University of Sarajevo
Faculty of Electrical Engineering
Department of Data Science and Artificial Intelligence

Laboratory Exercise 2

Artificial Intelligence

Authors: Benjamin Bandić, Nedim Bečić
Professor: Izudin Džafić

Contents

1 Goal of the Exercise

The goal of this laboratory exercise is to apply key concepts of Object-Oriented Programming (OOP) and fundamental C++ data structures to solve a practical problem. Students will build a port logistics simulation system, which involves modeling real-world entities like containers, ships, and trucks using classes and inheritance. The exercise will provide hands-on experience with the C++ Standard Template Library (STL), specifically focusing on the use of `std::vector`, `std::stack`, and `std::queue` to manage collections of objects and implement complex logic for loading and unloading cargo. By the end of this exercise, students will have a deeper understanding of how to design a class hierarchy and use appropriate data structures to manage state and behavior in a multi-component system.

2 Theoretical Introduction to C++ Concepts

2.1 C++ Fundamentals

2.1.1 Basic Data Types, `std::string`, and Enums

C++ has a set of built-in and standard library types for storing different kinds of data.

- `int`: Used for integer numbers (e.g., -5, 0, 100).
- `double`: Used for floating-point numbers (e.g., 3.14, -0.001).
- `char`: Used for single characters (e.g., 'a', 'Z', '!').
- `bool`: Represents boolean values, can only be `true` or `false`.
- `std::string`: A class from the standard library used to store sequences of characters (text). It is more powerful and safer than traditional C-style character arrays. You must include the `<string>` header to use it.
- `enum class` (Scoped Enumeration): A user-defined type that consists of a set of named integer constants called enumerators. Using `enum class` is the modern, preferred way to create enumerations in C++. It improves type safety by preventing enumerators from polluting the global namespace and avoids implicit conversions to integers. This makes the code more readable and robust by replacing "magic numbers" with descriptive names.

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

// Definition of a scoped enum
enum class TrafficLightState {
    Red,
    Yellow,
    Green
};

// This function takes an enum as an argument
void printAction(TrafficLightState state) {
    switch (state) {
        case TrafficLightState::Red:
            std::cout << "Action: Stop!" << std::endl;
            break;
        case TrafficLightState::Yellow:
            std::cout << "Action: Prepare to stop." << std::endl;
            break;
        case TrafficLightState::Green:
            std::cout << "Action: Go!" << std::endl;
            break;
    }
}

int main() {
    // Declare and initialize a variable of the enum type
    TrafficLightState current_state = TrafficLightState::Red;

    printAction(current_state);

    // Change the state
    current_state = TrafficLightState::Green;
    printAction(current_state);

    return 0;
}
/*
Expected Output:
Action: Stop!
Action: Go!
*/
```

2.1.2 Standard Input and Output

C++ uses streams for input and output. The standard library `<iostream>` provides two key objects:

- `std::cout`: The standard output stream, used to print data to the console using the insertion operator `<<`.
- `std::cin`: The standard input stream, used to read data from the console using the extraction operator `>>`.

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

int main() {
    std::string user_name;
    int user_age;

    std::cout << "Please enter your name: ";
    std::cin >> user_name; // Reads a single word

    std::cout << "Please enter your age: ";
    std::cin >> user_age;

    std::cout << "Hello, " << user_name << "! You are " << user_age <<
   " years old." << std::endl;
```

```cpp
    return 0;
}
```

2.1.3 Control Flow: Loops

Loops are used to execute a block of code repeatedly.

