실험 CPP-2: 다형성의 이해

1. 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. 실습 내용

아래에 나타낸 기초 클래스 LinkedList는 연결 리스트 자료구조를 제공하는 클래스이다. 리스트의 개별 노드는 Node 클래스로 나타나 있는데, 이는 int형 변수인 data와 다음에 연결된 노드의 포인터를 저장하는 Node 포인터형 변수 link로 구성된다. LinkedList 클래스에는 맨 처음의 노드를 가리키는 first 변수와 리스트의 노드의 개수를 저장하는 변수 current_size가 protected로 선언되어 있다. protected를 사용한 이유는 클래스 외부에서 직접 접근은 금하지만, 이 클래스를 상속하는 파생 클래스는 각 변수들을 사용할 수 있도록 하기 위해서이다. 또한 클래스는 4개의 멤버 함수를 가지는데, Insert() 함수는 리스트의 맨 앞에 노드를 삽입하는 작업을 수행하고, Delete() 함수는 리스트의 맨 뒤에 있는 노드를 삭제하는 작업을 수행한다(엄밀히 말해, 이 클래스의 초기 설정은 큐(Queue)의 정의를 따른다고 볼 수 있음). Print() 함수는 현재 리스트의 모습을 출력하는 함수이다. Print() 함수의 구현만을 제외한 LinkedList 클래스의 구현을 아래와 같이 보인다.

```
// Linked List Node
class Node{
public:
        int data;
        Node *link;
        Node(int element)
        {
                data = element;
                link = NULL;
        }
};

// Linked List Class
class LinkedList
{
protected:
        Node *first;
        int current_size;
public:
        LinkedList()
        {
                first = 0;
                current_size = 0;
        };
        int GetSize() { return current_size; };    // 리스트의 노드 개수를 리턴
        void Insert(int element);         // 맨 앞에 원소를 삽입
        virtual bool Delete(int &element);         // 맨 뒤의 원소를 삭제
        void Print();           // 리스트를 출력
};

void LinkedList::Insert(int element)
```

```
{
        Node *newnode = new Node(element);
        newnode->link = first;
        first = newnode;
        current_size++;
}

bool LinkedList::Delete(int &element)
{
        if(first == 0)
                return false;

        Node *current = first;
        Node *previous = NULL;

        while(1)
        {
                if(current->link == 0)          // find end node
                {
                        if(previous) previous->link = current->link;
                        else first = first->link;
                        break;
                }
                previous = current;
                current = current->link;
        }

        element = current->data;
        delete current;

        current_size--;

        return true;
}
```

[그림 1] LinkedList 클래스(및 Node 클래스)의 소스 코드

이 클래스는 또 다른 프로그래밍 작업에서 유용하게 사용할 수 있지만, 만약 int형이 아닌 double형, float형, string형, 또는 사용자 정의 클래스형 등을 저장할 필요가 발생할 경우에는 클래스에서 저장되는 데이터에 관련된 자료형을 모두 변경하여 사용하여야 한다. 그러나 교재 4-5-1에서 언급한 파라미터적 다형성(Parametric Polymorphism)의 기능을 사용하면, 이 클래스를 아주 약간 변경하는 것만으로 모든 자료형을 저장할 수 있는 연결 리스트 클래스를 작성할 수 있다. 저장되는 데이터 변수에 대해 템플릿(Template) 자료형을 사용하고 클래스의 선언부에 템플릿을 사용하는 것을 명시함으로써 템플릿 클래스를 만들 수 있다. 위 [그림 1]의 코드를 사용하여 이러한 클래스를 작성하고, 또한 구현이 나타나 있지 않은 Print() 함수를 직접 구현하여 본다.

작성된 템플릿 연결 리스트 클래스는, 아래와 같은 main() 함수를 통해 테스트해 볼 수 있다(string 자료형을 사용하기 위해 헤더 삽입부에 #include <string>을 꼭 삽입).

```
int main()
{
        double dVal;
        string strVal;
        LinkedList<double> dList;
        LinkedList<string> strList;
```

```
	dList.Insert(3.14);
	dList.Insert(123456);
	dList.Insert(-0.987654);
	dList.Print();
	dList.Delete(dVal);
	cout<<"삭제된 마지막 원소: "<<dVal<<endl;
	dList.Print();
	dList.Insert(777.777);
	dList.Print();
	dList.Delete(dVal);
	cout<<"삭제된 마지막 원소: "<<dVal<<endl;
	dList.Delete(dVal);
	cout<<"삭제된 마지막 원소: "<<dVal<<endl;
	dList.Print();
	dList.Delete(dVal);
	cout<<"삭제된 마지막 원소: "<<dVal<<endl;
	dList.Print();

	strList.Insert("This");
	strList.Insert("is a");
	strList.Insert("Template");
	strList.Insert("Example");
	strList.Print();
	strList.Delete(strVal);
	cout<<"삭제된 마지막 원소: "<<strVal<<endl;
	strList.Insert("Class");
	strList.Print();

	return 0;
}
```

