2020년도 2학기 컴퓨터공학설계및실험Ⅰ

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

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. 실습 구현 내용

LinkedList 및 Stack 클래스 선언 부분을 먼저 작성한 후, 각 멤버 변수와 메소드가 어떤 역할을 하는지 간단하게 정리한다.

Node 클래스와 LinkedList 클래스의 선언 부분(linkedlist.h 파일의 일부분)

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

Stack 클래스의 선언 부분(stack.h 파일의 일부분)

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

LinkedList 클래스

멤버 변수

(1) Node<T>\* first;

LikedList의 가장 앞의 원소를 가진 노드를 가리키는 포인터이다. 원소를 가지고가 추가될 경우 first는 가장 최근에 추가된 노드를 가리킨다. 노드의 원소는 임의의 자료형의 원소를 가질 수 있도록 템플릿 클래스의 임의의 자료형인 T로 한다.

(2) int current\_size;

현재 LinkedList에 저장된 원소(원소를 가지고 있는 노드)의 개수를 나타내는 정수형 변수이다.

멤버 함수

(1) LinkedList<T>::LinkedList()

LinkedList 클래스의 생성자이다. LinkedList의 첫 번째 노드를 가리키는 first를 NULL(=0)로 초기화하고 LinkedList에 저장된 노드의 개수를 나타내는 변수 current\_size를 0으로 초기화한다

input : 없음

output : 없음

(2) int LinkedList<T>::GetSize() const

현재 LinkedList에 저장된 노드의 개수 current\_size를 반환하는 함수이다. 함수 내에서 current\_size의 값이 변해서는 안되므로 const 선언을 붙인다.

input : 없음

output : 현재 LinkedList에 저장된 노드의 개수 current\_size

(3) void LinkedList<T>::Insert(T element)

LinkedList의 맨 앞에 새로운 원소 element를 가진 노드를 추가하는 함수이다. 새로운 노드 newNode를 동적할당한 후, newNode의 link에 기존 LinkedList의 first를 대입하여 newNode가 기존 LinkedList의 first를 가리키게 한다. 그 다음 first에 newNode를 대입하여 새로 추가된 newNode가 LinkedList의 첫번째 노드가 되게 한다. 마지막으로 현재 LinkedList의 노드의 개수인 current\_size를 1 증가시킨다.

input : 새로 추가할 노드에 저장할 템플릿 자료형의 원소 element

output : 없음

(4) bool LinkedList<T>::Delete(T& element)

LinkedList의 가장 마지막 노드를 제거한다. LinkedList에서 노드의 추가가 앞에서 이루어지기 때문에 가장 마지막 노드는 가장 처음 추가된 노드이다. LinkedList에서의 삭제는 Queue의 FIFO(First In First Out) 정책을 따르도록 구현한다. LinkedList의 first가 0(NULL)일 경우 저장된 노드가 하나도 없으므로 false를 반환한다. first가 NULL이 아닐 경우 Node<T>\* 변수 current가 first부터 시작하여 LinkedList의 가장 마지막 원소를 찾아서 이동한다. 끝 노드를 찾고 제거하는데 current 변수와 함께 Node<T>\* 변수 previous를 사용한다. 마지막 노드를 찾으면 current가 가리키는 마지막 노드의 데이터를 입력 인자로 받은 참조자 element에 저장 후 현재 저장된 노드의 개수인 current\_size를 1 감소시키고 true를 반환한다. Delete 함수는 LinkedList 클래스 포인터가 가리키는 인스턴스에 따라 다르게 동작하는 서브타입 다형성을 구현해야 하므로 선언할 때 virtual 키워드를 이용해 가상함수로 선언한다.

input : 제거될 노드의 데이터를 저장할 템플릿 자료형의 참조자 element

output : LinkedList에서 노드의 제거가 이루어졌을 경우 true, 그렇지 않을 경우 false를 반환

(5) void LinkedList<T>::Print() const

LinkedList의 가장 앞의 노드의 원소부터 마지막 원소까지 차례대로 형식에 맞게 출력하는 함수이다. 현재 출력할 노드를 가리킬 Node<T>\*형 변수 current를 first로 초기화 후 사용한다. 처음 current가 NULL일 경우 저장된 노드가 없으므로 아무것도 하지 않는다. 저장된 노드가 있을 경우 current가 NULL이 될 때까지 current가 가리키는 노드의 원소를 출력 후 current를 다음 노드로 이동시키는 과정을 반복한다.