- `for` **loop**: Used when you know the number of iterations in advance.
- `while` **loop**: Executes as long as a condition is true. The condition is checked *before* each iteration.
- `do-while` **loop**: Similar to a while loop, but the condition is checked *after* each iteration, meaning the loop body executes at least once.
- **Range-based** `for` **loop (C++11 and later)**: Provides a modern, simplified syntax for iterating over all elements in a container (like a `std::vector` or an array). It is less error-prone and easier to read than traditional index-based loops.

```cpp
#include <iostream>
#include <vector>
#include <string>

int main() {
    // Traditional for loop: prints numbers 0 through 4
    std::cout << "Traditional for loop:" << std::endl;
    for (int i = 0; i < 5; ++i) {
        std::cout << i << " ";
    }
    std::cout << "\n" << std::endl;

    // While loop: prints numbers 0 through 4
    std::cout << "While loop:" << std::endl;
    int j = 0;
    while (j < 5) {
        std::cout << j << " ";
        j++;
    }
    std::cout << "\n" << std::endl;

    // Range-based for loop: iterates over a collection
    std::cout << "Range-based for loop:" << std::endl;
    std::vector<std::string> colors = {"Red", "Green", "Blue"};

    // 'const auto& color' is modern C++ syntax. It means:
    // "for each element in the 'colors' vector, give me a
    // read-only reference (&) to it and call it 'color'".
    // 'auto' automatically deduces the type (std::string).
    for (const auto& color : colors) {
        std::cout << color << " ";
    }
    std::cout << std::endl;

    return 0;
}
/*
```

```
Expected Output:
Traditional for loop:
0 1 2 3 4

While loop:
0 1 2 3 4

Range-based for loop:
Red Green Blue
*/
```
‘‘‘

2.2 STL Data Structures

The C++ Standard Template Library (STL) provides a rich collection of data structures that are essential for efficient programming. These container classes manage storage for a collection of objects and provide member functions to access and manipulate them.

2.2.1 `std::vector`

A `std::vector` is a dynamic array that can grow and shrink in size. It stores elements in contiguous memory locations, which allows for fast random access. It is one of the most commonly used containers in C++.

- **Key Methods:**
  - `push_back(element)`: Adds an element to the end.
  - `pop_back()`: Removes the last element.
  - `size()`: Returns the number of elements.
  - `at(index)` or `operator[]`: Accesses an element at a specific index. `at()` provides bounds checking.
  - `clear()`: Removes all elements.

```cpp
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<std::string> fruits;

    fruits.push_back("Apple");
    fruits.push_back("Banana");
    fruits.push_back("Cherry");

    std::cout << "There are " << fruits.size() << " fruits." << std::
   endl;

    for (int i = 0; i < fruits.size(); ++i) {
        std::cout << "Fruit " << i << ": " << fruits[i] << std::endl;
    }

    fruits.pop_back(); // Remove "Cherry"
    std::cout << "After pop_back, the last fruit is: " << fruits.back()
    << std::endl;
    return 0;
}
/*
Expected Output:
There are 3 fruits.
Fruit 0: Apple
Fruit 1: Banana
Fruit 2: Cherry
After pop_back, the last fruit is: Banana
*/
```

2.2.2 std::stack

A `std::stack` is a container adapter that provides a Last-In, First-Out (LIFO) data structure. It behaves like a real-world stack of plates: you can only add a new plate to the top, and you can only remove the plate that is currently on top.

- **Key Methods:**
  - `push(element)`: Adds an element to the top.
  - `pop()`: Removes the top element.
  - `top()`: Returns a reference to the top element.
  - `empty()`: Returns ‘true‘ if the stack is empty.
  - `size()`: Returns the number of elements.

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> plates;

    plates.push(1); // Plate 1 is at the bottom
    plates.push(2);
    plates.push(3); // Plate 3 is at the top

    std::cout << "Processing plates:" << std::endl;
    while (!plates.empty()) {
        std::cout << "Taking plate: " << plates.top() << std::endl;
        plates.pop();
    }
    return 0;
}
/*
Expected Output:
Processing plates:
Taking plate: 3
Taking plate: 2
Taking plate: 1
*/
```

2.2.3 `std::queue`

A `std::queue` is a container adapter that provides a First-In, First-Out (FIFO) data structure. It works like a real-world queue or line: the first person to get in line is the first one to be served.

- **Key Methods:**
  - `push(element)`: Adds an element to the back of the queue.
  - `pop()`: Removes the element from the front.
  - `front()`: Returns a reference to the front element.
  - `back()`: Returns a reference to the back element.
  - `empty()`: Returns 'true' if the queue is empty.

