2020년도 2학기 컴퓨터공학설계및실험I

4주차 결과 보고서

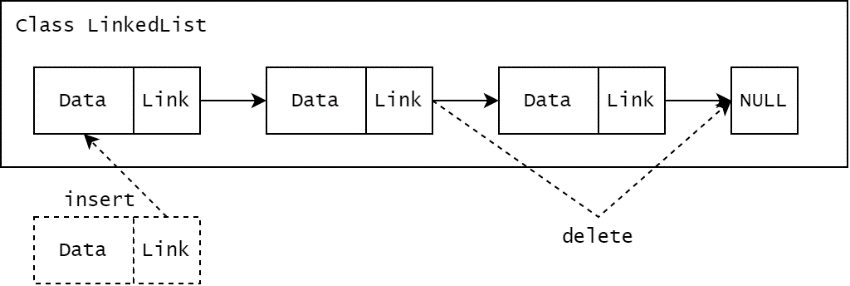
20170175 김태안

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.

1. 실습 구현 내용

**Linked List**



|  |  |
| --- | --- |
| template <class T>  class Node {  public:  T data;  Node \*link;    Node(T element){  data = element;  link = 0;  }  }; | template <class T>  class LinkedList {  protected:  Node<T> \*first;  int current\_size;    public:  LinkedList(){  first = 0;  current\_size = 0;  }    int GetSize(){  return current\_size;  }  void Insert(T element);  virtual bool Delete(T &element);  void Print();  }; |

Linked List를 구현하기 위해서 먼저 Linked List를 구성하는 노드를 정의한다. 노드는 데이터를 저장하는 data와 다음 노드를 가리킬 link로 구성된다. 이때, 위 코드와 같이 template을 사용하여 data를 선언해 다양한 종류의 데이터를 담을 수 있게 한다. 마찬가지로 Linked List도 노드에 맞는 원소를 입력 받을 수 있도록 template을 통해 다형성을 확보한다. Linked List는 첫번째 노드와 Linked List의 크기를 저장한다. Linked List의 메소드는 다음과 같다.

**LinkedList();**

Linked List를 생성한다.

**int GetSize();**

Linked List의 크기를 반환한다.

**void Insert(T element);**

Linked List의 맨 앞에 새 element를 삽입한다.

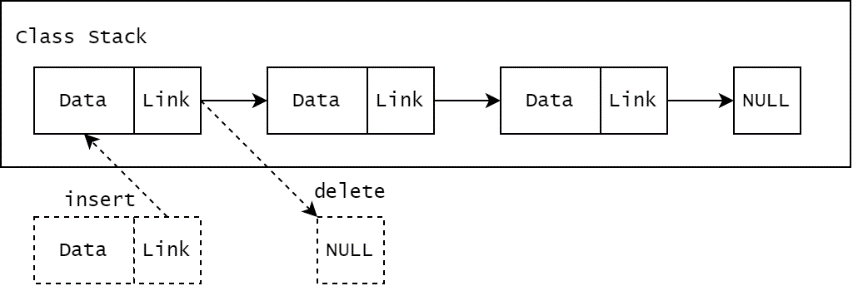
**virtual bool Delete(T &element);**

Linked List의 맨 뒤의 원소를 삭제한다.

**void Print();**

Linked List의 원소를 모두 출력한다.

**Stack**



|  |
| --- |
| template <class T>  class Stack:public LinkedList<T> {    public:  virtual bool Delete(T &element);  }; |

Stack은 위의 Linked List와 매우 유사하지만, 원소를 삭제할 때 맨 뒤가 아닌 맨 앞의 원소를 삭제한다. 따라서 Stack을 구현하기 위해서는 Linked List를 상속받은 뒤, Delete를 수정하는 것이 효율적이다. 이를 위해 Stack::Delete를 재정의한다.

**Delete(T &element);**

맨 앞의 원소를 삭제한다.