input : 없음

output : 없음

Stack 클래스

Stack 클래스는 LinkedList 클래스를 기반 클래스로 상속하는 파생 클래스로 LinkedList 클래스의 모든 멤버 변수를 그대로 사용하며 Delete 함수를 제외한 모든 멤버 함수를 그대로 사용한다. Stack은 LinkedList와 다르게 저장된 노드를 제거할 때 LIFO(Last In First Out) 정책을 따른다. 따라서 LinkedList에서 작성한 Delete 함수를 Stack 클래스에서 재정의(Overriding)해야 한다. 또한 서브타입 다형성을 구현하여 LinkedList 포인터가 Stack 클래스의 인스턴스를 가리킬 경우 LinkedList 포인터로 Delete 함수를 호출할 경우 Stack 클래스에서 재정의한 Delete 함수가 호출되도록 한다. Stack 클래스에서 재정의된 Delete 함수 역시 LinkedList 클래스에서 정의된 Delete 함수와 마찬가지로 가상함수이다.

재정의된 멤버 함수

bool Stack<T>::Delete(T& element)

가장 마지막에 추가된 노드가 가장 먼저 제거되는 LIFO 정책을 구현한 함수이다. LinkedList 클래스에서 상속받은 first가 0일 경우, Stack에 아무것도 저장되어 있지 않다는 의미이므로 false를 반환한다. first가 0이 아닐 경우 제거할 노드를 가리키는 Node<T>\*형 변수 delNode를 선언하여 delNode가 first를 가리키게 한다. 입력 인자로 받은 참조자 element에 제거할 노드 delNode에 저장된 원소를 저장한다. delNode를 delete 명령어로 제거하고 LinkedList 클래스에서 상속받은 current\_size를 1 감소시킨 후 true를 반환한다.

