;
    }

    void draw() const override {
        cout << "Drawing a " << getColor() << " triangle with sides "
             << side1 << ", " << side2 << ", " << side3 << endl;
        cout << "    /\\" << endl;
        cout << "   /  \\" << endl;
        cout << "  /    \\" << endl;
        cout << " /______\\" << endl;
    }

    void displayProperties() const override {
```

```
        cout << "Triangle Properties: Sides = " << side1 << ", " << side2 << ",
" << side3 << endl;
        cout << "Type: " << getTriangleType() << endl;
    }

    // Triangle-specific methods
    bool isValidTriangle() const {
        return (side1 + side2 > side3) &&
               (side1 + side3 > side2) &&
               (side2 + side3 > side1);
    }

    string getTriangleType() const {
        if (side1 == side2 && side2 == side3) {
            return "Equilateral";
        } else if (side1 == side2 || side2 == side3 || side1 == side3) {
            return "Isosceles";
        } else {
            return "Scalene";
        }
    }

    bool isRightTriangle() const {
        double a = side1, b = side2, c = side3;
        // Sort sides
        if (a > b) swap(a, b);
        if (b > c) swap(b, c);
        if (a > b) swap(a, b);

        // Check Pythagorean theorem (with small tolerance for floating point)
        return abs(a*a + b*b - c*c) < 0.001;
    }

    double getSide1() const { return side1; }
    double getSide2() const { return side2; }
    double getSide3() const { return side3; }
};

// Derived class - Square (special case of Rectangle)
class Square : public Rectangle {
public:
    Square(double side, const string& c = "Yellow")
        : Rectangle(side, side, c) {
        // Change the name to Square
        name = "Square";
        cout << "Square constructor: side = " << side << endl;
    }

    ~Square() {
        cout << "Square destructor: side = " << getLength() << endl;
    }

    void displayInfo() const override {
        cout << "=== SQUARE INFO ===" << endl;
        cout << "Shape: " << name << " (Color: " << color << ")" << endl;
```

```
        cout << "Area: " << fixed << setprecision(2) << calculateArea() << " sq
units" << endl;
        cout << "Perimeter: " << fixed << setprecision(2) <<
calculatePerimeter() << " units" << endl;
        cout << "Side: " << getLength() << " units" << endl;
        cout << "Diagonal: " << getDiagonal() << " units" << endl;
    }

    void draw() const override {
        cout << "Drawing a " << getColor() << " square with side " <<
getLength() << endl;
        cout << "*****" << endl;
        cout << "*   *" << endl;
        cout << "*   *" << endl;
        cout << "*****" << endl;
    }

    void displayProperties() const override {
        cout << "Square Properties: Side = " << getLength() << endl;
    }

    double getSide() const { return getLength(); }
    void setSide(double side) { setDimensions(side, side); }
};

// Function to demonstrate polymorphism
void processShape(const Shape& shape) {
    cout << "\n--- Processing Shape Polymorphically ---" << endl;
    shape.displayInfo();
    shape.draw();
    shape.displayProperties();
}

// Function to compare shapes
void compareShapes(const Shape& shape1, const Shape& shape2) {
    cout << "\n--- Comparing Shapes ---" << endl;
    cout << shape1.getName() << " vs " << shape2.getName() << endl;

    double area1 = shape1.calculateArea();
    double area2 = shape2.calculateArea();

    cout << "Areas: " << area1 << " vs " << area2 << endl;

    if (area1 > area2) {
        cout << shape1.getName() << " has larger area" << endl;
    } else if (area2 > area1) {
        cout << shape2.getName() << " has larger area" << endl;
    } else {
        cout << "Both shapes have equal area" << endl;
    }
}

int main() {
    cout << "Q21: Shape Hierarchy Demo" << endl;
```

```
    // Create shape objects
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);
    Triangle triangle(3.0, 4.0, 5.0);
    Square square(4.0);

    // Test calculations
     cout << "Circle (r=5): Area=" << circle.calculateArea() << ", Perimeter="
<< circle.calculatePerimeter() << endl;
     cout << "Rectangle (4x6): Area=" << rectangle.calculateArea() << ",
Perimeter=" << rectangle.calculatePerimeter() << endl;
     cout << "Triangle (3,4,5): Area=" << triangle.calculateArea() << ",
Perimeter=" << triangle.calculatePerimeter() << endl;
     cout << "Square (side=4): Area=" << square.calculateArea() << ",
Perimeter=" << square.calculatePerimeter() << endl;

    // Polymorphism test
    Shape* shapes[] = {&circle, &rectangle, &triangle, &square};
    cout << "Polymorphic calls:" << endl;
    for (int i = 0; i < 4; i++) {
        shapes[i]->displayInfo();
    }

    return 0;
}
```

Terminal Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ··· Code

```
cd "/Users/abhinavbansal/Desktop/NLP model 2/" && g++ Q21_Shapes.cpp -o Q21_Shapes && "
/Users/abhinavbansal/Desktop/NLP model 2/"Q21_Shapes
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 % cd "/Users/abhinavbansal/Desktop/N
LP model 2/" && g++ Q21_Shapes.cpp -o Q21_Shapes && "/Users/abhinavbansal/Desktop/NLP m
odel 2/"Q21_Shapes
Q21: Shape Hierarchy Demo
Shape constructor: Circle (Red)
Circle constructor: radius = 5
Shape constructor: Rectangle (Blue)
Rectangle constructor: 4 x 6
Shape constructor: Triangle (Green)
Triangle constructor: 3, 4, 5
Shape constructor: Rectangle (Yellow)
Rectangle constructor: 4 x 4
Square constructor: side = 4
Circle (r=5): Area=78.5398, Perimeter=31.4159
Rectangle (4x6): Area=24, Perimeter=20
Triangle (3,4,5): Area=6, Perimeter=12
Square (side=4): Area=16, Perimeter=16
Polymorphic calls:
=== CIRCLE INFO ===
Shape: Circle (Color: Red)
Area: 78.54 sq units
Perimeter: 31.42 units
Radius: 5.00 units
Diameter: 10.00 units
=== RECTANGLE INFO ===
Shape: Rectangle (Color: Blue)
Area: 24.00 sq units
Perimeter: 20.00 units
Length: 4.00 units
Width: 6.00 units
Diagonal: 7.21 units
=== TRIANGLE INFO ===
Shape: Triangle (Color: Green)
Area: 6.00 sq units
Perimeter: 12.00 units
Side 1: 3.00 units
Side 2: 4.00 units
Side 3: 5.00 units
Type: Scalene
=== SQUARE INFO ===
Shape: Square (Color: Yellow)
Area: 16.00 sq units
Perimeter: 16.00 units
Side: 4.00 units
Diagonal: 5.66 units
Square destructor: side = 4.00
Rectangle destructor: 4.00 x 4.00
Shape destructor: Square
Triangle destructor: 3.00, 4.00, 5.00
Shape destructor: Triangle
Rectangle destructor: 4.00 x 6.00
Shape destructor: Rectangle
Circle destructor: radius = 5.00
Shape destructor: Circle
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 %
```

