VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

ADVANCED PROGRAMMING (CO2039)

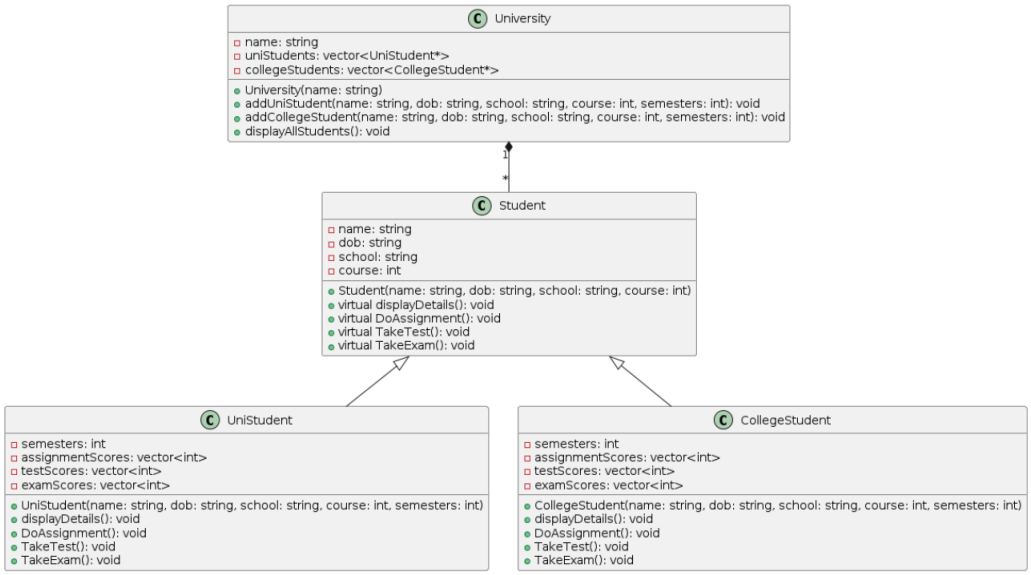
REPORT LAB 3

Student: Vũ Minh Quân - 2212828
Lecturer: PhD. Trương Tuấn Anh

HO CHI MINH CITY, APRIL 2024

Contents

1 Relationship UML Diagram



2 Part 1

I used vector instead of dynamic allocated array and here is the detailed explanation why using vector leads to good improvements:

Memory Management:

Vectors manage memory automatically. They dynamically resize themselves as needed, eliminating manual memory management and the risk of memory leaks.

In your improved code, vectors like `vector<int> assignmentScores;` automatically handle memory allocation and deallocation, simplifying your code and making it less error-prone.

Type Safety and Readability:

Vectors offer type safety, ensuring that you only store elements of the specified type. This reduces the chances of errors caused by storing incorrect data types in your arrays.

Using vectors makes your code more readable and understandable. For example, `vector<int> assignmentScores;` clearly indicates that `assignmentScores` is a vector of integers, improving code comprehension for developers.

Ease of Use and Flexibility:

Vectors provide a wide range of convenient functions and methods for manipulation, such as `push_back()` to add elements, `size()` to get the number of elements, and iterators for traversal.

You can easily iterate over vectors using range-based for loops (`for (const auto student : uniStudents)`) without worrying about array boundaries or manual memory management.

Reduced Code Complexity:

The use of vectors reduces the complexity of your code by eliminating the need for manual memory allocation, deallocation, and pointer arithmetic.

Your code becomes more straightforward and less error-prone, contributing to better maintainability and easier debugging.

Improved Performance:

Vectors are optimized for performance, offering efficient memory allocation strategies and access patterns.

While the performance difference may be negligible for small programs like yours, using vectors ensures that your code is well-optimized and scalable for larger datasets or more complex applications.

3 Part 2

I used singleton instead of normal class division and here is the detailed explanation why using singleton leads to good improvements. The singleton design pattern has been integrated into the C++ student management program to ensure that only one instance of the `University` class exists throughout the program's execution.

Single Instance: The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. In your program, the University class is designed as a Singleton, meaning there can be only one instance of the University object throughout the program's lifetime. This ensures that all parts of your program interact with the same university instance, avoiding duplication and inconsistency.

Global Access Point: By using the Singleton pattern, any part of your program can access the University instance using the static getInstance method. This provides a convenient and centralized way to manage and access the university object, especially in scenarios where multiple parts of the program need to interact with it.

Resource Management: The Singleton pattern helps in managing resources efficiently. In your program, the University class is responsible for managing student objects (UniStudent and CollegeStudent). By ensuring there's only one University instance, resource management becomes centralized. For example, memory allocation and deallocation for student objects are handled within the University class, reducing the complexity of resource management in other parts of the program.

Thread Safety: In a multi-threaded environment, the Singleton pattern can be implemented to provide thread-safe access to the singleton instance. While your current implementation doesn't address thread safety explicitly, the Singleton pattern can be extended to incorporate thread-safe mechanisms like locking to ensure safe concurrent access to the University instance.

Encapsulation and Flexibility: By encapsulating the creation and management of the University instance within the Singleton pattern, you achieve a level of abstraction and flexibility. Changes to how the University instance is created or accessed can be handled within the Singleton pattern implementation, without affecting other parts of the program that use the University class.

Reduced Memory Footprint: In certain scenarios, the Singleton pattern can help in reducing memory footprint by ensuring that only one instance of a class exists. This can be beneficial when dealing with resource-intensive objects or when memory efficiency is a concern.

Improved Code Organization: Using design patterns like Singleton improves the overall organization of your code. It separates concerns related to object creation and ensures a clear distinction between the singleton instance and other classes in your program.

4 Part 3

I used smart pointers, range for loops and auto to improve the code and here is the detailed explanation why using singleton leads to good improvements:

Memory Management: Smart pointers like unique_ptr automatically handle memory deallocation when the pointer goes out of scope, reducing the risk of memory leaks compared to raw pointers. In your program, using unique_ptr ensures that you don't need to manually delete objects, improving code robustness and reducing the chances of forgetting to release memory.

Safety and Readability: Smart pointers provide safety by enforcing ownership semantics. Each object is owned by exactly one unique_ptr, preventing accidental double deletion or memory access issues. The use of make_unique to create objects simplifies object creation and enhances code readability by clearly indicating ownership and resource management.

Reduced Boilerplate Code: Smart pointers eliminate the need for explicit memory management code (e.g., manual new and delete), reducing boilerplate code and making the codebase cleaner and more maintainable. You no longer need to worry about explicitly freeing memory, which reduces the chances of bugs related to memory management.

Efficient Iteration: Range-based for loops (for (const auto student : students)) simplify iteration over containers like vectors by handling iterators internally. This leads to more concise and expressive code compared to traditional loops. Range-based for loops also prevent common iterator-related errors, such as off-by-one errors or iterator invalidation, improving code reliability.

Modern C++ Idioms: Using smart pointers and modern C++ features like range-based for loops aligns your code with modern C++ best practices and idioms, making it easier for other developers to understand and maintain. It also takes advantage of the improvements and safety features introduced in newer versions of the C++ language, promoting code that is more robust and less error-prone.