Programming Language

Assignment #6

- Copy 시 소스 제공자와 카피 당사자 공히 **해당 과제 전체 0점 처리.**
- 반드시 자기 자신의 생각과 글로 작성할 것. 다른 곳에서 자료 인용 시 출처 반드시 기재 **(미기재 시 50% 감점)**
- 한글/영문 상관 없음.

11.2. Suppose someone designed a stack abstract data type in which the function top returned an access path (or pointer) rather than returning a copy of the top element. This is not a true data abstraction. Why? Give an example that illustrates the problem.

제대로 된 추상 데이터 타입을 설계하기 위해서는 추상 데이터 타입의 표현이 그 타입을 사용하는 프로그램 단위(클라이언트)로부터 숨겨져야 한다. 이 설계 특징을 "정보 은폐"라고 부르는데, 이는 클라이언트가 객체를 원래 설계된 바(제공되는 연산들)와 다른 방식으로 임의 조작하는 것을 막기 때문에, 객체의 **무결성**을 증가시키고, 이러한 무결성은 곧 코드의 **신뢰도** 증가로 이어지게 된다. 또한 정보 은폐는 프로그래머가 해당 객체를 다룰 때 이해하고 신경 써야 하는 변수와 코드 영역을 감소시키기 때문에 프로그램의 **구조를 단순하게** 만드는 역할을 한다.

문제와 같이 스택 추상 데이터 타입의 최상단 원소를 복사하여 반환하지 않고, 포인터로 반환하는 상황이라면 다음과 같이 표현할 수 있을 것이다.

```
Stack<int> stack{100};
// Stack size를 100으로 설정한 스택 추상 데이터 타입, 내부적으로 동적 할당 배열로 스택을 구현한다.
int * top_element = stack.top();
// 포인터로 최상단 원소를 반환하는 구조의 스택 추상 데이터 타입
*(top_element + 1) = 777
// Case 1
free(top_element);
// Case 2
```

만약 최상단 원소를 복사하여 반환한다면, 클라이언트 부분의 코드에서 할 수 있는 것은 단지 스택 추상 데이터 타입을 내부에서 어떤 처리가 이루어지는지 모

르는 일종의 블랙박스(내부 구조나 작동원리를 모르는 소프트웨어)로 간주하고 최상단 원소의 값을 사용할 수만 있는 반면에, 포인터를 반환하는 상황이라면 해당 스택 추상 데이터 타입이 저장되는 주소를 알 수 있다. 이는 사용자가 내부 구조를 수정할 수 있게 하는 길을 열어준 것이나 다름이 없는데, 가령 고수준의 사용자는 공간적 지역성을 고려한 배열 구조로 내부 구현이 작동한다는 것을 눈치채고 가져온 주소 인근에 있는 요소에 접근하거나, 값을 변경할 수 있고 (Case 1), 심지어 동적할당 된 메모리를 할당해제 할 수도(Case 2) 있을 것이다. 이렇게 되면 당연히 스택 객체는 예상된 동작과 다른 동작을 할 수 있는 여지가 생기고 무결성이 깨지게 된다. 이런 무결성의 파괴는 해당 설계 코드의 신뢰도를 감소시킨다. 그리고 이런 코드들은 프로그램 전체 구조의 복잡성을 증가시켜서 코드를 이해하기 더 난해하게 만들 것이다.

