Module 2: Low-Level APIs, CLR, Value/Reference Types, Optimization

(Tutorial / Lab)

HỌ VÀ TÊN (FULL NAME): NGUYỄN TRỌNG HƯỞNG

MÃ SỐ SINH VIÊN (STUDENT ID): 31231023691

LỚP (CLASS): CS0001

MÔN HỌC (SUBJECTS): Lập trình hệ thống – Systems Programming

CHUYÊN NGÀNH (MINOR): Khoa học máy tính – Computer Science

Lab Objective

Students will:

- Use low-level .NET APIs to interact with memory and data structures
- Understand the performance implications of value vs reference types
- Apply memory and speed optimization strategies
- Explore CLR internals with practical exercises

Question 1 – Understanding Value and Reference Types

1. Create a struct PointStruct with int X, Y
2. Create a class PointClass with the same fields
3. Instantiate both and assign one instance to another
4. Modify the copy and observe changes to the original

Tasks:

- Explain why the changes affect (or do not affect) the original
- Discuss implications for memory and performance

Trả lời Câu 1:

1. Mã nguồn (C# Code)

https://github.com/trongjhuongwr/SystemsProgramming_Labs.git

2. Tasks: Explain why the changes affect (or do not affect) the original

Tại sao Struct không đổi, còn Class lại đổi?

- Với Struct (Value Type):
  + Dữ liệu của ps1 (X=10, Y=20) được lưu trực tiếp trên Stack.
  + Khi lệnh ps2 = ps1 chạy, hệ thống sẽ sao chép từng bit dữ liệu từ ô nhớ của ps1 sang ô nhớ riêng của ps2.
  + Lúc này, ps1 và ps2 là hai thực thể độc lập. Sửa ps2 không liên quan gì đến ps1.
- Với Class (Reference Type):
  + Dữ liệu thực tế của đối tượng (X=10, Y=20) được lưu trên Heap.
  + Biến pc1 trên Stack chỉ chứa địa chỉ (con trỏ - pointer) trỏ tới vùng nhớ đó trên Heap.
  + Khi lệnh pc2 = pc1 chạy, hệ thống chỉ sao chép địa chỉ từ pc1 sang pc2.
  + Kết quả: Cả pc1 và pc2 cùng trỏ vào một đối tượng duy nhất trên Heap. Sửa thông qua pc2 thì pc1 nhìn vào cũng thấy thay đổi đó.

3. Tasks: Discuss implications for memory and performance

A. Về Bộ nhớ (Memory)

- Struct (Stack):
  + Được cấp phát tức thì (chỉ cần di chuyển stack pointer).
  + Tự động hủy khi ra khỏi phạm vi hàm (scope) → Không gây áp lực cho Garbage Collector (GC).
  + Hạn chế: Stack có kích thước nhỏ (thường 1MB), không nên dùng struct cho dữ liệu quá lớn.
- Class (Heap):
  + Phải tìm vùng trống trên Heap để cấp phát → chậm hơn.
  + Tốn thêm bộ nhớ cho phần tiêu đề đối tượng (Object Header, Method Table).
  + Phải chờ GC dọn dẹp → Gây áp lực lên hệ thống nếu tạo quá nhiều object ngắn hạn.

B. Về Hiệu năng (Performance)

- Copying:
  + Struct: Gán a = b sẽ copy toàn bộ giá trị. Nếu struct lớn (ví dụ có 100 biến int), việc copy sẽ rất chậm.
  + Class: Gán a = b chỉ copy địa chỉ (thường 4 hoặc 8 bytes), cực nhanh bất kể object lớn thế nào.
- Truy xuất (Access):
  + Struct: Truy xuất trực tiếp (Direct access) → Nhanh, thân thiện với bộ nhớ cache của CPU (Cache locality).
  + Class: Phải truy xuất gián tiếp qua con trỏ (Dereferencing) → Có thể gây Cache Miss, chậm hơn một chút.
- Kết luận:
  + Dùng Struct cho các đối tượng nhỏ, sống ngắn, dữ liệu đơn giản (như Point, Vector, Color, Date).
  + Dùng Class cho các đối tượng phức tạp, kích thước lớn, cần tính kế thừa và sống lâu dài trong ứng dụng.