```
// Q22: Abstract Base Class for Volume Calculation
// This program demonstrates abstract base classes with pure virtual functions
// for calculating volumes and surface areas of 3D shapes

#include <iostream>
#include <vector>
#include <memory>
#include <cmath>
#include <iomanip>
#include <algorithm>
using namespace std;

// Abstract base class for volume calculation
class VolumeCalculator {
protected:
    string shapeName;
    string material;
    double density;  // kg/m³

public:
    // Constructor
    VolumeCalculator(const string& name, const string& mat = "Unknown", double
d = 1.0)
        : shapeName(name), material(mat), density(d) {
        cout << "VolumeCalculator constructor: " << name << endl;
    }

    // Virtual destructor
    virtual ~VolumeCalculator() {
        cout << "VolumeCalculator destructor: " << shapeName << endl;
    }

    // Pure virtual functions - must be implemented by derived classes
    virtual double calculateVolume() const = 0;
    virtual double calculateSurfaceArea() const = 0;

    // Virtual function with default implementation
    virtual void displayInfo() const {
        cout << "Shape: " << shapeName << endl;
        cout << "Material: " << material << endl;
        cout << "Density: " << density << " kg/m³" << endl;
        cout << "Volume: " << calculateVolume() << " m³" << endl;
        cout << "Surface Area: " << calculateSurfaceArea() << " m²" << endl;
        cout << "Mass: " << calculateMass() << " kg" << endl;
    }

    // Non-virtual function
    double calculateMass() const {
        return calculateVolume() * density;
    }

    // Getters
    string getShapeName() const { return shapeName; }
    string getMaterial() const { return material; }
    double getDensity() const { return density; }
```

```
    // Pure virtual function for displaying dimensions
    virtual void displayDimensions() const = 0;

    // Virtual function that can be overridden for specific cost calculations
    virtual double calculateCost(double costPerKg) const {
        return calculateMass() * costPerKg;
    }
};

// Derived class - Sphere
class Sphere : public VolumeCalculator {
private:
    double radius;

public:
    Sphere(double r, const string& mat = "Steel", double d = 7850.0)
        : VolumeCalculator("Sphere", mat, d), radius(r) {
        cout << "Sphere constructor: radius = " << r << endl;
    }

    ~Sphere() {
        cout << "Sphere destructor" << endl;
    }

    // Implement pure virtual functions
    double calculateVolume() const override {
        return (4.0/3.0) * M_PI * radius * radius * radius;
    }

    double calculateSurfaceArea() const override {
        return 4.0 * M_PI * radius * radius;
    }

    // Override virtual function
    void displayInfo() const override {
        cout << "=== SPHERE INFO ===" << endl;
        VolumeCalculator::displayInfo();
        displayDimensions();
    }

    void displayDimensions() const override {
        cout << "Radius: " << radius << " m" << endl;
    }

    // Getters
    double getRadius() const { return radius; }

    // Override cost calculation for spheres (might have different pricing)
    double calculateCost(double costPerKg) const override {
        // Spheres might have 10% premium due to manufacturing complexity
        return calculateMass() * costPerKg * 1.1;
    }
};
```

```
// Derived class - Cube
class Cube : public VolumeCalculator {
private:
    double side;

public:
    Cube(double s, const string& mat = "Aluminum", double d = 2700.0)
        : VolumeCalculator("Cube", mat, d), side(s) {
        cout << "Cube constructor: side = " << s << endl;
    }

    ~Cube() {
        cout << "Cube destructor" << endl;
    }

    // Implement pure virtual functions
    double calculateVolume() const override {
        return side * side * side;
    }

    double calculateSurfaceArea() const override {
        return 6.0 * side * side;
    }

    // Override virtual function
    void displayInfo() const override {
        cout << "=== CUBE INFO ===" << endl;
        VolumeCalculator::displayInfo();
        displayDimensions();
    }

    void displayDimensions() const override {
        cout << "Side length: " << side << " m" << endl;
    }

    // Getters
    double getSide() const { return side; }
};

// Derived class - Cylinder
class Cylinder : public VolumeCalculator {
private:
    double radius;
    double height;

public:
    Cylinder(double r, double h, const string& mat = "Copper", double d =
8960.0)
        : VolumeCalculator("Cylinder", mat, d), radius(r), height(h) {
        cout << "Cylinder constructor: radius = " << r << ", height = " << h <<
endl;
    }

    ~Cylinder() {
        cout << "Cylinder destructor" << endl;
```

```
    }

    // Implement pure virtual functions
    double calculateVolume() const override {
        return M_PI * radius * radius * height;
    }

    double calculateSurfaceArea() const override {
        return 2.0 * M_PI * radius * (radius + height);
    }

    // Override virtual function
    void displayInfo() const override {
        cout << "=== CYLINDER INFO ===" << endl;
        VolumeCalculator::displayInfo();
        displayDimensions();
    }

    void displayDimensions() const override {
        cout << "Radius: " << radius << " m, Height: " << height << " m" <<
endl;
    }

    // Getters
    double getRadius() const { return radius; }
    double getHeight() const { return height; }
};