```cpp
#include <iostream>
#include <queue>
#include <string>

int main() {
    std::queue<std::string> customer_line;

    customer_line.push("Alice");
    customer_line.push("Bob");
    customer_line.push("Charlie");

    std::cout << "Serving customers:" << std::endl;
    while (!customer_line.empty()) {
        std::cout << "Now serving: " << customer_line.front() << std::endl;
        customer_line.pop();
    }
    return 0;
}
/*
Expected Output:
Serving customers:
Now serving: Alice
Now serving: Bob
Now serving: Charlie
*/
```

2.3 Object-Oriented Programming (OOP) in C++

OOP is a programming paradigm based on the concept of "objects," which can contain data (attributes) and code (methods).

2.3.1 Classes and Objects

A **class** is a blueprint for creating objects. An **object** is an instance of a class.

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

class Car {
public: // Access specifier
    // Attributes
    std::string brand;
    int year;

    // Method
    void printInfo() {
        std::cout << "Car: " << brand << ", Year: " << year << std::endl;
    }
};

int main() {
    Car my_car; // Create an object of type Car
    my_car.brand = "Toyota";
    my_car.year = 2021;
    my_car.printInfo();
    return 0;
}
```

2.3.2 Function Overloading

Function overloading allows you to define multiple functions with the same name but with different parameters (either type or number). The compiler automatically chooses the correct one to call based on the arguments provided.

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

void printValue(int val) {
    std::cout << "Integer: " << val << std::endl;
}

void printValue(double val) {
    std::cout << "Double: " << val << std::endl;
}

void printValue(const std::string& val) {
    std::cout << "String: " << val << std::endl;
}

int main() {
    printValue(100);
    printValue(3.14);
    printValue("Hello C++");
    return 0;
}
```

2.3.3 Inheritance and Polymorphism (Virtual Functions)

Inheritance allows a new class (derived class) to inherit properties and methods from an existing class (base class). **Polymorphism** allows objects of different derived classes to be treated as objects of a common base class. This is achieved through **virtual functions**.

```cpp
#include <iostream>
#include <string>
#include <vector>

// Base class
class Animal {
public:
    // virtual function allows derived classes to override it
    virtual void makeSound() const {
        std::cout << "Some generic animal sound" << std::endl;
    }
};

// Derived classes
class Dog : public Animal {
public:
    void makeSound() const override { // 'override' is good practice
        std::cout << "Woof!" << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() const override {
        std::cout << "Meow!" << std::endl;
    }
};

int main() {
    Dog my_dog;
    Cat my_cat;

    // Polymorphism in action
    Animal* animal_ptr = &my_dog;
    animal_ptr->makeSound(); // Calls Dog's version

    animal_ptr = &my_cat;
    animal_ptr->makeSound(); // Calls Cat's version
    return 0;
}
```

2.3.4 Abstraction (Abstract Classes)

Abstraction means hiding complex implementation details and showing only the necessary features. In C++, this is often achieved with **abstract classes**. An abstract class cannot be instantiated and is used as a base class. It is defined by having at least one **pure virtual function**.

```cpp
#include <iostream>

// Abstract base class
class Shape {
public:
    // Pure virtual function (no implementation)
    virtual double getArea() const = 0;
};

// Concrete derived class
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Provide implementation for the pure virtual function
    double getArea() const override {
        return 3.14159 * radius * radius;
    }
};

int main() {
    // Shape my_shape; // ERROR! Cannot create an instance of an
    abstract class.

    Circle my_circle(10.0);
    Shape* shape_ptr = &my_circle; // OK to have a pointer to an
    abstract type

    std::cout << "Area of circle: " << shape_ptr->getArea() << std::
    endl;
    return 0;
}
```

3 Main Task: Port Logistics and Unloading

The main goal is to implement the core classes for a port simulation and the logic for unloading a ship onto trucks based on a set of complex rules.

Task 1: Implement the `Container` Class Hierarchy

Create a class hierarchy to model different types of shipping containers.

- Create an **abstract base class** `Container` with protected attributes for `id` (string) and `weight` (int), and a virtual function `getInfo()` - display all information about the Container instance.
- Derive the following classes from `Container`:
  - `OrdinaryContainer`: Has no extra attributes.
  - `RefrigeratedContainer`: Has an additional attribute `temperature`.
  - `ToxicContainer`: Has an additional attribute `toxicity_level` (use an enum: `Low`, `Medium`, `High`).
  - `FragileContainer`: Has no extra attributes but implies special handling rules.

Task 2: Implement the `Ship` Class

Create a `Ship` class that can hold containers. The ship stores the containers in separate vectors based on their type.

- **Attributes:** `name`, `max_load_weight`, and separate `std::vectors` for each container type (e.g., `std::vector<OrdinaryContainer> ordinary_cargo;`).
- **Methods:**
  - Overloaded `add_container()` methods, one for each container type. Each method should check if adding the container exceeds the ship's `max_load_weight`.
  - A `print_manifest()` method that lists all containers on board, grouped by type.

Task 3: Implement `Truck` and `Port` Unloading Logic

Implement the logic for unloading the ship. The main complexity lies in the `Port` class, which generates an optimized unloading plan.

- `Truck` **Class:** Should have attributes for `max_capacity` (weight) and `max_containers`.
- `Port` **Class:** Should have a method `std::vector<Truck> create_unloading_plan(Ship& ship)`.
- **Unloading Rules:**
  1. **Priority:** `Toxic` containers must be unloaded first, then `Refrigerated`, then `Ordinary` and `Fragile`.

2. **Segregation:** `Toxic` and `Refrigerated` containers must be transported on dedicated trucks (e.g., a ”Toxic Waste” truck type). They cannot be mixed.
3. **Fragile Rule:** `Fragile` containers cannot be on the same truck as any container weighing more than 5000 kg.
4. **Optimization:** The goal is to use the minimum number of trucks. A new truck should only be used when the current one cannot accept the next container in the unloading sequence.

Example Program and Output