Question 2 – Data Structures & Memory Layout

1. Implement a large array of structs vs a large array of classes (same data)
2. Measure memory usage using GC.GetTotalMemory and timing operations on both arrays

Tasks:

- Compare memory footprint
- Compare access speed
- Explain why differences occur in terms of stack vs heap and CLR handling

Trả lời Câu 2:

1. Mã nguồn (C# Code)

https://github.com/trongjhuongwr/SystemsProgramming_Labs.git

2. Tasks Explanation

a.

Tại sao Memory Footprint (Dung lượng bộ nhớ) khác biệt lớn?

- **Struct Array (Mật độ cao):**
    - Mảng struct là một khối bộ nhớ liền mạch duy nhất.
    - Vì mỗi int là 4 bytes, Struct có 2 int = 8 bytes. Tổng cộng ~8MB. Không có "phí phát sinh".
- **Class Array (Chi phí ẩn - Overhead):**
    - Mảng class thực chất là Mảng các con trỏ (Array of References).
    - Bộ nhớ bao gồm:
        1. **Mảng chính:** 1 triệu địa chỉ trỏ tới object (trên máy 64-bit, mỗi địa chỉ là 8 bytes) = 8MB.
        2. **1 triệu Object rời rạc:** Mỗi object PointClass trên Heap không chỉ chứa dữ liệu (8 bytes) mà còn gánh thêm **Object Header** và **Method Table Pointer** (thường tốn thêm 16-24 bytes mỗi object).
    - Kết quả: Ta tốn bộ nhớ gấp 3-4 lần chỉ để lưu cùng một lượng dữ liệu.

b.

Tại sao Access Speed (Tốc độ truy xuất) struct lại nhanh hơn?

- **Struct (Tính cục bộ - Cache Locality):**
    - Dữ liệu nằm sát nhau. Khi CPU đọc phần tử i, nó tự động tải luôn i+1, i+2... vào bộ nhớ đệm (L1/L2 Cache).
    - Việc truy xuất diễn ra tuần tự, cực nhanh.
- **Class (Truy vết con trỏ - Pointer Chasing):**
    - Các object PointClass nằm rải rác lộn xộn khắp nơi trên Heap.

- Để truy cập classArray[i], CPU phải: Đọc địa chỉ từ mảng → Nhảy đến địa chỉ đó trên Heap → Lấy dữ liệu.
- Việc "nhảy cóc" này làm CPU không tận dụng được Cache, gây ra hiện tượng Cache Miss (trượt bộ nhớ đệm), làm chậm chương trình đáng kể.

c.

Vai trò của CLR và Stack/Heap

- **Phân bổ (Allocation):**
  - Struct Array: CLR chỉ cần cấp phát 1 lần duy nhất một khối lớn trên Heap (Lưu ý: Mảng struct vẫn nằm trên Heap vì Array là reference type, nhưng dữ liệu bên trong nó được xếp chồng lên nhau - packed).
  - Class Array: CLR phải thực hiện cấp phát 1.000.001 lần (1 lần cho mảng + 1 triệu lần cho từng object). Việc tìm chỗ trống trên Heap triệu lần rất tốn thời gian.
- **Áp lực lên Garbage Collector (GC Pressure):**
  - Khi hủy mảng Struct: GC chỉ cần dọn 1 object.
  - Khi hủy mảng Class: GC phải quét và dọn 1.000.001 object. Điều này có thể làm đứng ứng dụng (Freezing) trong vài giây.

Tóm lại: Sử dụng Struct cho danh sách dữ liệu lớn (như tọa độ đồ họa, xử lý tín hiệu) là kỹ thuật tối ưu hóa quan trọng nhất trong Lập trình hệ thống trên .NET.