// Derived class - Cone
class Cone : public VolumeCalculator {
private:
    double radius;
    double height;

public:
    Cone(double r, double h, const string& mat = "Plastic", double d = 950.0)
        : VolumeCalculator("Cone", mat, d), radius(r), height(h) {
        cout << "Cone constructor: radius = " << r << ", height = " << h <<
endl;
    }

    ~Cone() {
        cout << "Cone destructor" << endl;
    }

    // Implement pure virtual functions
    double calculateVolume() const override {
        return (1.0/3.0) * M_PI * radius * radius * height;
    }

    double calculateSurfaceArea() const override {
        double slantHeight = sqrt(radius * radius + height * height);
        return M_PI * radius * (radius + slantHeight);
    }
```

```
    // Override virtual function
    void displayInfo() const override {
        cout << "=== CONE INFO ===" << endl;
        VolumeCalculator::displayInfo();
        displayDimensions();
    }

    void displayDimensions() const override {
        cout << "Radius: " << radius << " m, Height: " << height << " m" <<
endl;
    }

    // Getters
    double getRadius() const { return radius; }
    double getHeight() const { return height; }

    // Override cost calculation for cones (might have different pricing)
    double calculateCost(double costPerKg) const override {
        // Cones might have 5% discount due to less material usage
        return calculateMass() * costPerKg * 0.95;
    }
};

// Derived class - Rectangular Prism
class RectangularPrism : public VolumeCalculator {
private:
    double length, width, height;

public:
    RectangularPrism(double l, double w, double h, const string& mat = "Wood",
double d = 600.0)
        : VolumeCalculator("Rectangular Prism", mat, d), length(l), width(w),
height(h) {
        cout << "RectangularPrism constructor: " << l << "x" << w << "x" << h
<< endl;
    }

    ~RectangularPrism() {
        cout << "RectangularPrism destructor" << endl;
    }

    // Implement pure virtual functions
    double calculateVolume() const override {
        return length * width * height;
    }

    double calculateSurfaceArea() const override {
        return 2.0 * (length * width + width * height + height * length);
    }

    // Override virtual function
    void displayInfo() const override {
        cout << "=== RECTANGULAR PRISM INFO ===" << endl;
        VolumeCalculator::displayInfo();
        displayDimensions();
```

```
    }

    void displayDimensions() const override {
        cout << "Dimensions: " << length << " x " << width << " x " << height
<< " m" << endl;
    }

    // Getters
    double getLength() const { return length; }
    double getWidth() const { return width; }
    double getHeight() const { return height; }
};

int main() {
    cout << "Q22: Volume Calculator Demo" << endl;

    // Create 3D objects
    Sphere sphere(5.0);
    Cube cube(4.0);
    Cylinder cylinder(3.0, 8.0);
    Cone cone(4.0, 6.0);
    RectangularPrism prism(3.0, 4.0, 5.0);

    // Test calculations
    cout << "Sphere (r=5): Volume=" << sphere.calculateVolume() << ", Surface
Area=" << sphere.calculateSurfaceArea() << endl;
    cout << "Cube (side=4): Volume=" << cube.calculateVolume() << ", Surface
Area=" << cube.calculateSurfaceArea() << endl;
    cout << "Cylinder (r=3,h=8): Volume=" << cylinder.calculateVolume() << ",
Surface Area=" << cylinder.calculateSurfaceArea() << endl;
    cout << "Cone (r=4,h=6): Volume=" << cone.calculateVolume() << ", Surface
Area=" << cone.calculateSurfaceArea() << endl;
    cout << "Prism (3x4x5): Volume=" << prism.calculateVolume() << ", Surface
Area=" << prism.calculateSurfaceArea() << endl;

    // Polymorphism test
    VolumeCalculator* objects[] = {&sphere, &cube, &cylinder, &cone, &prism};
    cout << "Polymorphic calls:" << endl;
    for (int i = 0; i < 5; i++) {
        objects[i]->displayInfo();
    }

    return 0;
}
```

Terminal Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ··· Code

```
cd "/Users/abhinavbansal/Desktop/NLP model 2/" && g++ Q22_VolumeCalculator.cpp -o Q22
olumeCalculator && "/Users/abhinavbansal/Desktop/NLP model 2/"Q22_VolumeCalculator
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 % cd "/Users/abhinavbansal/Desktop
LP model 2/" && g++ Q22_VolumeCalculator.cpp -o Q22_VolumeCalculator && "/Users/abhin
bansal/Desktop/NLP model 2/"Q22_VolumeCalculator
Q22: Volume Calculator Demo
VolumeCalculator constructor: Sphere
Sphere constructor: radius = 5
VolumeCalculator constructor: Cube
Cube constructor: side = 4
VolumeCalculator constructor: Cylinder
Cylinder constructor: radius = 3, height = 8
VolumeCalculator constructor: Cone
Cone constructor: radius = 4, height = 6
VolumeCalculator constructor: Rectangular Prism
RectangularPrism constructor: 3x4x5
Sphere (r=5): Volume=523.599, Surface Area=314.159
Cube (side=4): Volume=64, Surface Area=96
Cylinder (r=3,h=8): Volume=226.195, Surface Area=207.345
Cone (r=4,h=6): Volume=100.531, Surface Area=140.883
Prism (3x4x5): Volume=60, Surface Area=94
Polymorphic calls:
=== SPHERE INFO ===
Shape: Sphere
Material: Steel
Density: 7850 kg/m³
Volume: 523.599 m³
Surface Area: 314.159 m²
Mass: 4.11025e+06 kg
Radius: 5 m
=== CUBE INFO ===
Shape: Cube
Material: Aluminum
Density: 2700 kg/m³
Volume: 64 m³
Surface Area: 96 m²
Mass: 172800 kg
Side length: 4 m
```