```cpp
// main.cpp for Task 1

int main() {
    // 1. Create containers
    OrdinaryContainer ord1("ORD001", 6000);
    RefrigeratedContainer ref1("REF001", 7500, 4.5);
    ToxicContainer tox1("TOX001", 9000, Toxicity::High);
    FragileContainer frg1("FRG001", 3000);
    OrdinaryContainer ord2("ORD002", 4000);

    // 2. Create a ship and load it
    Ship my_ship("Evergreen", 50000);
    my_ship.add_container(ord1);
    my_ship.add_container(ref1);
    my_ship.add_container(tox1);
    my_ship.add_container(frg1);
    my_ship.add_container(ord2);

    std::cout << "--- Ship Manifest ---" << std::endl;
    my_ship.print_manifest();
    std::cout << "---------------------\n" << std::endl;

    // 3. Create a port and generate the unloading plan
    Port port_of_la;
    std::vector<Truck> trucks = port_of_la.create_unloading_plan(
   my_ship);

    // 4. Print the plan
    std::cout << "--- Unloading Plan ---" << std::endl;
    std::cout << "Total trucks needed: " << trucks.size() << std::endl;
    int truck_num = 1;
    for (const auto& truck : trucks) {
        std::cout << "\nTruck #" << truck_num++ << " (" << truck.
   get_type_str() << ")" << std::endl;
        std::cout << "Load: " << truck.get_current_weight() << "kg, "
                  << truck.get_container_count() << " containers." <<
   std::endl;
        truck.print_loaded_ids();
    }
    std::cout << "----------------------" << std::endl;

    return 0;
}
/*
Expected Output:
```

```
--- Ship Manifest ---
Ship: Evergreen, Current Load: 29500/50000 kg
Ordinary Containers:
 - ID: ORD001, Weight: 6000 kg
 - ID: ORD002, Weight: 4000 kg
Refrigerated Containers:
 - ID: REF001, Weight: 7500 kg, Temp: 4.5 C
Toxic Containers:
 - ID: TOX001, Weight: 9000 kg, Toxicity: High
Fragile Containers:
 - ID: FRG001, Weight: 3000 kg
---------------------

--- Unloading Plan ---
Total trucks needed: 3

Truck #1 (ToxicWaste)
Load: 9000kg, 1 containers.
 - Loaded: TOX001

Truck #2 (Refrigerated)
Load: 7500kg, 1 containers.
 - Loaded: REF001

Truck #3 (Standard)
Load: 13000kg, 3 containers.
 - Loaded: ORD001, FRG001, ORD002
----------------------
*/
```

4 Bonus Tasks

Bonus Task 1: Realistic Unloading with a Sorting Yard (`std::stack`)

Improve the simulation by adding a "sorting yard" in the port, which uses stacks to manage containers. This task introduces the LIFO principle for a more realistic sorting process.

- **Refactor Containers:** Remove the `FragileContainer` class. Modify `OrdinaryContainer` to include a `bool is_fragile` flag.
- **Implement Sorting Yard:** In the `Port` class, add a `std::vector<std::stack<OrdinaryContainer>> sorting_yard;`.
- **Update Port Logic:** Modify the `Port`'s processing method.
  1. First, unload `Toxic` and `Refrigerated` containers directly onto their trucks, as before.
  2. Then, unload all `OrdinaryContainers` from the ship into the `sorting_yard`. The rule for stacking is: **a heavier container cannot be placed on top of a lighter one**. If a container cannot be placed on any existing stack, a new stack is created.
  3. Finally, load `Standard` trucks from the `sorting_yard`. The new rule is to always take the **heaviest available container** from the tops of all stacks to optimize truck loading.

Example Program and Output

