National University of Computer & Emerging Sciences, Karachi
Spring-2025 School of Computing
Quiz No. 2
25th March 2025

Course Code: CS-1004	**Course Name: Object Oriented Programming**
Instructor Name / Names: Ms. Atiya Jokhio	
Section-C	**Student-ID:**

Time Allowed: 30 minutes. **Total Points**: 10

TYPE B

Question:1 Encircle the correct statement **[3 points]**

i) What will fin.read((char*)&obj, sizeof(obj)); do?
a) Write the object into the file.
b) Read the object from the file into obj.
c) Read the file as a text string.
d) Check the size of the file.
Answer: b) Read the object from the file into obj.

Given the statement fout.write((char*)&obj, sizeof(obj));, what does (char*)&obj mean?
a) It converts the object into a readable text format.
b) It points to the memory location of the object for binary writing.
c) It encrypts the object before saving.
d) It copies the object directly to file without size info.
Answer: b) It points to the memory location of the object for binary writing.

What is the main difference between put() and write()?
a) put() writes one character, write() writes multiple characters or objects.
b) put() is for text files only, write() is for binary files.
c) put() formats data, write() does not.
d) No difference; both are identical.
Answer: a) put() writes one character, write() writes multiple characters or objects.

Question:2 **[6 points]**

Design an abstract base class Passenger with a function displayType() that shows the type of passenger (implemented inside the base class) and a **pure virtual function** bookTicket() to be overridden by all derived classes.

- Create three derived classes: Each class must implement its specific version of bookTicket() and their own specific behaviors.
  - **RegularPassenger**
  - **FrequentFlyer**
  - **VIPPassenger**:

You need to **Demonstrating Polymorphism** by storing different types of passengers in an array of Passenger* and calling the respective methods on each passenger.

.

Solution:

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

```
// Abstract base class
class Passenger {
public:
    // Virtual destructor for proper cleanup
    virtual ~Passenger() {}

    // Concrete function implemented in the base class
    void displayType() {
        std::cout << "Passenger Type: " << getPassengerType() << std::endl;
    }

    // Pure virtual function that must be implemented by derived classes
    virtual void bookTicket() = 0;

protected:
    // This method will be implemented in derived classes to return the type
    virtual std::string getPassengerType() = 0;
};

// Derived class 1: RegularPassenger
class RegularPassenger : public Passenger {
public:
    void bookTicket() override {
        std::cout << "Booking a regular ticket..." << std::endl;
    }

protected:
    std::string getPassengerType() override {
        return "Regular Passenger";
    }
};

// Derived class 2: FrequentFlyer
class FrequentFlyer : public Passenger {
public:
    void bookTicket() override {
        std::cout << "Booking a ticket with frequent flyer discount..." << std::endl;
    }

protected:
    std::string getPassengerType() override {
        return "Frequent Flyer";
    }
};

// Derived class 3: VIPPassenger
class VIPPassenger : public Passenger {
```

```
public:
    void bookTicket() override {
        std::cout << "Booking a VIP ticket with priority services..." << std::endl;
    }

protected:
    std::string getPassengerType() override {
        return "VIP Passenger";
    }
};

int main() {
    const int numPassengers = 3;
    Passenger* passengers[numPassengers]; // Array of Passenger* pointers

    // Storing different types of passengers in the array
    passengers[0] = new RegularPassenger();
    passengers[1] = new FrequentFlyer();
    passengers[2] = new VIPPassenger();

    // Looping through the array and calling both displayType and bookTicket
    for (int i = 0; i < numPassengers; ++i) {
        passengers[i]->displayType();  // Display type of passenger
        passengers[i]->bookTicket();   // Call the overridden bookTicket method
        std::cout << std::endl;
    }

    // Proper memory management: Cleaning up dynamically allocated memory
    for (int i = 0; i < numPassengers; ++i) {
        delete passengers[i];
    }

    return 0;
}
```