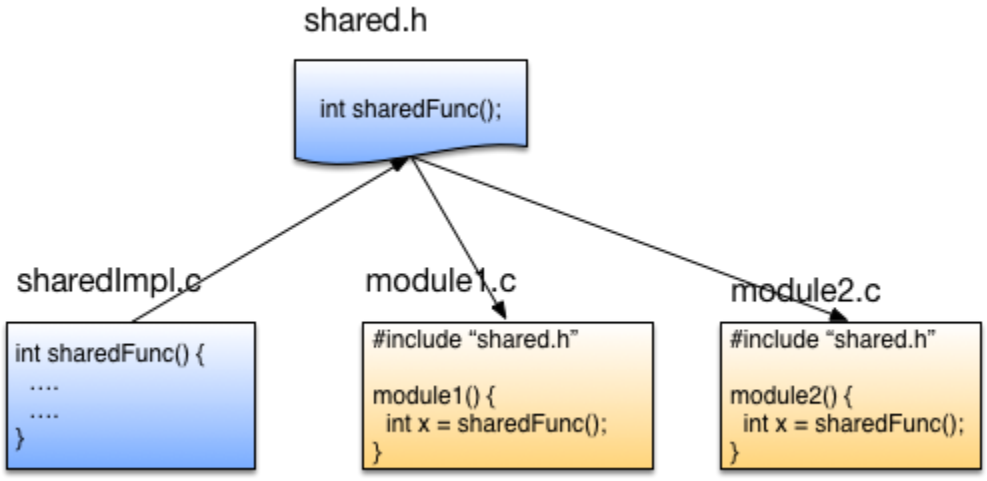
11.5. Why must the structure of nonpointer abstract data types be given in Ada package specifications?

Ada 패키지의 캡슐화 구조를 이용하여, 추상 데이터 유형에서 노출되면 안되는 구조 및 표현이 올바르게 "정보 은닉"되기 위해서이다. 이를 만족하는 추상 데이터 타입을 만들기 위해서 Ada에서는 포인터와 몸체 패키지(명세의 실제 구현을 제공하는 패키지) 활용하여 패키지 구조를 설계하거나, 본 문제에서와 같이 몸체 패키지와 포인터를 활용하지 않는 패키지 구조를 설계할 수 있는데, 이 경우 명세 패키지(캡슐화된 인터페이스를 제공하는 패키지)의 전용 부분(private part)에 타입의 표현을 나타내고, 타입에 대한 선언을 가시적인 부분에 나타낸다. 이를 Ada 코드로 표현하면 다음과 같다.

```
package Stack is
   type Stack_Structure_Type is private
   -- 클라이언트에서 가시적으로 접근 가능하게 가시적인 절에 선언
   -- 배정, 동등 비교 및 비동등 비교를 위한 내장 연산만 가능하게 전용 타입으로 선언
   private
      type Stack_Structure_Type_Inner;
   -- 내용을 숨기는 절(전용 절)에 실제 타입을 표현
   type Stack_Structure_Type is access Stack_Structure_Type_Inner;
   -- Stack_Structure_Type를 Stack_Structure_Type_Inner의 접근 타입(access type)으로 선언
end Stack;
```

11.8. Why didn't C++ eliminate the problems discussed in Problem 7(Explain the danger of C's approach to encapsulation.)?

C 언어에서 발생하는 캡슐화 문제는 언어적인 지원을 통해 코드 구조에 대해 은닉이나 보호가 이루어 지는 것이 아니라, 헤더 파일과, 구현 파일의 분리를 통해서만 이루어진다는 점에서 기인한다. 따라서 라이브러리 구현자가 구현 코드에서 노출하고 싶은 함수나 변수만을 헤더에 기재했더라도, 라이브러리의 사용자가 누락된 함수, 변수만 헤더에 넣기만 하면(디컴파일 또는 리버스 엔지니어링을 통해서 함수 구조를 파악할 수 있다) 구현 코드의 해당 부분을 그대로 사용할 수 있게 되는 문제를 가진다.



[C언어의 헤더-구현 파일 구조]

C++은 클래스 구조를 추가하면서 접근 제한자(private, public)와 프렌즈 키워드(클래스의 private 요소에 대한 접근 권한을 획득할 수 있게 함)를 통해 언어적인 수준의 캡슐화를 지원한다. 하지만 C++은 C의 하위 호환성을 지원하는 개발 철학을 가지고 있기 때문에, 클래스 단위로 이루어지지 않은 다른(C와 비슷한) 코드들에 대해서는 여전히 C와 동일한 캡슐화 문제를 가지고 있게 되었다.