1. 실습 환경

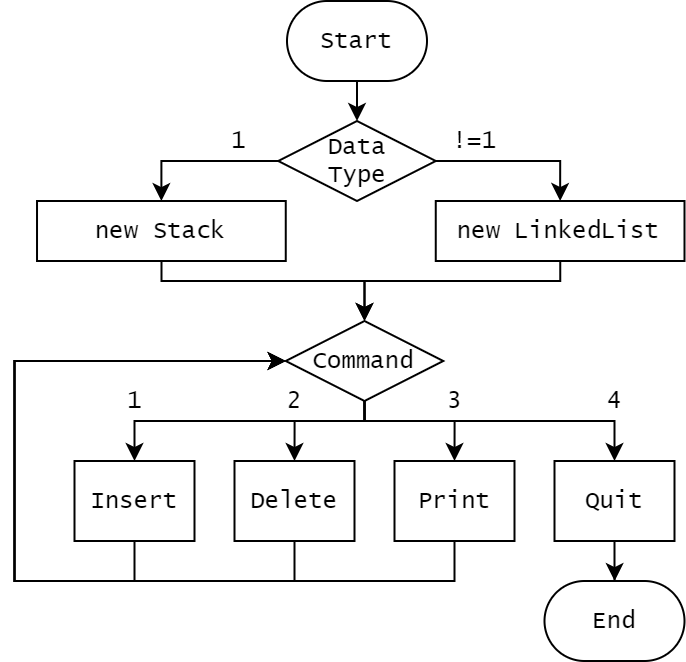
cspro.sogang.ac.kr

OS: Ubuntu 16.04.2 LTS (GNU/Linux 4/4/0-184-generic x86\_64)