[그림 2] 템플릿 연결 리스트 클래스 테스트용 main() 함수

그리고 위 main()함수를 수행하면 다음과 같은 결과가 출력된다.

```
[1|-0.987654]->[2|123456]->[3|3.14]
삭제된 마지막 원소: 3.14
[1|-0.987654]->[2|123456]
[1|777.777]->[2|-0.987654]->[3|123456]
삭제된 마지막 원소: 123456
삭제된 마지막 원소: -0.987654
[1|777.777]
삭제된 마지막 원소: 777.777
[1|Example]->[2|Template]->[3|is a]->[4|This]
삭제된 마지막 원소: This
[1|Class]->[2|Example]->[3|Template]->[4|is a]
```

[그림 3] 위의 main() 함수의 수행 결과

이렇게 템플릿 클래스를 작성한 이후, 이 클래스를 상속하여 스택 자료구조를 제공하는 클래스 Stack을 구현한다. 위의 연결 리스트와 스택의 다른 점은, 스택은 자료를 삭제할 때 맨 앞의 데이터를 삭제한다는 것이다. 그렇기 때문에, 위에서 구현한 템플릿 기반의 연결 리스트 클래스를 상속한 후 Delete() 함수만 재정의(Overriding)하여 맨 앞의 데이터 노드를 삭제하고 그 데이터를 넘겨주도록 하면 된다.

구현한 스택 클래스는 그 클래스 자료형의 변수나 포인터를 사용하여 사용할 수도 있지만, 연결 리스트 클래스가 기반 클래스인 파생 클래스이기 때문에, 교재 4-5-2에서 언급한 서브타입 다형성(Subtype Polymorphism)의 특성을 사용함으로써 기반 클래스인 연결 리스트 클래스의 포인터나 참조(Reference)를 사용하여 스택 클래스를 접근할 수 있다. 이렇게

할 때 정확한 작동을 보장하기 위해서는 4-5-2의 예제에서 설명한 바와 같이, 파생 클래스에서 재정의하는 함수가 virtual로 선언되어야 한다([그림 1]에서 LinkedList 클래스의 Delete() 함수가 virtual로 선언된 것을 확인할 수 있음). 클래스 Stack을 구현한 후, 다음의 main() 함수를 수행함으로써 테스트를 수행할 수 있다(prnMenu() 함수는 main() 함수에서 자주 호출되므로 따로 빼놓음). 스택 클래스가 잘 구현되었다면 다음 코드를 실행할 때 스택 또는 리스트의 선택에 따라 각 자료 구조의 원활한 작동이 보장된다.

```
void prnMenu()
{
        cout<<"*****************************************"<<endl;
        cout<<"* 1. 삽입     2. 삭제     3. 출력    4. 종료 *"<<endl;
        cout<<"*****************************************"<<endl;
        cout<<endl;
        cout<<"원하시는 메뉴를 골라주세요: ";
}

int main()
{
        // 스택 및 연결 리스트 테스트용 코드
        int mode, selectNumber, tmpItem;
        LinkedList<int> *p;
        bool flag = false;

        cout<<"자료구조 선택(1: Stack, Other: Linked List): ";
        cin>>mode;

        // 기반 클래스의 포인터를 사용하여 기반 클래스 뿐만 아니라
        // 파생 클래스의 인스턴스 또한 접근할 수 있다.
        if(mode == 1)
                p = new Stack<int>();          // 정수를 저장하는 스택
        else
                p = new LinkedList<int>();    // 정수를 저장하는 연결 리스트

        // 처리 부분
        do{
                prnMenu();
                cin>>selectNumber;
                switch(selectNumber)
                {
                case 1:
                        cout<<"원하시는 값을 입력해주세요: ";
                        cin>>tmpItem;
                        p->Insert(tmpItem);
                        cout<<tmpItem<<"가 삽입되었습니다."<<endl;
                        break;
                case 2:
                        if(p->Delete(tmpItem)==true)
                                cout<<tmpItem<<"가 삭제되었습니다."<<endl;
                        else
                                cout<<"비어있습니다. 삭제 실패"<<endl;
                        break;
                case 3:
                        cout<<"크기: "<<p->GetSize()<<endl;
                        p->Print();
                        break;
                case 4:
                        flag = true;
                        break;
                default:
                        cout<<"잘못 입력하셨습니다."<<endl;
```

```
                        break;
                }
                if(flag)
                        break;
        }
        while(1);

        return 0;
}
```

[그림 4] 템플릿 연결 리스트 및 스택 클래스 테스트용 main() 함수

위 코드는 처음에 어떤 자료 구조를 선택하느냐에 따라 동작 방식이 바뀐다. 다음은 위 코드의 동작 예를 나타낸 것이다.