```
=== CYLINDER INFO ===
Shape: Cylinder
Material: Copper
Density: 8960 kg/m³
Volume: 226.195 m³
Surface Area: 207.345 m²
Mass: 2.0267e+06 kg
Radius: 3 m, Height: 8 m
=== CONE INFO ===
Shape: Cone
Material: Plastic
Density: 950 kg/m³
Volume: 100.531 m³
Surface Area: 140.883 m²
Mass: 95504.4 kg
Radius: 4 m, Height: 6 m
=== RECTANGULAR PRISM INFO ===
Shape: Rectangular Prism
Material: Wood
Density: 600 kg/m³
Volume: 60 m³
Surface Area: 94 m²
Mass: 36000 kg
Dimensions: 3 x 4 x 5 m
RectangularPrism destructor
VolumeCalculator destructor: Rectangular Prism
Cone destructor
VolumeCalculator destructor: Cone
Cylinder destructor
VolumeCalculator destructor: Cylinder
Cube destructor
VolumeCalculator destructor: Cube
Sphere destructor
VolumeCalculator destructor: Sphere
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 %
```

```
// Q23: Exception Handling
// This program demonstrates exception handling in C++ with custom exceptions,
// try-catch blocks, and exception specifications

#include <iostream>
#include <string>
#include <vector>
#include <stdexcept>
#include <limits>
#include <sstream>
#include <typeinfo>
#include <memory>
#include <fstream>
using namespace std;

// Custom Exception Classes
class InvalidInputException : public exception {
private:
    string message;
    int errorCode;

public:
    InvalidInputException(const string& msg, int code = 1001)
        : message("Invalid Input: " + msg), errorCode(code) {}

    const char* what() const noexcept override {
        return message.c_str();
    }

    int getErrorCode() const { return errorCode; }
};

class NegativeNumberException : public InvalidInputException {
public:
    NegativeNumberException(double value)
        : InvalidInputException("Negative number not allowed: " +
to_string(value), 1002) {}
};

class DivisionByZeroException : public exception {
private:
    string message;

public:
    DivisionByZeroException() : message("Division by zero attempted") {}

    const char* what() const noexcept override {
        return message.c_str();
    }
};

class OutOfRangeException : public exception {
private:
    string message;
    double value;
```

```
    double minRange;
    double maxRange;

public:
    OutOfRangeException(double val, double min, double max)
        : value(val), minRange(min), maxRange(max) {
        message = "Value " + to_string(val) + " is out of range [" +
                  to_string(min) + ", " + to_string(max) + "]";
    }

    const char* what() const noexcept override {
        return message.c_str();
    }

    double getValue() const { return value; }
    double getMinRange() const { return minRange; }
    double getMaxRange() const { return maxRange; }
};

class FileOperationException : public exception {
private:
    string message;
    string filename;

public:
    FileOperationException(const string& file, const string& operation)
        : filename(file) {
        message = "File operation failed: " + operation + " on file '" + file +
"'";
    }

    const char* what() const noexcept override {
        return message.c_str();
    }

    string getFilename() const { return filename; }
};

class MemoryAllocationException : public bad_alloc {
private:
    string message;
    size_t requestedSize;

public:
    MemoryAllocationException(size_t size) : requestedSize(size) {
        message = "Memory allocation failed for " + to_string(size) + " bytes";
    }

    const char* what() const noexcept override {
        return message.c_str();
    }

    size_t getRequestedSize() const { return requestedSize; }
};
```

```
// Enum for different input types
enum class InputType {
    INTEGER,
    DOUBLE,
    STRING,
    DIVISION,
    SQUARE_ROOT,
    ARRAY_ACCESS,
    FILE_OPERATION,
    MEMORY_ALLOCATION,
    FACTORIAL,
    QUIT
};

// Function to display menu
void displayMenu() {
    cout << "\n=== Exception Handling Demo Menu ===" << endl;
    cout << "1. Integer Input (throws for negative)" << endl;
    cout << "2. Double Input (throws for out of range)" << endl;
    cout << "3. String Input (throws for empty)" << endl;
    cout << "4. Division Operation (throws for zero divisor)" << endl;
    cout << "5. Square Root (throws for negative)" << endl;
    cout << "6. Array Access (throws for out of bounds)" << endl;
    cout << "7. File Operation (throws for file errors)" << endl;
    cout << "8. Memory Allocation (throws for large allocation)" << endl;
    cout << "9. Factorial Calculation (throws for negative/large)" << endl;
    cout << "0. Quit" << endl;
    cout << "Enter your choice: ";
}

// Main process_input function with exception handling
void process_input() {
    int choice;

    try {
        displayMenu();
        cin >> choice;

        // Check for input failure
        if (cin.fail()) {
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            throw InvalidInputException("Invalid menu choice - not a number");
        }

        InputType inputType = static_cast<InputType>(choice);

        switch (inputType) {
            case InputType::INTEGER: {
                cout << "Enter an integer (positive only): ";
                int value;
                cin >> value;

                if (cin.fail()) {
                    cin.clear();
```

```
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                throw InvalidInputException("Invalid integer input");
            }

            if (value < 0) {
                throw NegativeNumberException(value);
            }

            cout << "Success! You entered: " << value << endl;
            cout << "Square: " << value * value << endl;
            break;
        }

        case InputType::DOUBLE: {
            cout << "Enter a double value (0.0 to 100.0): ";
            double value;
            cin >> value;

            if (cin.fail()) {
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                throw InvalidInputException("Invalid double input");
            }

            if (value < 0.0 || value > 100.0) {
                throw OutOfRangeException(value, 0.0, 100.0);
            }

            cout << "Success! You entered: " << value << endl;
            cout << "Percentage: " << value << "%" << endl;
            break;
        }

        case InputType::STRING: {
            cout << "Enter a non-empty string: ";
            cin.ignore(); // Clear the newline from previous input
            string value;
            getline(cin, value);

            if (value.empty()) {
                throw InvalidInputException("Empty string not allowed");
            }

            if (value.length() > 50) {
                throw InvalidInputException("String too long (max 50
characters)");
            }

            cout << "Success! You entered: '" << value << "'" << endl;
            cout << "Length: " << value.length() << " characters" << endl;
            break;
        }

        case InputType::DIVISION: {
            cout << "Enter dividend: ";
```

```
                double dividend;
                cin >> dividend;

                cout << "Enter divisor: ";
                double divisor;
                cin >> divisor;