Compiler: g++ (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609

1. 실습 결과 및 분석

**실습 코드**

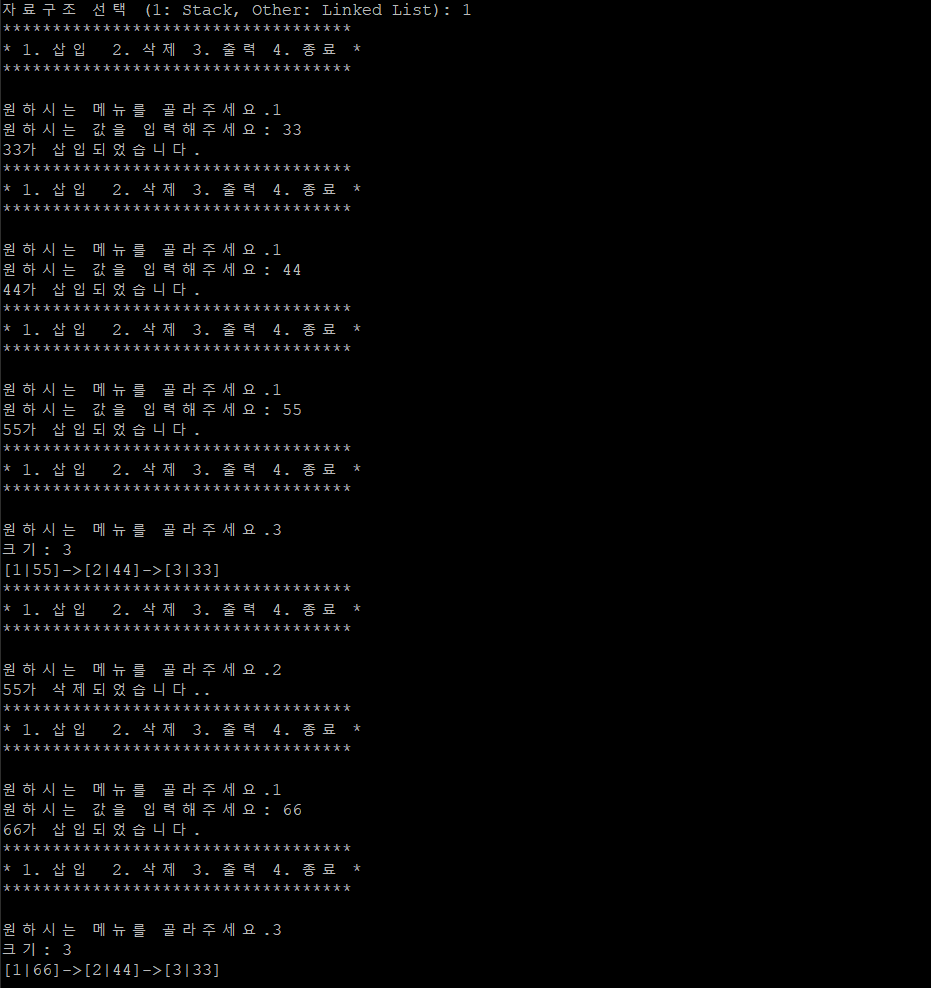
****

구현한 Linked List와 Stack을 테스트하기 위한 테스트 코드를 위와 같이 구현했다. 이때, 테스트 코드는 Data Type에 따라 다른 프로세스가 진행되는 것이 아니라, Data Type이 선언된 이후 같은 프로세스를 통해 Insert, Delete, Print, Quit이 진행된다. 즉, 어떤 Data Type을 선택하든 데이터를 저장하는 p는 모두 같은 Insert 명령어, Delete 명령어, Print 명령어에 인자로 주어지고, 명령어는 명령을 수행한다. 이는 서브타입 다형성을 구현하기 위해 데이터를 저장하는 p를 LinkedList 포인터로 선언했기 때문이다. 따라서 Data Type을 선택한 뒤, p의 형식이 결정되고, p가 Stack일 경우여도 서브타입 다형성이 구현되며 오류가 발생하지 않는다. 이후 명령어는 Insert와 Print는 상속받은 메소드를 사용하고, Delete는 재정의되었기 때문에 문제 없이 실행된다. 다음은 서브타입 다형성을 구현하기 위한 코드의 일부이다.

|  |
| --- |
| …  LinkedList<int> \*p;  …  if(mode == 1)  p = new Stack<int>();  else  p = new LinkedList<int>(); |