input : 제거될 노드의 데이터를 저장할 템플릿 자료형의 참조자 element

output : LinkedList에서 노드의 제거가 이루어졌을 경우 true, 그렇지 않을 경우 false를 반환

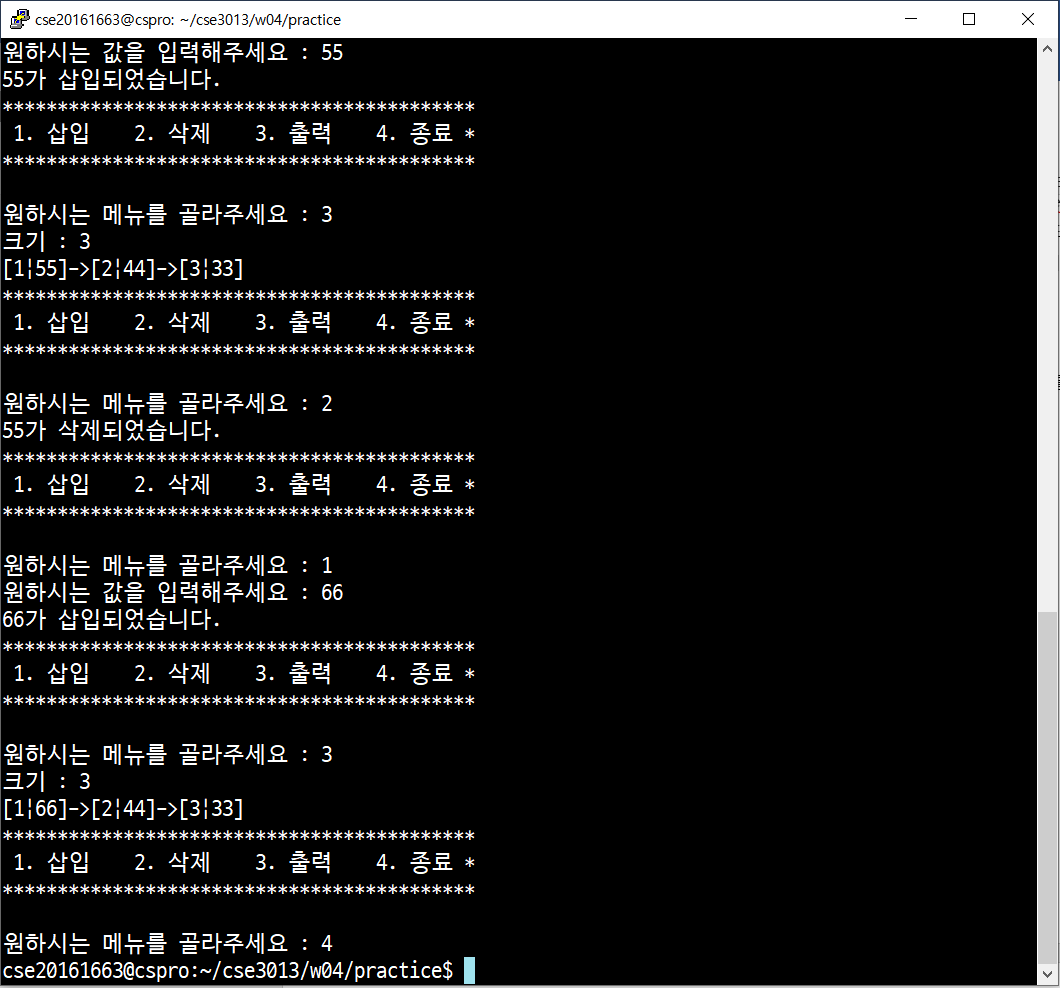
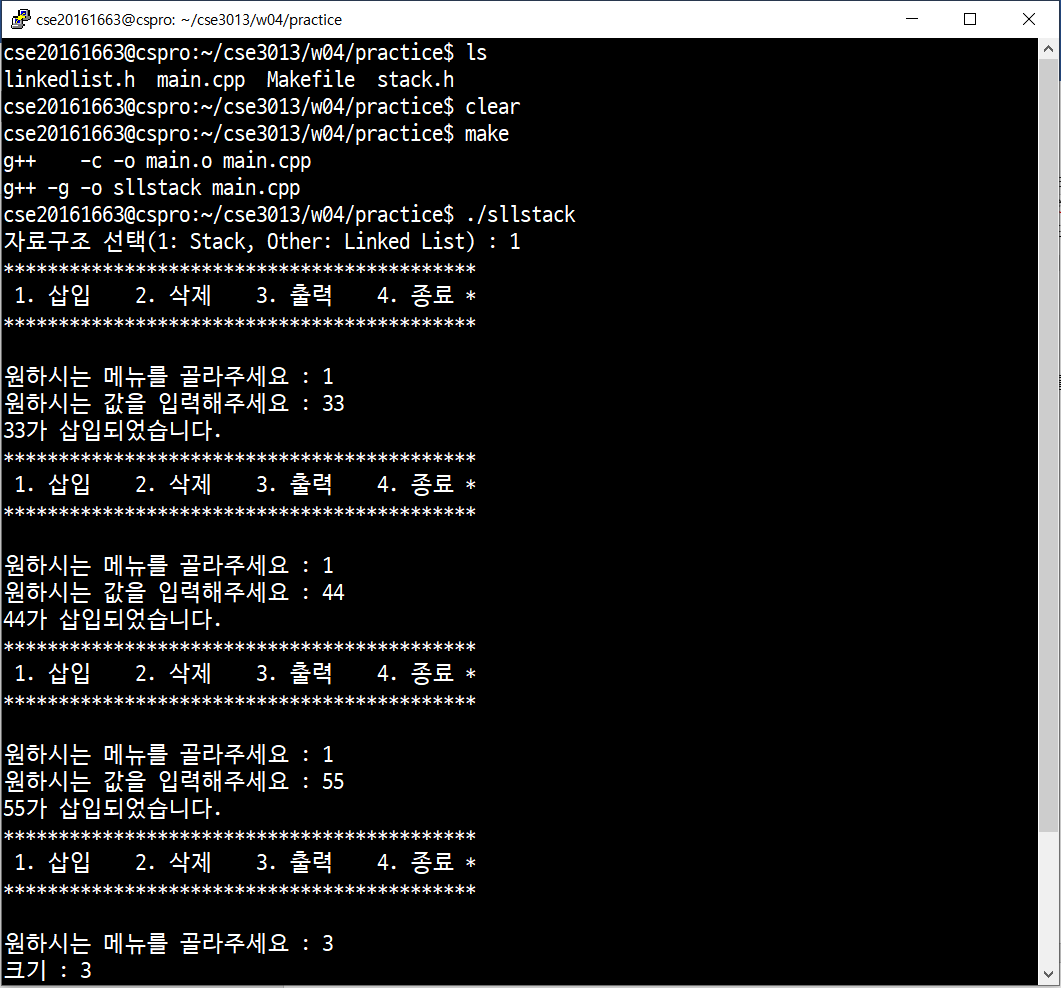
3. 실습 환경

