Predicted Exam Paper

Based on your professor's notes, here's a plausible set of questions for your final exam. Remember, the key is to demonstrate a solid understanding of Object-Oriented Programming (OOP) concepts, data structures, and file handling in C++.

Question # 01

[40 Marks: 20 each]

Instructions: Write complete C++ code for the classes described below. Ensure you demonstrate an understanding of inheritance, polymorphism, and code reusability. Standard getters, setters, and constructors (default and parameterized for all data members) should be included; you can use the comment `// standard getters, setters and constructors (defaults and for all data members) are part of the class` to save time writing their full implementations.

a) Shape Hierarchy and Reusability (20 Marks)

Consider a hierarchy of geometric shapes. You are required to implement the following:

1. A base class named `Shape`.
    a. **Protected data members:** `name` (string), `type` (string).
    b. **Public member functions:**
        i. A constructor to initialize `name` and `type`.
        ii. A virtual function `double calculateArea()` that returns 0.0.
        iii. A function `void displayShapeInfo()` that prints the name and type of the shape.
        iv. Appropriate getters and setters.
2. A derived class named `Circle` that inherits publicly from `Shape`.
    a. **Protected data members:** `radius` (double).
    b. **Public member functions:**
        i. A constructor that initializes the `Shape` part and the `radius`. It should reuse the `Shape` constructor.
        ii. Override `double calculateArea()` to calculate and return the area of a circle (Formula: π·radius2).
        iii. Override `void displayShapeInfo()` to display the circle's name, type, and radius. It should reuse the `Shape::displayShapeInfo()` function.
        iv. Appropriate getters and setters for `radius`.
3. A derived class named `Rectangle` that inherits publicly from `Shape`.
    a. **Protected data members:** `length` (double), `width` (double).
    b. **Public member functions:**
        i. A constructor that initializes the `Shape` part and the `length` and `width`. It should reuse the `Shape` constructor.
        ii. Override `double calculateArea()` to calculate and return the area of a rectangle (Formula: length·width).
        iii. Override `void displayShapeInfo()` to display the rectangle's name, type, length, and width. It should reuse the `Shape::displayShapeInfo()` function.
        iv. Appropriate getters and setters for `length` and `width`.

Hint: Focus on how constructors of derived classes can call constructors of the base class. Pay attention to function overriding and the use of `virtual` functions for polymorphism.

b) Abstract Class and Polymorphism (20 Marks)

Define an abstract C++ class named `Employee`.

1. The `Employee` class should have the following:

    a. **Protected data members:** `employeeID` (int), `employeeName` (string).
    b. **Public member functions:**
        i. A constructor to initialize `employeeID` and `employeeName`.
        ii. A non-virtual function `void show()` that displays the `employeeID` and `employeeName`.
        iii. A pure virtual function `double calculatePay()` that returns 0.0.
        iv. Appropriate getters and setters.
2. Create two child classes, `SalariedEmployee` and `HourlyEmployee`, that inherit publicly from `Employee`.
    a. `SalariedEmployee`**:**
        i. **Additional protected data members:** `annualSalary` (double), `performanceBonus` (double).
        ii. A constructor to initialize all its data members and the base class members.
        iii. Override the `calculatePay()` function to return the monthly salary (annual salary / 12 + performance bonus).
        iv. Override the `show()` function to display all details of a salaried employee, including information from the base `Employee` class.
    b. `HourlyEmployee`**:**
        i. **Additional protected data members:** `hourlyRate` (double), `hoursWorked` (double).
        ii. A constructor to initialize all its data members and the base class members.
        iii. Override the `calculatePay()` function to return the pay (hourly rate * hours worked).
        iv. Override the `show()` function to display all details of an hourly employee, including information from the base `Employee` class.

Hint: An abstract class cannot be instantiated. Pure virtual functions make a class abstract. Ensure derived classes provide concrete implementations for all pure virtual functions.