**Stack**

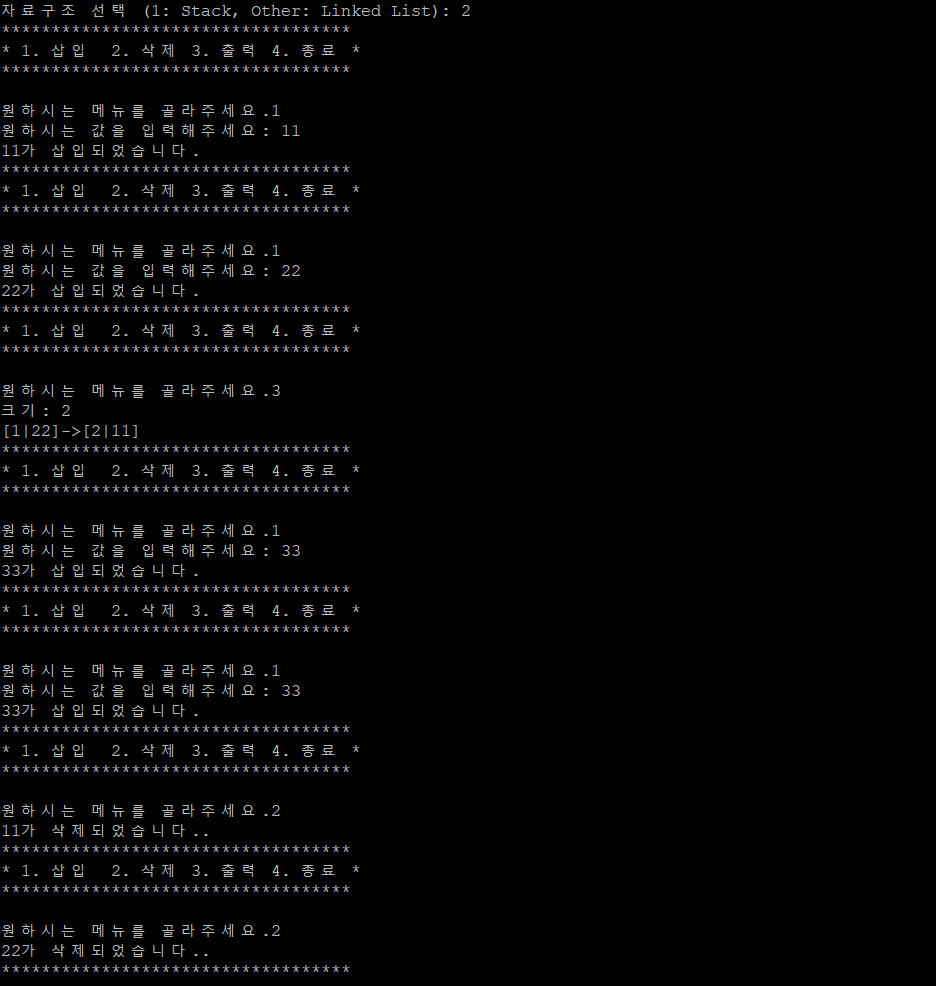
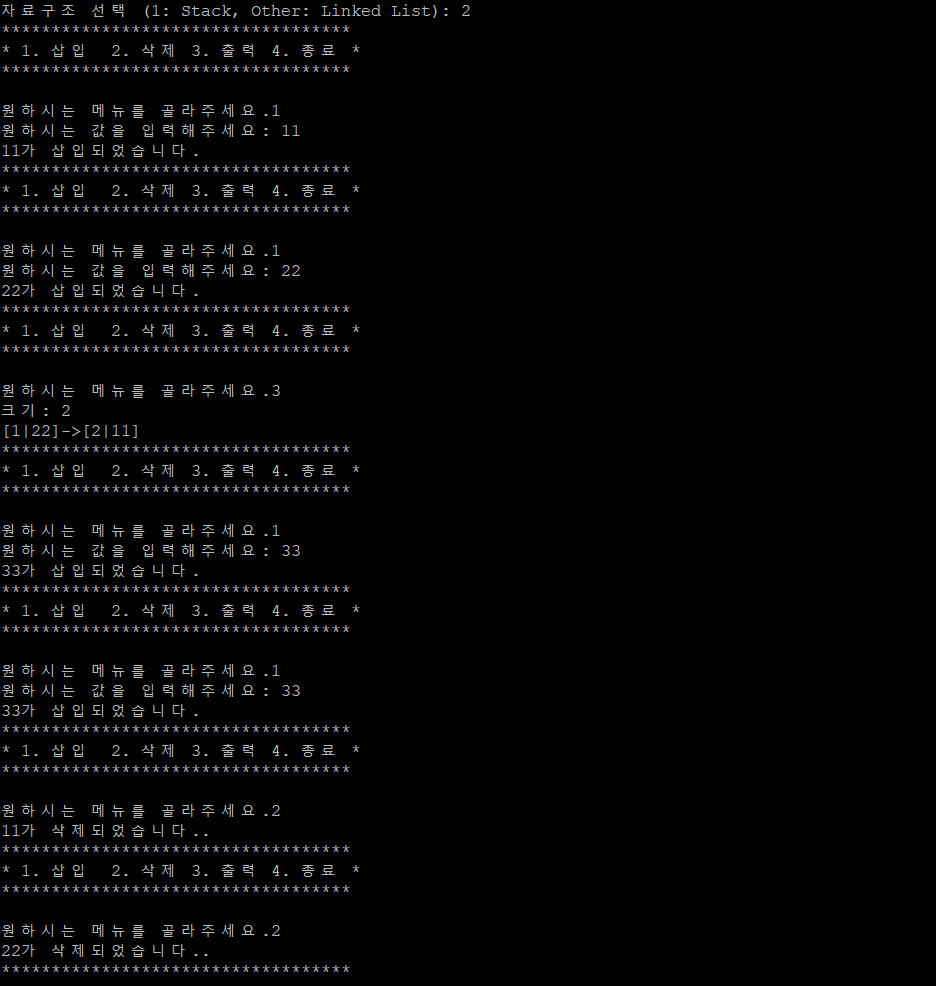
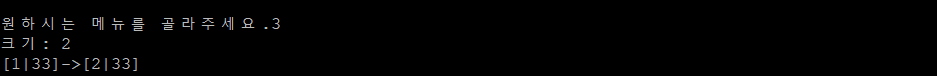
구현한 Stack의 테스트 결과는 다음과 같다.