```cpp
// main.cpp for Bonus Task 1
int main() {
    Ship my_ship("Maersk", 60000);
    my_ship.add_container(OrdinaryContainer("ORD001", 9000, false));
    my_ship.add_container(OrdinaryContainer("ORD002", 4000, false));
    my_ship.add_container(OrdinaryContainer("FRG001", 3000, true)); //
   Fragile
    my_ship.add_container(OrdinaryContainer("ORD003", 9500, false));
    my_ship.add_container(ToxicContainer("TOX001", 12000, Toxicity::
   Medium));

    Port my_port;
    // The port's method now internally handles sorting and then
   loading
    std::vector<Truck> trucks = my_port.process_ship_arrival(my_ship);

    std::cout << "--- Unloading Plan (with Sorting Yard) ---" << std::
   endl;
    // ... (printing logic is the same as in the main task) ...
}
/*
Expected Output:
--- Unloading Plan (with Sorting Yard) ---
Total trucks needed: 3

```

```
Truck #1 (ToxicWaste)
Load: 12000kg, 1 containers.
 - Loaded: TOX001

Truck #2 (Standard)
Load: 18500kg, 2 containers.
 - Loaded: ORD003, ORD001

Truck #3 (Standard)
Load: 7000kg, 2 containers.
 - Loaded: ORD002, FRG001
-------------------------------------------
*/
```

Bonus Task 2: Departure Simulation with a Security Checkpoint (`std::queue`)

Extend the simulation to include a departure terminal where loaded trucks must wait in a queue for a security inspection before they can leave the port. This task introduces the FIFO principle for managing a queue of resources.

- **Update `Truck` Class:** Add a `TruckState` enum (`LOADING`, `WAITING_FOR_INSPECTION`, `DEPARTED`) and an `int inspection_time` attribute. The inspection time should depend on the cargo type (e.g., toxic takes longer).
- **Create `DepartureTerminal` Class:** This new class manages the exit process.
  - **Attributes:** A `std::queue<Truck> waiting_queue`, and a `std::vector<Truck> inspection_lanes` to represent a limited number of inspection spots (e.g., 2).
  - **Methods:**
    - `add_truck_to_queue(Truck t)`: Adds a filled truck to the queue.
    - `simulate_timestep()`: The core simulation logic. In each step, it:
      1. Decrements the remaining inspection time for trucks in the lanes.
      2. Moves finished trucks out of the lanes.
      3. Moves trucks from the front of the queue into any free lanes.
    - `print_status()`: Shows the current state of the queue and lanes.
- **Update Main Program:** After the `Port` fills the trucks, instead of just printing them, add them to the `DepartureTerminal`'s queue. Then, run a simulation loop, calling `simulate_timestep()` and `print_status()` until the terminal is clear.

Example Program and Output

```cpp
// main.cpp for Bonus Task 2
int main() {
    // ... (Setup and loading of ship/port is done first) ...
    // Assume `truck_list` is the vector of trucks filled by the Port

    DepartureTerminal terminal(2); // Terminal with 2 inspection lanes

    for (auto& truck : truck_list) {
        terminal.add_truck_to_queue(truck);
    }

    int time = 0;
    while (!terminal.is_finished()) {
        std::cout << "--- Time: " << time++ << " ---" << std::endl;
        terminal.simulate_timestep();
        terminal.print_status();
        std::cout << std::endl;
    }
    std::cout << "--- All trucks have departed! ---" << std::endl;
}
/*
Expected Output (shortened):
--- Time: 0 ---
Trucks in queue: 1
In inspection: Truck TOX001 (10 min left), Truck REF001 (5 min left)
Departed: 0

--- Time: 1 ---
Trucks in queue: 1
In inspection: Truck TOX001 (9 min left), Truck REF001 (4 min left)
Departed: 0
...
--- Time: 5 ---
Trucks in queue: 0
In inspection: Truck TOX001 (5 min left), Truck STD001 (2 min left)
Departed: 1

...
--- All trucks have departed! ---
*/
```