Predicted C++ Final Exam Paper

Question 1: Class Design and Hierarchy (40 marks)

Part a (20 marks)

Write code for the following: In the hierarchy of shapes (Circle, Rectangle), implement a **Shape** base class with a **virtual show()** function. Derived classes should override this function and implement specific details for the shape (like `Circle` and `Rectangle`). Use inheritance and polymorphism to demonstrate the code.

Part b (20 marks)

Define an **abstract class** with a **pure virtual function show()** and add a data member. Write two child classes (`Circle` and `Rectangle`) that override `show()`. Demonstrate with appropriate constructors, getters, and setters.

Code for Question 1

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

// Abstract base class
class Shape {
protected:
    string name;
public:
    virtual void show() = 0;  // Pure virtual function to display shape
    virtual double area() = 0;  // Pure virtual function for area calculation
};

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

```
    Circle(string n, double r) : radius(r) { name = n; }
    void show() override {
        cout << "Shape: " << name << "\nRadius: " << radius << endl;
    }
    double area() override {
        return 3.14 * radius * radius;
    }
};

// Derived class Rectangle
class Rectangle : public Shape {
private:
    double length, width;
public:
    Rectangle(string n, double l, double w) : length(l), width(w) {
       name = n; }
    void show() override {
        cout << "Shape: " << name << "\nLength: " << length <<
           "\nWidth: " << width << endl;
    }
    double area() override {
        return length * width;
    }
};

int main() {
    Circle c("Circle", 5);
    Rectangle r("Rectangle", 4, 6);
    Shape* s1 = &c;  // Upcasting to Shape pointer
    Shape* s2 = &r;
    s1->show();  // Polymorphism
    cout << "Area: " << s1->area() << endl;
    s2->show();  // Polymorphism
    cout << "Area: " << s2->area() << endl;
    return 0;
}
```

Question 2: Theory-based Concepts (40 marks)

Part a (8 marks)

Describe **polymorphism** with an example. Explain how virtual functions enable polymorphism in C++.

Part b (8 marks)

Explain **exception handling** in C++. Provide an example with custom exceptions.

Part c (8 marks)

Explain **multiple inheritance** and **virtual inheritance** with examples. Discuss how virtual inheritance solves the **diamond problem**.

Part d (8 marks)

What is **upcasting** and **downcasting**? Provide code examples to demonstrate both.

Answer for Question 2

- **Polymorphism**: Polymorphism in C++ allows a base class pointer or reference to call derived class methods at runtime. Virtual functions enable this by allowing derived classes to override base class functions.

```
class Base {
public:
    virtual void show() { cout << "Base class\n"; }
};
class Derived : public Base {
public:
    void show() override { cout << "Derived class\n"; }
};
int main() {
    Base* b = new Derived();
    b->show();  // Calls Derived's show due to polymorphism
}
```

- **Exception Handling**: Uses `try`, `catch`, and `throw`. Custom exceptions can inherit from `std::exception`.

```
class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Custom Exception!";
    }
};
int main() {
    try {
        throw MyException();
    } catch (const MyException& e) {
        std::cout << e.what() << std::endl;
    }
}
```

- **Multiple and Virtual Inheritance**: Multiple inheritance allows a class to inherit from multiple base classes. Virtual inheritance prevents duplicate base class instances in the diamond problem.

```
class A {
public:
    virtual void show() { cout << "Class A\n"; }
};
class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {
public:
    void show() override { cout << "Class D\n"; }
};
```

- **Upcasting and Downcasting**: Upcasting converts a derived class pointer to a base class pointer; downcasting converts a base class pointer to a derived class pointer using `dynamic_cast`.

```
Base* b = new Derived();  // Upcasting
Derived* d = dynamic_cast<Derived*>(b);  // Downcasting
```

Question 3: Templates, Arrays, and Inheritance (40 marks)

Part a (10 marks)

Write a **template class** to store an array of values of the same type. Include a constructor, getter, setter, and overloaded operators for + and - for element-wise operations.

Part b (10 marks)

Write a **template function** to swap two variables of any type. Implement exception handling, assuming swap operations might throw exceptions.

Part c (10 marks)

Write code for **2D array handling** using templates. Provide code to access elements and manipulate rows and columns.