원소를 삭제할 때, 맨 앞의 원소가 삭제되는 것을 확인할 수 있다.

**Linked List**

구현한 Linked List의 테스트 결과는 다음과 같다.

Linked List는 맨 마지막 원소부터 삭제되는 것을 확인할 수 있다.

1. 과제

**Array**

Array는 이전 프로젝트에서 사용한 Array를 그대로 사용하되 다양한 Data Type을 사용하기 위해 Template을 추가해야 한다. Template을 사용한 배열 포인터 data와 그 길이 len을 원소로 가진다.

|  |
| --- |
| template <class T>  class Array {  protected:  T \*data;  int len;  public:  Array(int size);  ~Array();  int length() const;  virtual T & operator [] (int i);  virtual T operator [] (int i) const;  void print();  }; |

Array가 사용하는 메소드는 다음과 같다.

**Array(int size);**

size만큼의 배열을 생성한다.

**~Array();**

소멸자

**int length() const;**

배열의 길이를 반환한다.

**virtual T & operator [] (int i);**

배열의 인덱스를 받아 탐색한 결과를 반환한다.

**virtual T operator [] (int i) const;**

배열에 새 값을 할당한다.

**void print();**

배열의 내용을 출력한다.

**GrowableArray**

Growable Array는 범위를 벗어난 배열을 입력 받으면 에러메시지를 출력하는 Array와 달리, 범위가 벗어나면 배열의 길이를 현재의 2배로 늘리는 배열이다. 기존 배열의 기능을 사용하기 위해 Array를 상속하며, 마찬가지로 다양한 Data Type을 사용하기 위해 Template을 사용하며, [] 연산자에 대해 Array와는 다른 역할을 수행하기 위해 해당 함수는 새로 정의한다.

|  |
| --- |
| template <class T>  class GrowableArray:public Array<T> {  public:  GrowableArray(int size) : Array<T> (size);  ~GrowableArray();  virtual T & operator[] (int i);  virtual T operator[] (int i) const;  }; |

GrowableArray의 메소드는 다음과 같다.

**GrowableArray(int size) : Array<T> (size);**

Array를 생성한다.

**~GrowableArray();**

소멸자

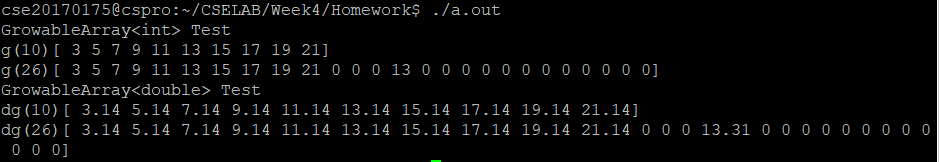
**virtual T & operator[] (int i);**

i가 기존 배열의 범위를 넘어가면 길이가 두배인 새 배열을 생성하고, 데이터를 복사한다.

**virtual T operator[] (int i) const;**

i번째 원소를 반환한다.

GrowableArray를 테스트한 결과는 다음과 같다.



범위 밖의 주소에 값을 저장하려 하자 배열의 길이가 두배가 되며 저장된다. 정수와 실수 모두 저장할 수 있다.

1. 결론

이번 실습에서는 C++에서의 다형성에 대해 학습할 수 있었다. Template을 사용하면 미리 Data Type을 정하지 않고도 Class와 함수를 구현할 수 있으며, 나중에 해당 Class나 함수를 사용할 때 Data Type을 지정해 사용할 수 있다. 이를 통해 Data Type만 다른 같은 코드의 반복을 줄이며, 중복된 선언을 할 필요가 없어 코드의 이해도 쉬워진다.

서브타입 다형성을 구현할 경우, Class에 상관없이 같은 명령어를 사용할 수 있다. 이 경우 Data Type마다 다른 프로세스를 진행할 필요가 없어 알고리즘을 단순화시킬 수 있다.