2020년도 2학기 컴퓨터공학설계및실험Ⅰ

4주차 기초 C++ 프로그래밍 CPP-2 : LinkedList/Stack 예비보고서

20161663 허재성

1. 실습 목적

객체 지향 프로그래밍은 상속과 더불어, 다형성(Polymorphism)을 제공함으로써 프로그래밍의 효율성과 유연성 및 재사용성을 달성한다. C++에서 제공하는 몇몇 방법들을 통해 객체 지향 프로그래밍의 이점을 살펴볼 수 있다. 이번 실험에서는 상속과 더불어 교재에서 언급한 다형성의 예를 살펴봄으로써 효율적인 프로그래밍을 위한 기술을 익힌다.

2. 관련 이론

캡슐화(Encapsulation) : 객체 지향 프로그래밍(OOP)에서의 프로그래밍 패러다임으로 객체의 속성, 데이터에 해당하는 멤버 변수와 기능, 행동에 해당하는 멤버 함수를 하나로 묶고 그 실제 구현 내용의 일부를 외부로부터 감추어 은닉하는 것이다.[1] 보통 데이터에 해당하는 멤버 변수를 은닉하여 클래스 외부에서 직접 접근하지 못하도록 한다. 데이터 및 기능을 외부로부터 감추기 위해 프로그래밍 언어적 측면에서 접근 지정자를 두어 은닉의 정도를 구현한다. C++의 경우 접근 지정자로 클래스 외부에서의 접근을 막는 private, 외부에서의 접근을 허용하는 public, 클래스와 해당 클래스를 상속하는 파생 클래스에서의 접근을 허용하는 protected가 있다.

재정의(Overriding) : 기반 클래스를 상속하는 파생 클래스가 있을 때, 기반 클래스의 멤버 함수를 파생 클래스가 상속받은 그대로 사용하는 것이 아니라 파생 클래스에 맞게 새로 작성하는 것을 의미한다. 기반 클래스에 존재하는 멤버 함수와 이름, 매개변수의 자료형 및 개수, 반환형까지 완전히 같은 함수를 파생 클래스에 만드는 것을 재정의라고 한다. 이것은 에러가 아니며 실행 후에 파생 클래스의 이름으로 해당 멤버 함수를 호출하면 파생 클래스에서 새로 작성한 함수가 호출된다.[2] 기반 클래스로부터 상속 받은 멤버 함수의 내용을 그대로 사용하는 것이 아니라 파생 클래스에 맞게 변경하는 것을 재정의라고 한다. 다중 정의(Overloading)와 다른 점은 다중 정의는 이름만 같고 매개 변수가 다른 함수를 새로 추가하는 것이고, 재정의(Overriding)은 기반 클래스로부터 상속받은 멤버 함수의 내용을 변경하는 것이다.[3]

다형성(Polymorphism) : 객체지향개념에서 다형성이란 ‘여러 가지 형태를 가질 수 있는 능력’을 의미한다.[4] 하나의 인터페이스를 사용하여 여러 형태의 데이터 타입(또는 함수)들을 사용할 수 있게 하는 특징을 의미한다. 다형성에는 파라미터적 다형성(Parametric Polymorphism), 서브타입 다형성(Subtype Polymorphism), 그리고 엄밀한 의미의 다형성이 아니지만 같이 언급되기도 하는 애드혹 다형성(Ad-hoc Polymorphism)이 있다. 파라미터적 다형성은 함수나 클래스를 정의할 때 같은 기능을 하지만 자료형만 다른 경우, 모든 코드를 다시 작성해야 하는 문제를 해결하는 다형성으로 C++에서는 템플릿(Template, 틀)을 이용하여 파라미터적 다형성을 구현한다. 애드혹 다형성은 하나의 개체가 여러 특성을 가지는 일반적인 다형성과 달리 여러 개체가 하나의 개체처럼 행동한다는 의미로 함수의 다중 정의를 통해 미리 여러 함수를 작성해 놓고 인자를 다르게 호출하여 그에 맞는 함수가 호출되는 방식이 그 예이다. 함수와 연산자 다중 정의가 애드혹 다형성을 구현한 것에 해당한다. 객체지향 프로그래밍에서 흔히 이야기 하는 다형성은 서브타입 다형성으로 대체가능성을 의미한다. 어떤 자료형 T의 객체를 요구하는 상황에서 자료형이 T인 객체뿐만 아니라 T의 서브타입을 가지는 객체도 대신 사용할 수 있는 것이 서브타입 다형성이다. A라는 기반 클래스와 A를 상속하는 파생 클래스 B, C가 있으면 A의 객체를 요구하는 상황에서 B, C의 객체를 대신 사용할 수도 있다. C++에서는 A 포인터형 변수가 A 객체 뿐만 아니라 B, C 객체도 가리킬 수 있는 것으로 구현된다.[5]

3. 실습 방법

연결 리스트 자료구조를 제공하는 LinkedList 클래스를 완성하고 LinkedList 클래스를 기반클래스로 하여 상속과 재정의, 다형성을 이용하여 LinkedList 클래스를 상속하는 파생 클래스 Stack을 작성한다. Stack 자료 구조의 특성을 이용하여 LIFO(Last In First Out)을 구현한다.

관련 이론에서 알아본 함수의 재정의와 다형성을 이용하여 LinkedList/Stack 문제를 해결한다. 교재에 구현되어 있는 LinkedList 클래스를 기반 클래스로 하고 Stack 클래스를 LinkedList 클래스를 상속하는 파생 클래스로 한다.