11.11. What are the arguments for and against the Objective-C design that method access cannot be restricted?

찬성(함수 접근을 제한하지 않아도 된다):

함수 접근을 제한하지 않으면, 언어 사용자가 추가적인 접근 제한 문법을 익힐 필요가 없는 면에서 언어 사용자의 부담이 줄어들 수 있고, 접근 제한 없이 함수를 호출할 수 있으므로 함수 호출에 대한 유연성을 가질 수 있다. 또한 함수 접근 제한자에 대한 구문 트리 처리와 컴파일 시점의 접근 제한된 함수 호출 오류 검출과 같은 복잡한 컴파일러 설계 비용을 줄일 수 있다.

반대(함수 접근을 제한해야한다)

함수 접근을 막을 수 있는 방법이 언어단에서 없으면 캡슐화를 제대로 지원할 수 없고, 이는 언어에서 사용하는 클래스 객체의 무결성의 저하, 신뢰성의 하락을 가져온다. 또한 의도치 않은 함수 호출의 가능으로 인해 클래스 설계 의도에 반하는 코드들이 언어 사용자에 의해서 쓰여질 수 있고, 이런 코드들은 원래 설계 의도의 숙지와 더불어 추가적인 의미 파악이 필요하므로 언어의 단순성에 저해를 가져온다. 또한 자바(Reflection으로 런타임에서 동적 함수 접근 제한자 설정 지원), Ruby(동적 함수 접근 제한자 설정 지원)와 같은 언어들에서처럼 동적 함수 접근 제한 기능을 구현한 경우, 언어의 함수 호출에 대한 유연성을 가짐과 동시에(추가적인 접근 제한자 조작이 필요하지만), 캡슐화가 필요한 경우에는 사용할 수 있게 절충하여 함수 접근 제한을 구현할 수도 있다.

11.14. Describe a situation where a C# struct is preferable to a C# class.

C#의 구조체와 클래스는 멤버 변수와 함수를 가지고 있을 수 있다는 점에서 서로 유사하지만 **두가지의 큰 차이점**이 존재한다.

1. 클래스 객체는 힙에 할당되지만, 구조체 객체는 스택에 할당된다(멤버 변수의 크기 총합이 16 바이트 내 인 경우).
2. 구조체는 상속을 할 수 없다.

따라서 상속 구조를 사용하거나, 큰 데이터를 다루어야 하는 경우에는 클래스를 사용하는 것이 좋고, **상속을 사용하지 않고 작은 데이터를 다뤄야 하는 경우에는 구조체를 사용하는 것이 좋다.** 왜냐하면 힙에 메모리를 할당하고 해제하는 과정은 자

동으로 쓰레기 수집(힙에 할당한 객체에 대한 모든 참조가 없어지면 메모리 회수)이 이루어지는 C#의 특성상 스택을 사용하는 것보다 오버헤드가 훨씬 크고 속도도 느리기 때문이다.

11.17. The namespace of the C# standard library, System, is not implicitly available to C# programs. Do you think this is a good idea? Defend your answer.

사실 이 문제는 현 시점에서 논의하자면 약간의 오류가 있다. 작년인 2021년에 출시된 .NET 6, C# 10버전에서는 "implicit using"이라는 암시적 네임스페이스 사용 방법이 추가었다. 이 기능 덕분에 현 시점에서 엄밀히 말하면 C#에서도 "global using global::System; "과 같은 구문을 통해 **System 네임스페이스를 암시적으로 사용할 수 있다.**

이런 현 상황과 별개로 System 네임스페이스를 암시적으로 사용하는 것에 대한 내 의견은, 무조건 System 네임스페이스가 그냥 무조건 암시적으로 사용할 수 있는 것이라면 내가 사용할 수 있는 이름 공간에 제약이 생기는 것이므로 별로 좋지 않다고 생각한다. 하지만 이를 "implicit using" 기능과 같이 **사용자가 원하는 네임스페이스를 선택해서** 암시적으로 사용할 수 있는 형태의 구현이라면, 이 기능(암시적 사용)이 없는 것 보다 있는 것이 훨씬 낫다고 생각한다.

Reference

11장 추상 데이터 타입과 캡슐화 구조 - 프로그래밍 언어론 제 10판 (Robert W. Sebesta)

AdaCore - Intro to ada: Privacy

Wikibooks – Ada Programming/Types/Access

C# 10 Implicit Using and File-Scoped Namespaces

<The End of the Assignment>