Question 3 – Just-In-Time (JIT) Compilation

1. Describe how the Just-In-Time (JIT) compiler works in .NET
2. Explain when JIT compilation occurs during program execution

Tasks:

- Illustrate with a small code snippet how IL is converted to native code at runtime

- Discuss performance implications of JIT compilation

Trả lời Câu 3:

1. Cơ chế hoạt động của JIT (Just-In-Time) Compiler

Trong .NET, mã nguồn C# không được dịch thẳng ra mã máy (Native Code) ngay lập tức. Quy trình diễn ra như sau:

1. **Biên dịch lần 1 (Compilation):** Trình biên dịch C# (Roslyn) dịch mã nguồn (.cs) thành Ngôn ngữ trung gian (Microsoft Intermediate Language - MSIL hoặc IL). Mã IL này được lưu trong file .dll hoặc .exe.
2. **Biên dịch lần 2 (JIT Compilation):** Khi chương trình chạy, CLR (Common Language Runtime) sẽ kích hoạt JIT Compiler.
3. **Dịch sang mã máy:** JIT đọc mã IL và dịch nó sang mã máy (Native Code: x86 hoặc x64) tương thích với cấu trúc phần cứng hiện tại của máy tính.
4. **Thực thi:** CPU thực thi mã máy này.

2. Khi nào JIT diễn ra? (When JIT compilation occurs)

Đúng như tên gọi "Just-In-Time" (Vừa kịp lúc), JIT hoạt động theo cơ chế On-Demand (Theo nhu cầu):

- JIT KHÔNG dịch toàn bộ chương trình ngay khi bật lên (điều này sẽ làm khởi động rất chậm).
- JIT chỉ dịch từng phương thức (method) tại thời điểm phương thức đó được gọi lần đầu tiên.
- **Caching:** Sau khi dịch xong, mã máy sẽ được lưu vào bộ nhớ đệm (Memory Cache). Các lần gọi hàm tiếp theo sẽ dùng lại mã máy đã dịch, không cần JIT can thiệp nữa -> Tốc độ cực nhanh.

3. Minh họa chuyển đổi từ IL sang Native Code (Code Snippet Illustration)

Hãy xem một ví dụ đơn giản với hàm cộng hai số để thấy sự biến đổi:

A. Mã nguồn C# (High-Level)

```
int Add(int a, int b)
{
   return a + b;
}
```

B. Mã IL (Intermediate Language)

Đây là những gì trình biên dịch C# tạo ra. Nó hoạt động dựa trên ngăn xếp (Stack-based).

```
// Code IL (giả lập)
ldarg.0     // Đẩy tham số thứ 1 (a) vào stack
ldarg.1     // Đẩy tham số thứ 2 (b) vào stack
add         // Lấy 2 số từ stack ra, cộng lại, đẩy kết quả vào stack
ret         // Trả về giá trị trên đỉnh stack
```

C. Mã máy (Native Assembly - x64)

Đây là những gì JIT tạo ra tại Runtime để CPU hiểu. Nó làm việc trực tiếp với thanh ghi (Registers).

```
// Code Assembly (x64)
lea eax, [rcx+rdx]  // Cộng giá trị thanh ghi RCX (a) và RDX (b), lưu vào EAX
ret                 // Trả về
```

Nhận xét: Ta có thể thấy mã IL rất chung chung (không quan tâm CPU là Intel hay AMD), còn mã Native thì phụ thuộc chặt chẽ vào kiến trúc phần cứng.

4. Thảo luận về Hiệu năng (Performance Implications)

Việc sử dụng JIT mang lại hai mặt vấn đề về hiệu năng mà lập trình viên hệ thống cần nắm:

Ưu điểm (Pros):

1. **Tối ưu hóa động (Runtime Optimization):** Vì JIT dịch mã ngay trên máy người dùng, nó biết chính xác CPU đó có hỗ trợ các tập lệnh cao cấp (như AVX2, SSE4)

hay không để sinh mã tối ưu nhất. Trình biên dịch tĩnh (AOT) khó làm được điều này.