                if (cin.fail()) {
                    cin.clear();
                    cin.ignore(numeric_limits<streamsize>::max(), '\n');
                    throw InvalidInputException("Invalid numeric input for
division");
                }

                if (divisor == 0.0) {
                    throw DivisionByZeroException();
                }

                double result = dividend / divisor;
                cout << "Success! " << dividend << " / " << divisor << " = " <<
result << endl;
                break;
            }

            case InputType::SQUARE_ROOT: {
                cout << "Enter a number for square root: ";
                double value;
                cin >> value;

                if (cin.fail()) {
                    cin.clear();
                    cin.ignore(numeric_limits<streamsize>::max(), '\n');
                    throw InvalidInputException("Invalid numeric input for
square root");
                }

                if (value < 0) {
                    throw NegativeNumberException(value);
                }

                double result = sqrt(value);
                cout << "Success! √" << value << " = " << result << endl;
                break;
            }

            case InputType::ARRAY_ACCESS: {
                vector<int> arr = {10, 20, 30, 40, 50};
                cout << "Array contents: ";
                for (size_t i = 0; i < arr.size(); i++) {
                    cout << "[" << i << "]=" << arr[i] << " ";
                }
                cout << endl;

                cout << "Enter array index (0-" << (arr.size()-1) << "): ";
                int index;
```

```
                cin >> index;

                if (cin.fail()) {
                    cin.clear();
                    cin.ignore(numeric_limits<streamsize>::max(), '\n');
                    throw InvalidInputException("Invalid index input");
                }

                if (index < 0 || index >= static_cast<int>(arr.size())) {
                    throw out_of_range("Array index " + to_string(index) +
                                       " is out of bounds [0, " +
to_string(arr.size()-1) + "]");
                }

                cout << "Success! arr[" << index << "] = " << arr[index] <<
endl;
                break;
            }

            case InputType::FILE_OPERATION: {
                cout << "Enter filename to read: ";
                string filename;
                cin >> filename;

                ifstream file(filename);
                if (!file.is_open()) {
                    throw FileOperationException(filename, "open for reading");
                }

                string line;
                int lineCount = 0;
                cout << "File contents:" << endl;
                while (getline(file, line) && lineCount < 5) { // Read max 5
lines
                    cout << "Line " << (lineCount + 1) << ": " << line << endl;
                    lineCount++;
                }

                if (lineCount == 0) {
                    cout << "File is empty" << endl;
                } else {
                    cout << "Successfully read " << lineCount << " lines" <<
endl;
                }

                file.close();
                break;
            }

            case InputType::MEMORY_ALLOCATION: {
                cout << "Enter number of integers to allocate: ";
                size_t count;
                cin >> count;

                if (cin.fail()) {
```

```
                    cin.clear();
                    cin.ignore(numeric_limits<streamsize>::max(), '\n');
                    throw InvalidInputException("Invalid count for memory
allocation");
                }

                // Simulate memory allocation failure for very large requests
                if (count > 1000000) {
                    throw MemoryAllocationException(count * sizeof(int));
                }

                try {
                    unique_ptr<int[]> ptr(new int[count]);

                    // Initialize array
                    for (size_t i = 0; i < count; i++) {
                        ptr[i] = static_cast<int>(i * i);
                    }

                    cout << "Success! Allocated array of " << count << "
integers" << endl;
                    cout << "First few values: ";
                    for (size_t i = 0; i < min(count, size_t(5)); i++) {
                        cout << ptr[i] << " ";
                    }
                    cout << endl;

                } catch (const bad_alloc& e) {
                    throw MemoryAllocationException(count * sizeof(int));
                }
                break;
            }

            case InputType::FACTORIAL: {
                cout << "Enter a number for factorial (0-20): ";
                int n;
                cin >> n;

                if (cin.fail()) {
                    cin.clear();
                    cin.ignore(numeric_limits<streamsize>::max(), '\n');
                    throw InvalidInputException("Invalid input for factorial");
                }

                if (n < 0) {
                    throw NegativeNumberException(n);
                }

                if (n > 20) {
                    throw OutOfRangeException(n, 0, 20);
                }

                long long factorial = 1;
                for (int i = 1; i <= n; i++) {
                    factorial *= i;
```

```
                }

                cout << "Success! " << n << "! = " << factorial << endl;
                break;
            }

            case InputType::QUIT:
                cout << "Exiting program..." << endl;
                return;

            default:
                throw InvalidInputException("Invalid menu choice: " +
to_string(choice));
        }

    } catch (const NegativeNumberException& e) {
        cout << "❌ Negative Number Error: " << e.what() << endl;
        cout << "   Error Code: " << e.getErrorCode() << endl;
        cout << "   Solution: Please enter a positive number" << endl;

    } catch (const DivisionByZeroException& e) {
        cout << "❌ Division Error: " << e.what() << endl;
        cout << "   Solution: Please enter a non-zero divisor" << endl;

    } catch (const OutOfRangeException& e) {
        cout << "❌ Range Error: " << e.what() << endl;
        cout << "   Valid range: [" << e.getMinRange() << ", " <<
e.getMaxRange() << "]" << endl;
        cout << "   Your input: " << e.getValue() << endl;

    } catch (const FileOperationException& e) {
        cout << "❌ File Error: " << e.what() << endl;
        cout << "   File: " << e.getFilename() << endl;
        cout << "   Solution: Check if file exists and is readable" << endl;

    } catch (const MemoryAllocationException& e) {
        cout << "❌ Memory Error: " << e.what() << endl;
        cout << "   Requested: " << e.getRequestedSize() << " bytes" << endl;
        cout << "   Solution: Try a smaller allocation size" << endl;

    } catch (const InvalidInputException& e) {
        cout << "❌ Input Error: " << e.what() << endl;
        cout << "   Error Code: " << e.getErrorCode() << endl;
        cout << "   Solution: Please check your input format" << endl;

    } catch (const out_of_range& e) {
        cout << "❌ Standard Range Error: " << e.what() << endl;
        cout << "   Solution: Please use a valid index" << endl;

    } catch (const bad_alloc& e) {
        cout << "❌ Standard Memory Error: " << e.what() << endl;
        cout << "   Solution: Insufficient memory available" << endl;