Question # 02

[40 Marks: 8 each]

Instructions: Write precise descriptions for each of the following concepts. Support your answers with C++ code examples where appropriate.

a) Polymorphism in OOP:

Explain the concept of polymorphism in Object-Oriented Programming. Describe the difference between compile-time (static) polymorphism and run-time (dynamic) polymorphism. Provide C++ code examples to illustrate both types (e.g., function overloading for static, virtual functions for dynamic).

b) Inheritance and its Types:

Define inheritance and explain its importance in C++. Describe at least three types of inheritance (e.g., single, multiple, multilevel, hierarchical, hybrid) with simple C++ code snippets or diagrams illustrating their structure. Discuss the concept of access specifiers (public, protected, private) in the context of inheritance.

c) Abstract Classes vs. Concrete Classes:

Explain the difference between an abstract class and a concrete class in C++. Discuss the role of pure virtual functions in creating abstract classes. Provide a small C++ code example demonstrating an abstract class and a class that inherits from it. When would you choose to use an abstract class?

d) Friend Functions and Friend Classes:

What are friend functions and friend classes in C++? Explain why they might be used, and discuss the implications of their use on encapsulation. Provide a C++ code example of a friend function accessing private members of a class.

e) The this Pointer:

Explain the purpose and usage of the this pointer in C++. Provide C++ code examples where the this pointer is implicitly used and where it is explicitly used (e.g., to resolve name conflicts between member variables and parameters, or to return a reference to the current object from a member function).

Question # 03

[40 Marks: 10 each]

Instructions: Provide C++ code or answer the following parts, keeping in mind a comprehensive template class `Matrix<T>` designed to store a 2D array of values of the same type `T`. Assume this class has protected data members (e.g., `T** data_ptr`, `int rows`, `int cols`), standard conventional names for getters/setters, all possible constructors (default, parameterized, copy), overloaded operators (e.g., `+`, `-`, `*`, `[]`), and other related member functions.

a) Template Specialization for Matrix<char> (10 Marks)

If you were to specialize the Matrix<T> template for T = char to handle character matrices (e.g., for displaying patterns or simple text art), describe what specific functionalities or member functions you might add or modify in this specialization. Provide a conceptual C++ code snippet for the declaration of such a specialization and one unique member function.

b) Exception Handling for Matrix Operations (10 Marks)

Consider a member function T getElement(int r, int c) in your Matrix<T> class that returns the element at row r and column c. Write the code for this function, including exception handling. It should throw an std::out_of_range exception if the provided row or column indices are invalid (i.e., outside the matrix dimensions).

c) Extending Matrix<T> to Matrix3D<T> (10 Marks)

Describe how you would design a new template class Matrix3D<T> that inherits from or utilizes your Matrix<T> class (or its concepts) to represent a 3D matrix (e.g., T*** data_ptr_3d, int depth, int rows, int cols). Briefly outline the key data members and one significant member function (e.g., a constructor or an element access function getElement3D(int d, int r, int c)) for Matrix3D<T>. You can use a mix of English description and pseudo-code/C++ declarations.

d) Validating Template Parameters for Matrix<T> (10 Marks)

Suppose you want to restrict the type T in Matrix<T> to only arithmetic types (like int, float, double). How could you enforce such a constraint at compile time? Describe a C++ technique (e.g., using static_assert with type traits like std::is_arithmetic) to achieve this. Provide a conceptual code snippet within the Matrix<T> class definition.

Question # 04

[40 Marks: 10 each]

Instructions: Provide C++ code or answer the following parts. These parts are not directly related to each other.

a) Binary File I/O with Fixed-Length Records (10 Marks)

Consider a class Student where each data member is of a fixed length (e.g., char studentID[10], char name[50], int age). You need to write a C++ function void updateStudentAge(std::fstream &fs, const char* targetStudentID, int newAge) that finds a Student object in an opened binary file stream (fs) by studentID and updates its age. Assume the file is opened in binary mode for both reading and writing. The file cursor should initially be at the beginning (fs.seekg(0, std::ios::beg);).

```
C++
// Assume Student class structure:
// class Student {
// public:
//     char studentID[10];
//     char name[50];
//     int age;
//     // ... other members, constructors, etc.
// };

void updateStudentAge(std::fstream &fs, const char* targetStudentID, int newAge) {
    // Your code here
    // 1. Read Student records one by one.
    // 2. Compare studentID with targetStudentID.
    // 3. If found, move file pointer back to the start of that record.
    // 4. Update the age and write the modified Student object back.
}
```

b) Implementing a Substring Function for a Custom Text Class (10 Marks)