2. **Profile-Guided Optimization (PGO):** JIT hiện đại có thể theo dõi cách code chạy thực tế để tái biên dịch (Re-JIT) những đoạn code nóng (hot path) cho hiệu năng cao hơn nữa (Tiered Compilation).

Nhược điểm (Cons):

1. **Thời gian khởi động (Startup Latency):** Chương trình .NET thường khởi động chậm hơn C++ một chút vì tốn thời gian "Warm-up" (chờ JIT dịch các hàm cơ bản lúc đầu).
2. **Tốn bộ nhớ (Memory Overhead):** Hệ thống phải tốn RAM để chứa cả mã IL gốc, trình biên dịch JIT, và mã Native mới sinh ra.

Question 4 – Compiler vs Interpreter

1. Compare a compiler and an interpreter
2. Give one example of each in the context of programming languages

Tasks:

- Discuss advantages and disadvantages of each approach
- Explain how .NET uses a hybrid approach with IL and JIT compilation

Trả lời Câu 4:

1. So sánh Compiler và Interpreter (Compare a compiler and an interpreter)

Chúng ta có thể so sánh hai mô hình này dựa trên cách chúng xử lý mã nguồn:

Tiêu chí	Trình biên dịch (Compiler)	Trình thông dịch (Interpreter)
Cơ chế hoạt động	Dịch toàn bộ mã nguồn sang mã máy (Native Code) tạo thành file thực thi (.exe) trước khi chạy.	Dịch và thực thi từng dòng lệnh (line-by-line) ngay tại thời điểm chạy (Runtime).
Thời điểm phát hiện lỗi	Phát hiện lỗi cú pháp (Syntax Error) trước khi chạy (Compile-time).	Chỉ phát hiện lỗi khi chạy đến dòng lệnh đó (Runtime).
Hiệu năng	Tốc độ thực thi rất nhanh vì mã đã được tối ưu hóa sẵn cho phần cứng.	Chậm hơn vì vừa phải dịch vừa phải chạy, và thiếu các tối ưu hóa sâu.
Tính di động (Portability)	Thấp. Muốn chạy trên OS khác (ví dụ từ Windows sang Linux), phải biên dịch lại (Re-compile).	Cao. Chỉ cần máy đó có cài Interpreter là chạy được source code ngay.

2. Ví dụ (Examples)

- **Compiler:** C++, C, Go, Rust. (Code xong phải Build ra file .exe mới chạy được).
- **Interpreter:** Python, JavaScript, PHP, Ruby. (Viết xong chạy ngay script .py hay .js).

3. Thảo luận Ưu và Nhược điểm (Tasks: Advantages and Disadvantages)

A. Compiler (Biên dịch)

- **Ưu điểm:** Hiệu năng cao nhất (Best performance), bảo mật mã nguồn tốt hơn (vì chỉ phân phối file binary).
- **Nhược điểm:** Quá trình phát triển chậm do phải chờ Compile liên tục. Khó chạy đa nền tảng.

B. Interpreter (Thông dịch)

- **Ưu điểm:** Phát triển nhanh (Rapid Development), dễ dàng gỡ lỗi (Debug), chạy đa nền tảng dễ dàng.
- **Nhược điểm:** Tốc độ chậm, mã nguồn công khai (dễ bị xem trộm logic).