스택으로 작동

```
자료구조 선택(1: Stack, Other: Linked List): 1
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 33
33가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 44
44가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 55
55가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 3
크기: 3
[1|55]->[2|44]->[3|33]
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 2
55가 삭제되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 66
66가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 3
크기: 3
[1|66]->[2|44]->[3|33]
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요:
```

연결 리스트로 작동

```
자료구조 선택(1: Stack, Other: Linked List): 2
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 11
11가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 22
22가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 3
크기: 2
[1|22]->[2|11]
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 33
33가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 1
원하시는 값을 입력해주세요: 44
44가 삽입되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 2
11가 삭제되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 2
22가 삭제되었습니다.
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요: 3
크기: 2
[1|44]->[2|33]
*****************************************
* 1. 삽입     2. 삭제     3. 출력    4. 종료 *
*****************************************

원하시는 메뉴를 골라주세요:
```

3. 실험 방법

3-1. 문제 해결

기본 클래스인 LinkedList 클래스를 먼저 구현하여야 한다. 이 클래스는 [그림 1]의 선언부 및 구현부를 그대로 옮긴 후 Print() 함수를 [그림 3]의 출력 예의 형태로 출력될 수 있도록 코드를 작성함으로써 만들 수 있다.

이후, C++의 템플릿을 사용하여 기본 클래스를, 파라미터적 다형성이 지원되는 템플릿 기반의 연결 리스트 클래스로 변환한다. 이는 단지 저장되는 데이터의 자료형을 임의의 타입으로 바꾸고, 그 임의의 타입을 템플릿 자료형으로 쓸 것이라는 것을 알려 주는 선언부를 클래스의 선언에 추가하고, 클래스 이름을 LinkedList<템플릿자료형>의 형태로 수정함으로써 매우 손쉽게 구성할 수 있다. 교재 및 인터넷 자료 등을 참조한다.

그 다음, 이 클래스를 상속하는 Stack 클래스를 작성하는데, 이 또한 템플릿을 사용하는 클래스라는 사실을 유념하여야 한다. 스택의 정의에 따라, 단지 Delete() 함수만을 새롭게 재정의하여 구현함으로써 간단히 파생 클래스를 만들 수 있다. 그리고 [그림 4]의 코드를 반복 수행함으로써 작성한 프로그램 코드의 무결성을 검증한다.

실습 내용이 잘 이해되지 않을 경우, 교재(특히 4-4, 4-5)의 설명을 충분히 검토한다.

3-2. 프로그램 작성

본 실험은 Unix 또는 Linux환경에서 g++를 사용한다. 생각한 해결방법을 다음과 같은 단계를 거쳐 프로그래밍을 수행한다.

1. 문제 해결에 필요한 클래스들을 설계한다.
2. 간단한 예를 만들어 이에 자신이 설계한 클래스에 적용하여 실행 및 검토한다.
3. 잘못된 입력에 대해 적절한 에러 메시지를 출력해준다.
4. 문제의 주어진 예 외에도 여러 입력을 적용하여 자신의 프로그램을 수행시켜 보고 필요한 경우 디버깅 과정을 통하여 수행과정을 살펴보거나 발견된 오류를 정정한다.

3-3. 프로그램 작성 확인

담당 강사 또는 조교에게 작성한 프로그램을 확인 받는다.

4. 숙제 및 보고서 작성

4-1. 예비 보고서

주어진 문제 및 3-1의 문제 해결에 관한 내용을 이해하고 이 문제를 효율적으로 해결하기 위한 방법을 생각하여 이를 1쪽 이내로 요약하여 제출하시오. 문제 해결을 위한 간단한 단계별 수행 내용, 자료구조 등을 기술하시오.

작성한 예비 보고서는 실험 시작 전에 제출하여야한다.

4-2. 숙제

실험 당일 강사가 지정한 문제를 해결하는 프로그램을 해결하는 프로그램을 작성한다.

4-3. 결과 보고서

아래에 보인 사항을 작성하여 다음 실험 시간에 제출하시오.

1. 실험 시간에 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술하시오.
2. 숙제 문제를 해결하기 위한 알고리즘 및 자료구조를 요약하여 기술하시오.
3. 작성한 프로그램을 공지한 제출요령에 의거하여 기한 내에 제출하시오, 요청이 있을 경우 실험 시간에 작성한 프로그램도 아울러 제출한다.

실험 CPP-2: 예비 보고서

전공: 학년: 학번: 이름

실험 CPP-2: 결과 보고서

전공: 학년: 학번: 이름