- Due No due date
- Points 0
- Available after 11 Aug at 0:00

Sample Practical Exam 1

Instructions

This exam is CLOSED BOOK and must be completed on the lab computers during your enrolled practical session. You are permitted to use the following resources:

- C++ reference **https://cplusplus.com/reference/ (https://cplusplus.com/reference/)**
- C++ tutorial **https://cplusplus.com/doc/ (https://cplusplus.com/doc/)**
- One blank A4 paper

Accessing other websites or resources (including but not limited to previous assignments, myuni materials, github, google, AI tools and plugins) is NOT permitted.

PLEASE ENSURE YOU ARE CODING ON REDHAT LINUX, as you will not be able to compile the code otherwise.

The work you submit must be your own and you must not receive any assistance in producing your work. You must not discuss the questions with anyone who has not completed the exam.

BEFORE YOU LEAVE: All paper used for working out must handed in to tutors. It must NOT be removed from the room.

Submit the all code files with correct names and extensions to Gradescope

You have 90 minutes to work on this exam, once this quiz closes, no further work will be accepted.

Question 1

Which class **violates** the Single Responsibility Principle?

```
// Option A
class EmailSender {
    void sendEmail(string message) { /* send email */ }
};

// Option B
class UserManager {
    void createUser(User user) { /* create user */ }
    void deleteUser(int userId) { /* delete user */ }
    void sendWelcomeEmail(User user) { /* send email */ }
    void generateUserReport(int userId) { /* generate report */ }
};

// Option C
class Calculator {
    int add(int a, int b) { return a + b; }
    int subtract(int a, int b) { return a - b; }
};
```

A) Option A
B) Option B
C) Option C
D) None of the above

Answer: B - UserManager has multiple responsibilities: user management, email sending, and report generation.

Question 2

Which design best **follows** the Open/Closed Principle for adding new shape types?

// Design A

```
class Shape {
public:
    virtual double calculateArea() = 0;
};
class Circle : public Shape {
    double calculateArea() override { /* implementation */ }
};

// Design B
class AreaCalculator {
    double calculateArea(string shapeType, double param1, double param2) {
        if (shapeType == "circle") return 3.14 * param1 * param1;
        else if (shapeType == "rectangle") return param1 * param2;
        // Need to modify this method for new shapes
    }
};
```

A) Design A
B) Design B
C) Both designs
D) Neither design

Answer: A - Design A allows extension through inheritance without modifying existing code.

Question 3

Which **violates** the Liskov Substitution Principle?

```
class Bird {
public:
    virtual void fly() { cout << "Flying"; }
```

```
};

// Option A
class Eagle : public Bird {
    void fly() override { cout << "Eagle flying high"; }
};

// Option B
class Penguin : public Bird {
    void fly() override {
        throw exception("Penguins cannot fly!");
    }
};

// Option C
class Sparrow : public Bird {
    void fly() override { cout << "Sparrow flying fast"; }
};
```

A) Eagle class
B) Penguin class
C) Sparrow class
D) None violate LSP

Answer: B - Penguin changes the expected behavior by throwing an exception instead of flying.

Question 4

Which interface design **violates** the Interface Segregation Principle?

// Option A

```
interface Workable {
    void work();
};
interface Eatable {
    void eat();
};

// Option B
interface Worker {
    void work();
    void eat();
    void sleep();
    void attendMeeting();
    void writeCode();
    void designUI();
};

// Option C
interface Printable {
    void print();
};
```

A) Option A
B) Option B
C) Option C
D) All are good designs
Answer: B - Worker interface forces classes to implement methods they might not need.

Question 5

Which code better **follows** the Dependency Inversion Principle?

```
// Version A
class EmailService {
    void sendEmail(string message) { /* send email */ }
};
class NotificationManager {
    EmailService emailService;
public:
    void sendNotification(string message) {
        emailService.sendEmail(message);
    }
};

// Version B
interface MessageService {
    virtual void sendMessage(string message) = 0;
};
class NotificationManager {
    MessageService* messageService;
public:
    NotificationManager(MessageService* service) : messageService(service) {}
    void sendNotification(string message) {
        messageService->sendMessage(message);
    }
};
```

A) Version A
B) Version B
C) Both are equivalent
D) Neither follows DIP

Answer: B - Version B depends on abstraction (interface) rather than concrete implementation.