Consider a custom class Text that mimics some functionalities of the std::string class. It internally manages a dynamically allocated C-style string.

```
C++
class Text {
    char* data;
    size_t length;
public:
    // ... constructors, destructor, getters, etc. ...
    Text(const char* s = ""); // Constructor
    ~Text();                  // Destructor
    Text(const Text& other);  // Copy Constructor
    Text& operator=(const Text& other); // Copy Assignment

    // TODO: Implement the substring function
    Text substring(size_t start_index, size_t num_chars) const;
};
```

Write the implementation for the `Text substring(size_t start_index, size_t num_chars) const` member function. This function should return a new `Text` object representing the portion of the current `Text` object starting at `start_index` and containing `num_chars` characters. Include basic error checking (e.g., if `start_index` is out of bounds, or `num_chars` requests more characters than available from `start_index`, adjust `num_chars` or return an empty `Text` object). Specify any assumptions you make.

c) Processing an Array of Pointers with NULL Values (10 Marks)

A class Resource has a data member int resourceID; and a member function void processResource(). In your main logic, you have an array Resource* resource_pointers[N]; (where N is a constant). Some pointers in this array might be NULL or nullptr. Write a C++ code snippet that iterates through this array, and for every non-null pointer, it calls the processResource() method and prints the resourceID.

```
C++
// Assume Resource class structure:
// class Resource {
// public:
//     int resourceID;
//     void processResource() { /* ... processes the resource ... */ std::cout << "Processing
" << resourceID << std::endl;}
//     Resource(int id) : resourceID(id) {}
// };
//
// const int N = 10; // Example size
// Resource* resource_pointers[N];
```

```
// // ... array is populated, some elements might be nullptr ...
```

Your code should safely handle the null pointers.

d) Implementing Queue Peek using Stacks (10 Marks)

Consider a class Queue<T> (First-In, First-Out) that is implemented using two Stack<T> objects (Last-In, First-Out). The Queue<T> class has the standard push(T value) and T pop() operations. You need to implement the T peek() member function for this Queue<T> class. The peek() function should return the value at the front of the queue without removing it. Describe the logic and provide the C++ code for T peek(), assuming you have two stacks, say stack_in and stack_out, as data members.

Hint: To peek(), you might need to transfer elements from one stack to another to bring the front element of the queue to the top of one of the stacks, and then transfer them back to maintain the queue's state if necessary.

C++

```
// template <typename T>
// class Stack {
// public:
//     void push(T val);
//     T pop();
//     T top() const;
//     bool isEmpty() const;
// };

template <typename T>
class Queue {
    Stack<T> stack_in;
    Stack<T> stack_out;
public:
    // Assume push and pop are already implemented correctly
    void push(T value) {
        stack_in.push(value);
    }

    T pop() {
        if (stack_out.isEmpty()) {
            while (!stack_in.isEmpty()) {
                stack_out.push(stack_in.pop());
            }
        }
        if (stack_out.isEmpty()) {
            throw std::runtime_error("Queue is empty"); // Or handle as per requirements
        }
        return stack_out.pop();
    }

    T peek() const {
        // Your code here:
        // 1. If stack_out is not empty, its top is the front of the queue.
        // 2. If stack_out is empty, transfer all elements from stack_in to stack_out.
        //    The top of stack_out will then be the front.
        // 3. IMPORTANT: This is a const function. If you modify the stacks
        //    (like transferring elements), you must ensure the logical state
        //    of the queue (as observed by non-const operations like pop)
        //    is preserved. A common way for peek() in a two-stack queue is to
        //    make stack_out mutable or to perform the transfer, peek, and then
        //    transfer back if strict const-correctness for internal state is required,
        //    or, more practically, design pop() and peek() to share the same logic
        //    for bringing the front element to stack_out.
        //
        // For this question, let's assume stack_out might need to be populated
```

```
        // if it's empty, similar to how pop() works.
    }
};
```

This is my best guess based on the information. The key will be to understand the underlying C++ concepts your professor wants to test in each section. Good luck with your preparation! 👍

Would you like me to elaborate on any specific question or concept? Or perhaps we can start a learning plan on one of these topics if you feel unsure about any of them?