    } catch (const runtime_error& e) {
        cout << "❌ Runtime Error: " << e.what() << endl;
```

```
        cout << "   Type: " << typeid(e).name() << endl;

    } catch (const logic_error& e) {
        cout << "❌ Logic Error: " << e.what() << endl;
        cout << "   Type: " << typeid(e).name() << endl;

    } catch (const exception& e) {
        cout << "❌ Standard Exception: " << e.what() << endl;
        cout << "   Type: " << typeid(e).name() << endl;

    } catch (...) {
        cout << "❌ Unknown Exception: An unexpected error occurred" << endl;
        cout << "   This is a catch-all handler for any unhandled exception" <<
endl;
    }
}

// Function to demonstrate basic exception handling
void demonstrateBasicExceptions() {
    cout << "Testing basic exceptions:" << endl;

    // Test division by zero
    try {
        double a = 10, b = 0;
        if (b == 0) throw DivisionByZeroException();
        cout << "Result: " << a / b << endl;
    } catch (const DivisionByZeroException& e) {
        cout << "Caught: " << e.what() << endl;
    }

    // Test negative number
    try {
        double value = -5.0;
        if (value < 0) throw NegativeNumberException(value);
        cout << "Square root: " << sqrt(value) << endl;
    } catch (const NegativeNumberException& e) {
        cout << "Caught: " << e.what() << endl;
    }

    // Test out of range
    try {
        double value = 150;
        if (value < 0 || value > 100) throw OutOfRangeException(value, 0, 100);
        cout << "Valid percentage: " << value << "%" << endl;
    } catch (const OutOfRangeException& e) {
        cout << "Caught: " << e.what() << endl;
    }
}

// Function to demonstrate nested exceptions
void demonstrateNestedExceptions() {
    cout << "Testing nested exceptions:" << endl;

    try {
        try {
```

```
            throw InvalidInputException("Inner exception");
        } catch (const InvalidInputException& e) {
            cout << "Inner catch: " << e.what() << endl;
            throw runtime_error("Outer exception");
        }
    } catch (const runtime_error& e) {
        cout << "Outer catch: " << e.what() << endl;
    }
}

// Function with exception specification
void demonstrateExceptionSpecifications() noexcept {
    cout << "Testing noexcept function (no exceptions thrown)" << endl;
    try {
        // This function promises not to throw
        int result = 42;
        cout << "Safe operation result: " << result << endl;
    } catch (...) {
        cout << "This should never execute" << endl;
    }
}

// RAII demonstration
class ResourceManager {
private:
    string resourceName;
    bool acquired;

public:
    ResourceManager(const string& name) : resourceName(name), acquired(false) {
        cout << "Acquiring resource: " << resourceName << endl;
        acquired = true;
    }

    ~ResourceManager() {
        if (acquired) {
            cout << "Releasing resource: " << resourceName << endl;
        }
    }

    void useResource() {
        if (!acquired) throw runtime_error("Resource not acquired");
        cout << "Using resource: " << resourceName << endl;
    }
};

void demonstrateRAII() {
    cout << "Testing RAII:" << endl;
    try {
        ResourceManager rm("Database Connection");
        rm.useResource();
        throw runtime_error("Simulated error");
    } catch (const exception& e) {
        cout << "Exception caught: " << e.what() << endl;
        cout << "Resource automatically cleaned up" << endl;
```

```
    }
}

int main() {
    cout << "Q23: Exception Handling Demo" << endl;

    // Demonstrate different exception handling concepts
    demonstrateBasicExceptions();
    demonstrateNestedExceptions();
    demonstrateExceptionSpecifications();
    demonstrateRAII();

    // Test standard library exceptions
    try {
        vector<int> vec(5);
        cout << "Accessing valid index: " << vec.at(2) << endl;
        cout << "Accessing invalid index: " << vec.at(10) << endl;
    } catch (const out_of_range& e) {
        cout << "Standard exception caught: " << e.what() << endl;
    }