CSPRO 서버 : CSPRO.sogang.ac.kr

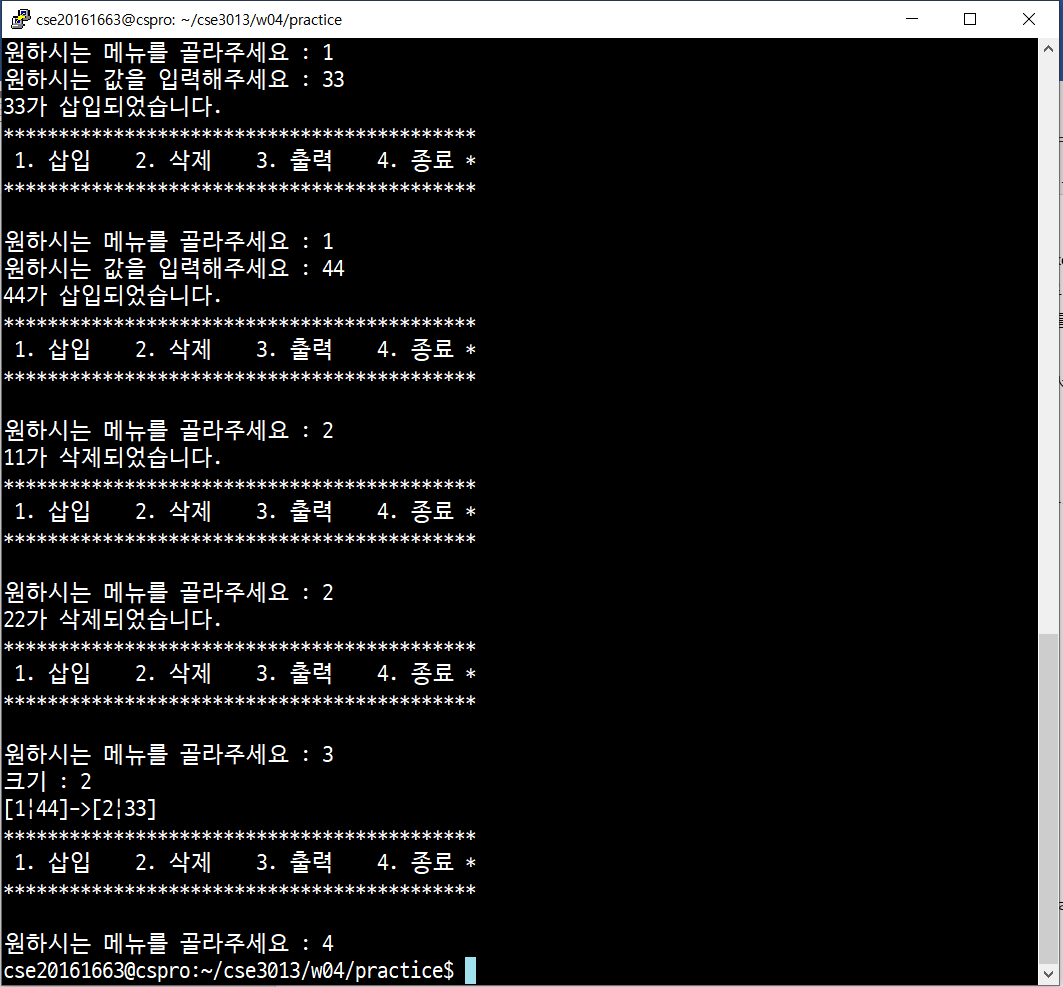
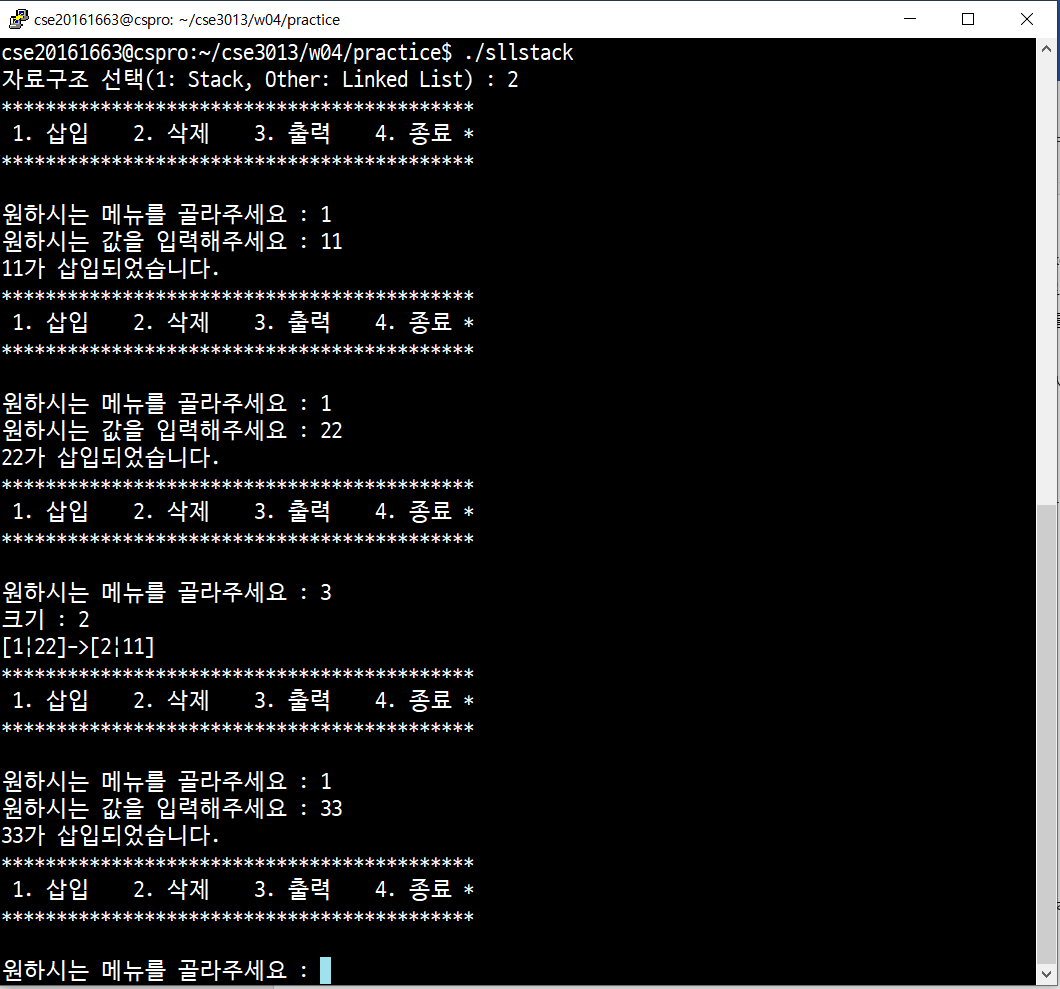
4. 실습 결과 및 분석