4. Giải thích cách tiếp cận lai của .NET (Hybrid approach with IL and JIT)

Đây là phần quan trọng nhất để trả lời câu hỏi *"Explain how .NET uses a hybrid approach"*. .NET không chọn phe nào, mà nó lấy ưu điểm của cả hai:

1. **Giai đoạn 1 (Giống Compiler):**
    - Mã nguồn C# được biên dịch ra IL (Intermediate Language).
    - *Lợi ích:* Giúp bắt lỗi cú pháp ngay từ đầu và tạo ra file .dll gọn nhẹ, độc lập phần cứng.
2. **Giai đoạn 2 (Giống Interpreter nhưng nhanh hơn):**
    - Khi chạy, JIT Compiler sẽ dịch IL sang Mã máy (Native Code).
    - *Khác biệt:* Thay vì dịch từng dòng rồi vứt đi như Interpreter truyền thống, JIT dịch xong sẽ lưu lại (Cache) kết quả.
    - *Lợi ích:* Lần đầu chạy hàm sẽ hơi chậm (giống Interpreter), nhưng từ lần thứ 2 trở đi sẽ chạy nhanh như C++ (giống Compiler).

Kết luận: .NET sử dụng mô hình lai để đạt được sự cân bằng: "Viết code linh hoạt như Java/Python nhưng chạy nhanh gần bằng C++".

Question 5 – Membership Card Printing Program

1. Write a C# program that prints a membership card for a CRM system
2. Ensure the program automatically installs required fonts on the computer if they are not present

Tasks:

- Demonstrate the program printing a sample card with name, ID, and membership level
- Explain how your program handles font installation and system dependencies
- Discuss any potential security or permission issues when installing fonts programmatically

Trả lời Câu 5:

1. Mã nguồn (C# Code)

https://github.com/trongjhuongwr/SystemsProgramming_Labs.git

2. Tasks: Explain

Xử lý phụ thuộc và Cài đặt Font (Handling Dependencies)

Chương trình sử dụng hai tầng xử lý:

1. **Tầng Hệ thống (Low-Level API):**
    - Sử dụng hàm AddFontResource từ gdi32.dll. Đây là một API của Windows (Unmanaged Code).
    - Hàm này thông báo cho Windows biết: "Có một file font ở đường dẫn này, hãy thêm nó vào bảng Font của hệ thống ngay lập tức".
    - Cách này giúp ứng dụng chạy được font mà không cần copy file vào thư mục C:\Windows\Fonts. Khi khởi động lại máy, font này sẽ tự mất (tránh rác hệ thống).
2. **Tầng Ứng dụng (.NET):**
    - Sử dụng PrivateFontCollection để quản lý font nội bộ. Đây là cách an toàn để đảm bảo dù API hệ thống thất bại, chương trình vẫn có thể cố gắng load font cho riêng nó sử dụng mà không ảnh hưởng các phần mềm khác.

3. Tasks: Security & Permission Issues

a. Vấn đề Quyền quản trị (Admin Privileges/UAC)

- **Cách cài đặt vĩnh viễn:** Nếu ta muốn cài font vĩnh viễn, ta phải copy file vào C:\Windows\Fonts và ghi khóa vào Registry (HKEY_LOCAL_MACHINE). Hành động này bắt buộc phải có quyền Administrator.
- **Rủi ro:** Nếu ứng dụng của chúng ta chạy ở máy người dùng bình thường (User Mode), chương trình sẽ bị Crash ngay lập tức với lỗi UnauthorizedAccessException.
- **Giải pháp:** Sử dụng AddFontResource (như code trên) hoặc load font từ Memory thường không yêu cầu quyền Admin cao nhất, thân thiện với người dùng hơn.

b. Rủi ro bảo mật từ Font (Font Parsing Vulnerabilities)

- Trong quá khứ, các file Font (TTF, OTF) là vector tấn công ưa thích của Hacker. Hệ điều hành phải phân tích (parse) cấu trúc phức tạp của file font ở cấp độ hạt nhân (Kernel mode).
- Nếu hacker chèn mã độc vào file font và chương trình tự động cài đặt nó, hacker có thể leo thang đặc quyền (Privilege Escalation) chiếm quyền kiểm soát máy tính.
- Bài học: Không bao giờ tự động tải và cài đặt font từ Internet mà không kiểm tra chữ ký số hoặc nguồn gốc.