    cout << "Exception handling demonstration completed" << endl;
    return 0;
}
```

Terminal Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ···  Code  + ⌄  ···

cd "/Users/abhinavbansal/Desktop/NLP model 2/" && g++ Q23_ExceptionHandling.cpp -o Q23_
ExceptionHandling && "/Users/abhinavbansal/Desktop/NLP model 2/"Q23_ExceptionHandling
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 % cd "/Users/abhinavbansal/Desktop/N
LP model 2/" && g++ Q23_ExceptionHandling.cpp -o Q23_ExceptionHandling && "/Users/abhin
avbansal/Desktop/NLP model 2/"Q23_ExceptionHandling
Q23: Exception Handling Demo
Testing basic exceptions:
Caught: Division by zero attempted
Caught: Invalid Input: Negative number not allowed: -5.000000
Caught: Value 150.000000 is out of range [0.000000, 100.000000]
Testing nested exceptions:
Inner catch: Invalid Input: Inner exception
Outer catch: Outer exception
Testing noexcept function (no exceptions thrown)
Safe operation result: 42
Testing RAII:
Acquiring resource: Database Connection
Using resource: Database Connection
Releasing resource: Database Connection
Exception caught: Simulated error
Resource automatically cleaned up
Accessing valid index: 0
Accessing invalid index: Standard exception caught: vector
Exception handling demonstration completed
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 %
```

```cpp
// Q24: Templates (Generic Programming)
// This program demonstrates function templates, class templates,
// template specialization, and template metaprogramming

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <memory>
#include <typeinfo>
using namespace std;

// Function Templates
template<typename T>
T findMax(const T& a, const T& b) {
    return (a > b) ? a : b;
}

template<typename T>
T findMin(const T& a, const T& b) {
    return (a < b) ? a : b;
}

template<typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

template<typename T>
void bubbleSort(vector<T>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swapValues(arr[j], arr[j + 1]);
            }
        }
    }
}

// Variadic Templates
template<typename T>
T sum(T value) {
    return value;
}

template<typename T, typename... Args>
T sum(T first, Args... args) {
    return first + sum(args...);
}

// Class Templates
template<typename T>
```

```
class Stack {
private:
    vector<T> elements;
    size_t maxSize;

public:
    explicit Stack(size_t max = 100) : maxSize(max) {}

    void push(const T& element) {
        if (elements.size() >= maxSize) {
            throw overflow_error("Stack overflow");
        }
        elements.push_back(element);
    }

    T pop() {
        if (empty()) {
            throw underflow_error("Stack underflow");
        }
        T top = elements.back();
        elements.pop_back();
        return top;
    }

    const T& top() const {
        if (empty()) {
            throw underflow_error("Stack is empty");
        }
        return elements.back();
    }

    bool empty() const {
        return elements.empty();
    }

    size_t size() const {
        return elements.size();
    }

    void display() const {
        cout << "Stack contents: ";
        for (const auto& elem : elements) {
            cout << elem << " ";
        }
        cout << endl;
    }
};

template<typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;

public:
```

```
    Pair() : first(T1{}), second(T2{}) {}
    Pair(const T1& f, const T2& s) : first(f), second(s) {}

    T1 getFirst() const { return first; }
    T2 getSecond() const { return second; }
    void setFirst(const T1& f) { first = f; }
    void setSecond(const T2& s) { second = s; }

    void display() const {
        cout << "(" << first << ", " << second << ")";
    }
};

// Template Specialization
template<>
class Stack<bool> {
private:
    vector<bool> elements;
    size_t maxSize;

public:
    explicit Stack(size_t max = 100) : maxSize(max) {}

    void push(bool element) {
        if (elements.size() >= maxSize) {
            throw overflow_error("Stack overflow");
        }
        elements.push_back(element);
    }

    bool pop() {
        if (empty()) {
            throw underflow_error("Stack underflow");
        }
        bool top = elements.back();
        elements.pop_back();
        return top;
    }

    bool empty() const {
        return elements.empty();
    }

    void display() const {
        cout << "Bool Stack: ";
        for (bool elem : elements) {
            cout << (elem ? "true" : "false") << " ";
        }
        cout << endl;
    }

    int countTrue() const {
        return count(elements.begin(), elements.end(), true);
    }
};
```

```
// Function Template Specialization
template<>
string findMax<string>(const string& a, const string& b) {
    return (a.length() > b.length()) ? a : b;
}

// Template Metaprogramming
template<int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0> {
    static const int value = 1;
};

void demonstrateFunctionTemplates() {
    cout << "Function Templates:" << endl;

    // Basic function templates
    cout << "Max(10, 20): " << findMax(10, 20) << endl;
    cout << "Max(3.14, 2.71): " << findMax(3.14, 2.71) << endl;
    cout << "Min('a', 'z'): " << findMin('a', 'z') << endl;

    // Variadic templates
    cout << "Sum(1, 2, 3, 4, 5): " << sum(1, 2, 3, 4, 5) << endl;

    // Sorting
    vector<int> numbers = {64, 34, 25, 12, 22, 11, 90};
    cout << "Before sorting: ";
    for (int n : numbers) cout << n << " ";
    cout << endl;

    bubbleSort(numbers);
    cout << "After sorting: ";
    for (int n : numbers) cout << n << " ";
    cout << endl;
}

void demonstrateClassTemplates() {
    cout << "Class Templates:" << endl;

    // Stack template
    Stack<int> intStack;
    intStack.push(10);
    intStack.push(20);
    intStack.push(30);
    intStack.display();
    cout << "Popped: " << intStack.pop() << endl;
    intStack.display();

    // Pair template
    Pair<int, string> p1(42, "Hello");
```

```
    Pair<double, char> p2(3.14, 'A');
    cout << "Pair 1: ";
    p1.display();
    cout << endl;
    cout << "Pair 2: ";
    p2.display();
    cout << endl;
}

void demonstrateTemplateSpecialization() {
    cout << "Template Specialization:" << endl;

    // Specialized Stack<bool>
    Stack<bool> boolStack;
    boolStack.push(true);
    boolStack.push(false);
    boolStack.push(true);
    boolStack.display();
    cout << "True count: " << boolStack.countTrue() << endl;

    // Specialized string comparison
    string s1 = "short";
    string s2 = "longer string";
    cout << "Max string by length: " << findMax(s1, s2) << endl;
}

void demonstrateTemplateMetaprogramming() {
    cout << "Template Metaprogramming:" << endl;

    // Compile-time factorial calculation
    cout << "Factorial<5>: " << Factorial<5>::value << endl;
    cout << "Factorial<0>: " << Factorial<0>::value << endl;
    cout << "Factorial<7>: " << Factorial<7>::value << endl;
}

int main() {
    cout << "Q24: Templates Demo" << endl;

    try {
        demonstrateFunctionTemplates();
        demonstrateClassTemplates();
        demonstrateTemplateSpecialization();
        demonstrateTemplateMetaprogramming();

    } catch (const exception& e) {
        cout << "Exception: " << e.what() << endl;
    }

    cout << "Template demonstration completed" << endl;
    return 0;
}
```