LinkedList 클래스에서 구현되지 않은 멤버 함수 Print를 구현하여 LinkedList 클래스의 구현을 마친다. Stack 클래스를 따로 정의하는 것이 아니라 LinkedList를 상속하여 LinkedList 클래스의 파생 클래스 Stack 클래스를 작성한다. 상속을 통해 LinkedList와 Stack 클래스의 공통된 멤버 변수(LinkedList/Stack의 원소 역할을 하는 Node 객체를 가리키는 포인터 first, 저장된 원소의 개수를 나타내는 current\_size), 공통된 멤버 함수(current\_size를 반환하는 GetSize, 맨 앞에 원소를 삽입하는 Insert, 모든 원소를 출력하는 Print 함수)를 Stack 클래스에 따로 작성할 필요가 없어 LinkedList 클래스에 작성한 코드를 재사용할 수 있다. 이로 인해 코드 작성 시간을 절약하고 필요 시 코드의 변경, 수정을 손쉽게 할 수 있다.

LikedList 자료 구조와 Stack 자료구조는 원소를 제거하는 데에서 차이가 있다. 문제에서 정의한 LinkedList는 Queue처럼 FIFO(First In First Out)의 특성을 가지고 있고 Stack은 LIFO 특성을 가지고 있다. 따라서 Stack 클래스에서는 LinkedList의 Delete 함수를 상속하여 그대로 사용할 수 없고 상속받은 Delete 함수를 Stack 클래스에 맞게 재정의(Overriding)할 필요가 있다.

또한 다형성을 구현하기 위해 LinkedList형 포인터 변수 p를 선언한다. 사용자가 요구하는 자료구조에 따라 LinkedList 객체 또는 Stack 객체를 동적 할당하여 p가 가리키게 한다. 서브타입 다형성에 의해 LinkedList 클래스의 포인터 변수 p는 LinkedList 클래스의 인스턴스 또는 LinkedList 클래스의 파생 클래스의 인스턴스를 가리킬 수 있다.

일반적인 멤버 함수는 클래스 포인터 변수의 정적 타입을 따른다. LinkedList 포인터 변수 p로 멤버 함수를 호출하면 포인터 변수 p가 가리키는 인스턴스가 LinkedList 객체가 아니라 파생 클래스 Stack의 객체일지라도 Stack 클래스에 재정의된 멤버 함수가 아니라 LinkedList에 정의된 함수를 호출한다. 하나의 클래스 포인터 변수 p가 LinkedList 자료구조와 Stack 자료구조의 모든 형태를 가질 수 있도록 Delete 함수가 동적 타입을 따르도록 해야 한다. 이때 사용하는 키워드가 virtual로 함수에 virtual를 선언하여 가상 함수로 만들면 해당 함수는 포인터 형이 아니라 포인터가 실제로 가리키는 인스턴스의 형에 따라 호출된다. Delete 함수를 제외한 모든 멤버 함수는 LinkedList의 함수를 그대로 사용하면 되므로 가상 함수가 아니라 일반 함수로 선언하고, Delete 함수는 LinkedList와 Stack이 서로 다른 역할을 하므로 가상함수로 선언하여 포인터 p가 실제로 가리키는 인스턴스의 형에 따라 LinkedList에 정의된 Delete 함수 또는 Stack에 재정의된 Delete 함수가 호출되도록 한다.

마지막으로 파라미터적 다형성을 구현한다. 파라미터적 다형성을 구현하지 않으면 LinkedList와 Stack의 노드에 저장할 데이터의 자료형에 따라 LinkedList, Stack 클래스를 따로 작성해줘야 한다. 같은 기능에 자료형만 다른 클래스를 자료형에 따라 일일이 작성할 필요 없이 파라미터적 다형을 구현하여 하나만 작성하면 된다. template 키워드를 이용하여 파라미터적 다형성을 구현한다. 클래스의 선언부 위와 멤버 함수의 정의부 위에 template <class T>를 넣어 주고 클래스와 함수 부분에서 자료형에 따라 바뀌는 부분을 T로 바꿔주면 된다.

주의 사항으로 A 클래스를 선언하고 정의할 때 일반적으로 클래스의 선언(멤버 변수, 멤버 함수의 선언을 포함)은 헤더 파일 A.h에, 멤버 함수의 구현은 소스 파일 A.cpp에 작성하지만 템플릿을 사용하여 A 클래스를 템플릿 클래스로 작성할 경우 선언과 구현을 A.h 파일에 모두 기술해야 한다. 템플릿 함수의 경우에도 마찬가지다.

4. 기타

파라미터적 다형성을 구현할 때 템플릿에 디폴트로 자료형을 설정해 줄 수 있다. 예를 들어 template <class T = int>로 설정해 주면 해당 클래스의 객체를 생성하거나 함수를 호출할 때 따로 자료형을 명시하지 않으면 int로 간주하여 객체를 생성하고 함수를 호출한다.

초기에는 템플릿 선언에 class만 사용 가능했지만 typename 키워드도 사용할 수 있다. 즉, template <class T> 대신 template <typename T>를 사용할 수 있으며 의미, 기능상의 차이는 없다. T 대신 임의의 문자를 사용할 수 있으며 하나의 자료형뿐만 아니라 다음과 같이 둘 이상의 자료형을 템플릿으로 선언할 수 있다.

Ex. template <class T1, class T2>, template <class U = int, class V = float, class W = char> [6]

5. 참고 문헌

[1] 위키백과, “캡슐화”, https://ko.wikipedia.org/wiki/%EC%BA%A1%EC%8A%90%ED%99%94

[2][5] 서강대학교 컴퓨터 공학과 교수진, “컴퓨터공학 설계 및 실험 I”, *서강대학교 공학부 컴퓨터공학과*, 2012

[3][4] 남궁 성, “Java의 정석 3rd Edition”, *도우출판*, 2016

[6] 윤성우, “윤성우의 열혈 C++ 프로그래밍[개정판]”, *오렌지미디어,*  2010