Question 6

Analyze this code. Which SOLID principle is **violated**?

```
class OrderProcessor {
    void processOrder(Order order) {
        // Validate order
        if (order.items.empty()) throw exception("Empty order");

        // Calculate total
        double total = 0;
        for (auto item : order.items) {
            total += item.price * item.quantity;
        }

        // Save to database
        Database db;
        db.save(order);

        // Send confirmation email
        cout << "Sending email to " << order.customerEmail << endl;

        // Print receipt
        cout << "Printing receipt..." << endl;

        // Update inventory
        InventoryManager inv;
        inv.updateStock(order.items);
```

```
    }
};
```

A) Single Responsibility Principle
B) Open/Closed Principle
C) Liskov Substitution Principle
D) Interface Segregation Principle
Answer: A - The class handles validation, calculation, database operations, email, printing, and inventory - too many responsibilities.

Question 7

What is the output of this recursive function when called with mystery(4)?

```
int mystery(int n) {
    if (n <= 1) return 1;
    return n * mystery(n - 1);
}
```

A) 4
B) 10
C) 24
D) Stack overflow

Answer: C - This computes factorial: 4! = 4 × 3 × 2 × 1 = 24.

Question 8

Which function is tail recursive?

```
// Function A
int sumA(int n) {
    if (n <= 0) return 0;
    return n + sumA(n - 1);
```

```
}

// Function B
int sumB(int n, int accumulator = 0) {
    if (n <= 0) return accumulator;
    return sumB(n - 1, accumulator + n);
}

// Function C
int sumC(int n) {
    if (n <= 0) return 0;
    int result = sumC(n - 1);
    return n + result;
}
```

A) Function A
B) Function B
C) Function C
D) All are tail recursive
Answer: B - Function B has the recursive call as the last operation with no computation after it.

Question 9

What is the primary benefit of using Abstract Data Types?

A) Faster execution speed
B) Reduced memory usage
C) Data encapsulation and interface abstraction
D) Automatic memory management

Answer: C - ADTs hide implementation details and provide clean interfaces.

Question 10

Which statement best describes the relationship between ADT and its implementation?

A) ADT and implementation must use the same data structure
B) ADT defines what operations are available, implementation defines how
C) ADT is always faster than direct implementation
D) ADT can only be implemented using arrays

Answer: B - ADT specifies interface and behavior, implementation handles the "how".

Question 11

For a Stack ADT, compare these implementations:

```
// Implementation A: Array-based
class ArrayStack {
    int arr[1000];
    int top;
    // Fixed size, O(1) operations
};

// Implementation B: Linked List-based
class LinkedStack {
    Node* head;
    // Dynamic size, O(1) operations
};
```

When would you choose Implementation A over B?

A) When you need unlimited capacity
B) When memory usage is critical and size is predictable
C) When frequent resizing is expected
D) Never, B is always better

Answer: B - Array implementation has better memory locality and lower overhead when size is known.

Coding Question 1: Magic Potion Inventory

Create a program for a Magic Potion Inventory in files practice1.h & practice1.cpp. All the methods should go in a class called "PotionCraft"

The Magic Potion Inventory has the following behaviors and properties:

- The inventory uses a vector of strings to store potion names in the order they were brewed
- You can brew any potion (including duplicates) and add it to the inventory
- You can consume a specific potion by name (removes the first instance if duplicates exist) from the inventory. If the potion doesn't exist, return an empty string ("")
- You can brew multiple potions at once by providing a list (vector of potion names). All potions are added to the inventory in the order they appear in the recipe
- You can copy all current potions to an external container by reference. The external container will have all inventory potions appended to it, while the original inventory remains unchanged

Method Definitions:

Method	Parameters	Return Type	Description
getPotions()	None	std::vector<std::string>	Return all potions in inventory
brewPotion(std::string name)	string	void	Add a potion to inventory
consumePotion(std::string name)	string	string	Remove specific potion by name

brewMultiplePotions(std::vector<std::string> potions)	vector<string>	void	Add multiple potions to inventory
fillExternalContainer(std::vector<std::string>& container)	vector<string>&	vector<string>	Fills an external container with all current potions without removing them from inventory

Coding Question 2: Modified Fibonacci

Write a C++ program modified_fibonacci.cpp to generate the first n numbers of the Modified Fibonacci sequence using tail recursion.

The Modified Fibonacci sequence is defined as:

MF(0) = 2

MF(1) = 3

MF(n) = MF(n-1) * MF(n-2) for n ≥ 2

Requirements:

- Use tail recursion for the main logic - the recursive call must be the last operation
- The function must generate a vector containing the first n Modified Fibonacci numbers
- Helper functions with accumulator parameters are required
- Follow SOLID and ADT principles where/if applicable
- Do NOT put the function inside a class

Method Definitions:

Method	Parameters	Return Type	Description
modifiedFibonacci()	int	std::vector<int>	Generate first n numbers of Modified Fibonacci sequence using tail recursion