Terminal Output:

```cpp
// Q25: STL (Standard Template Library)
// This program demonstrates STL containers, iterators, and algorithms
// with focus on vector operations including erase() function

#include <iostream>
#include <vector>
#include <list>
#include <map>
#include <set>
#include <unordered_map>
#include <queue>
#include <stack>
#include <deque>
#include <algorithm>
#include <iterator>
#include <numeric>
#include <functional>
#include <string>
using namespace std;

// Utility function to print containers
template<typename Container>
void printContainer(const Container& container, const string& name) {
    cout << name << ": ";
    for (const auto& element : container) {
        cout << element << " ";
    }
    cout << endl;
}

void demonstrateVectorOperations() {
    cout << "Vector Operations:" << endl;

    vector<int> vec = {10, 20, 30, 40, 50};
    printContainer(vec, "Original");

    // Insert operations
    vec.insert(vec.begin() + 2, 25);
    printContainer(vec, "After insert");

    // Erase single element
    vec.erase(vec.begin() + 1);
    printContainer(vec, "After erase single");

    // Erase range
    vec.erase(vec.begin() + 1, vec.begin() + 3);
    printContainer(vec, "After erase range");

    // Remove-erase idiom
    vec = {1, 2, 3, 2, 4, 2, 5};
    printContainer(vec, "Before remove");
    vec.erase(remove(vec.begin(), vec.end(), 2), vec.end());
    printContainer(vec, "After remove 2s");
}
```

```
void demonstrateSTLContainers() {
    cout << "STL Containers:" << endl;

    // List
    list<int> lst = {3, 1, 4, 1, 5};
    lst.sort();
    cout << "Sorted list: ";
    for (int n : lst) cout << n << " ";
    cout << endl;

    // Map
    map<string, int> m = {{"apple", 5}, {"banana", 3}, {"cherry", 8}};
    cout << "Map: ";
    for (const auto& p : m) {
        cout << p.first << ":" << p.second << " ";
    }
    cout << endl;

    // Set
    set<int> s = {3, 1, 4, 1, 5, 9, 2, 6};
    cout << "Set: ";
    for (int n : s) cout << n << " ";
    cout << endl;

    // Stack
    stack<int> stk;
    stk.push(1); stk.push(2); stk.push(3);
    cout << "Stack pop: ";
    while (!stk.empty()) {
        cout << stk.top() << " ";
        stk.pop();
    }
    cout << endl;

    // Queue
    queue<string> q;
    q.push("first"); q.push("second"); q.push("third");
    cout << "Queue: ";
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;
}

void demonstrateSTLAlgorithms() {
    cout << "STL Algorithms:" << endl;

    vector<int> vec = {64, 34, 25, 12, 22, 11, 90};
    printContainer(vec, "Original");

    // Sort
    sort(vec.begin(), vec.end());
    printContainer(vec, "Sorted");
```

```
    // Binary search
    bool found = binary_search(vec.begin(), vec.end(), 25);
    cout << "Found 25: " << (found ? "Yes" : "No") << endl;

    // Find
    auto it = find(vec.begin(), vec.end(), 22);
    if (it != vec.end()) {
        cout << "Found 22 at position: " << distance(vec.begin(), it) << endl;
    }

    // Count
    vector<int> nums = {1, 2, 3, 2, 4, 2, 5};
    int count2s = count(nums.begin(), nums.end(), 2);
    cout << "Count of 2s: " << count2s << endl;

    // Transform
    vector<int> squares;
    transform(nums.begin(), nums.end(), back_inserter(squares),
              [](int x) { return x * x; });
    printContainer(squares, "Squares");

    // Accumulate
    int sum = accumulate(nums.begin(), nums.end(), 0);
    cout << "Sum: " << sum << endl;
}

void demonstrateIterators() {
    cout << "Iterators:" << endl;

    vector<int> vec = {1, 2, 3, 4, 5};

    // Forward iterator
    cout << "Forward: ";
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    // Reverse iterator
    cout << "Reverse: ";
    for (auto it = vec.rbegin(); it != vec.rend(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    // Range-based for loop
    cout << "Range-based: ";
    for (const auto& element : vec) {
        cout << element << " ";
    }
    cout << endl;
}

void demonstrateAdvancedSTL() {
    cout << "Advanced STL:" << endl;
```

```
    // Lambda with algorithm
    vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Count even numbers
    int evenCount = count_if(vec.begin(), vec.end(), [](int x) { return x % 2
== 0; });
    cout << "Even numbers: " << evenCount << endl;

    // Remove odd numbers
    vec.erase(remove_if(vec.begin(), vec.end(), [](int x) { return x % 2 != 0;
}), vec.end());
    printContainer(vec, "After removing odds");

    // Priority queue
    priority_queue<int> pq;
    pq.push(3); pq.push(1); pq.push(4); pq.push(1); pq.push(5);
    cout << "Priority queue: ";
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    cout << endl;

    // Unordered map
    unordered_map<string, int> umap = {{"one", 1}, {"two", 2}, {"three", 3}};
    cout << "Unordered map: ";
    for (const auto& p : umap) {
        cout << p.first << ":" << p.second << " ";
    }
    cout << endl;
}

int main() {
    cout << "Q25: STL Demo" << endl;

    try {
        demonstrateVectorOperations();
        demonstrateSTLContainers();
        demonstrateSTLAlgorithms();
        demonstrateIterators();
        demonstrateAdvancedSTL();

    } catch (const exception& e) {
        cout << "Exception: " << e.what() << endl;
    }

    cout << "STL demonstration completed" << endl;
    return 0;
}
```

Terminal Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

```
cd "/Users/abhinavbansal/Desktop/NLP model 2/" && g++ Q25_STL.cpp -o Q25_STL && "/Users
/abhinavbansal/Desktop/NLP model 2/"Q25_STL
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 % cd "/Users/abhinavbansal/Desktop/N
LP model 2/" && g++ Q25_STL.cpp -o Q25_STL && "/Users/abhinavbansal/Desktop/NLP model 2
/"Q25_STL
Q25: STL Demo
Vector Operations:
Original: 10 20 30 40 50
After insert: 10 20 25 30 40 50
After erase single: 10 25 30 40 50
After erase range: 10 40 50
Before remove: 1 2 3 2 4 2 5
After remove 2s: 1 3 4 5
STL Containers:
Sorted list: 1 1 3 4 5
Map: apple:5 banana:3 cherry:8
Set: 1 2 3 4 5 6 9
Stack pop: 3 2 1
Queue: first second third
STL Algorithms:
Original: 64 34 25 12 22 11 90
Sorted: 11 12 22 25 34 64 90
Found 25: Yes
Found 22 at position: 2
Count of 2s: 3
Squares: 1 4 9 4 16 4 25
Sum: 19
Iterators:
Forward: 1 2 3 4 5
Reverse: 5 4 3 2 1
Range-based: 1 2 3 4 5
Advanced STL:
Even numbers: 5
After removing odds: 2 4 6 8 10
Priority queue: 5 4 3 1 1
Unordered map: three:3 two:2 one:1
STL demonstration completed
abhinavbansal@Abhinav-ka-MacBook-Air-2 NLP model 2 %
```