Code for Question 3

Part a

```
template<typename T>
class Array {
    T* data;
    int size;
public:
```

```
    Array(int s) : size(s) {
        data = new T[s];
    }
    ~Array() { delete[] data; }
    T get(int i) { return data[i]; }
    void set(int i, T value) { data[i] = value; }
    Array operator+(const Array& rhs) const {
        Array result(size);
        for (int i = 0; i < size; i++) {
            result.set(i, data[i] + rhs.get(i));
        }
        return result;
    }
    Array operator-(const Array& rhs) const {
        Array result(size);
        for (int i = 0; i < size; i++) {
            result.set(i, data[i] - rhs.get(i));
        }
        return result;
    }
};
```

Part b

```
template<typename T>
void swap(T& a, T& b) {
    try {
        T temp = a;
        a = b;
        b = temp;
    } catch (const std::exception& e) {
        std::cerr << "Exception during swap: " << e.what() <<
           std::endl;
    }
}
```

Part c

```
template<typename T>
class Matrix {
    T** data;
    int rows, cols;
public:
    Matrix(int r, int c) : rows(r), cols(c) {
        data = new T*[rows];
        for (int i = 0; i < rows; i++) {
            data[i] = new T[cols];
        }
    }
    ~Matrix() {
```

```
        for (int i = 0; i < rows; i++) {
            delete[] data[i];
        }
        delete[] data;
    }
    T get(int i, int j) { return data[i][j]; }
    void set(int i, int j, T value) { data[i][j] = value; }
};
```

Question 4: Binary File Handling and Classes (40 marks)

Part a (10 marks)

Consider a class with fixed-length data members. Write functions to read and write objects in an opened binary file stream.

Part b (10 marks)

Write code for a **Text class** similar to the built-in string class. Implement functions like `append`, `length`, and `compare`.

Part c (10 marks)

Create a class with **N pointers** as data members, some of which may be `NULL` or `nullptr`. Write code to handle and manage these pointers, ensuring no null pointer dereferences.

Part d (10 marks)

Write code for **Queue** (FIFO) and **Stack** (LIFO) classes with methods like `enqueue`, `dequeue`, `push`, `pop`.

Code for Question 4

Part a

```
class Person {
public:
    char name[50];
    int age;
};
void writePersonToFile(std::ofstream& ofs, const Person& p) {
    ofs.write(reinterpret_cast<const char*>(&p), sizeof(Person));
}
void readPersonFromFile(std::ifstream& ifs, Person& p) {
    ifs.read(reinterpret_cast<char*>(&p), sizeof(Person));
}
```

Part b

```
class Text {
    std::string data;
public:
    Text(const std::string& str = "") : data(str) {}
    void append(const std::string& str) { data += str; }
    size_t length() const { return data.length(); }
    int compare(const Text& other) const { return
       data.compare(other.data); }
};
```

Part c

```
class PointerArray {
    int* arr[10];
public:
    PointerArray() {
        for (int i = 0; i < 10; i++) {
            arr[i] = nullptr;
        }
    }
    ~PointerArray() {
        for (int i = 0; i < 10; i++) {
            delete arr[i];
        }
    }
    void set(int index, int* ptr) {
        if (index < 0 || index >= 10) {
            throw std::out_of_range("Index out of range");
        }
        arr[index] = ptr;
    }
    int* get(int index) const {
        if (index < 0 || index >= 10) {
            throw std::out_of_range("Index out of range");
        }
        return arr[index];
    }
};
```

Part d

```
template<typename T>
class Queue {
    T* data;
    int front, rear, capacity;
public:
    Queue(int size) : front(-1), rear(-1), capacity(size) {
        data = new T[size];
```

```
    }
    ~Queue() { delete[] data; }
    void enqueue(T value) {
        if ((rear + 1) % capacity == front) {
            std::cout << "Queue is full" << std::endl;
            return;
        }
        if (front == -1) front = 0;
        rear = (rear + 1) % capacity;
        data[rear] = value;
    }
    void dequeue() {
        if (front == -1) {
            std::cout << "Queue is empty" << std::endl;
            return;
        }
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % capacity;
        }
    }
};

template<typename T>
class Stack {
    T* data;
    int top, capacity;
public:
    Stack(int size) : top(-1), capacity(size) {
        data = new T[size];
    }
    ~Stack() { delete[] data; }
    void push(T value) {
        if (top == capacity - 1) {
            std::cout << "Stack is full" << std::endl;
            return;
        }
        data[++top] = value;
    }
    void pop() {
        if (top == -1) {
            std::cout << "Stack is empty" << std::endl;
            return;
        }
        top--;
    }
};
```