최종 테스트 코드로 구현한 LinkedList와 Stack을 테스트한 결과는 다음과 같다.

4-1. Stack으로 동작



2. LinkedList로 동작



4-1, 4-2 두 경우 모두 공통적으로 새로운 노드를 추가할 경우 가장 앞에 추가되는 것을 확인할 수 있다. 노드를 제거하는 과정에서만 차이를 보인다.

4-1처럼 Stack으로 동작할 경우 삭제 명령어를 통해 노드를 삭제하면 가장 앞에 있는 노드가 삭제되는 것을 알 수 있다. 노드를 추가할 때 가장 앞에 추가하므로 Stack에서의 삭제는 마지막에 추가된 노드를 먼저 삭제하는 것을 확인할 수 있다. Stack의 LIFO 정책이 구현된 것을 알 수 있다. 4-2처럼 LinkedList로 동작할 경우 삭제 명령어를 통해 노드를 삭제하면 가장 뒤의 노드가 먼저 삭제되는 것을 확인할 수 있다. 노드를 추가할 때 가장 앞에 추가하므로 LinkedList에서의 삭제는 가장 먼저 추가된 노드를 먼저 삭제하는 것을 알 수 있다. 이번 실습에서의 LinkedList는 Queue와 같이 FIFO 정책이 구현된 것을 알 수 있다.

최종 테스트 코드에서는 처음 입력에 따라 LinkedList 인스턴스와 Stack 인스턴스 둘 다 사용할 수 있다. 하지만 LinkedList형 포인터 변수와 Stack형 포인터 변수를 둘 다 선언할 필요 없이 기반 클래스인 LinkedList형 포인터 변수 p만 선언해도 LinkedList 인스턴스와 Stack 인스턴스를 둘 다 다룰 수 있다.

main.cpp 파일의 일부분

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

LinkedList형 포인터 p로 함수를 호출할 경우 포인터의 정적 타입은 LinkedList이므로 가상 함수가 아닌 함수를 호출할 경우 포인터 p가 가리키는 실제 인스턴스가 파생 클래스의 인스턴스라도 파생 클래스에 재정의된 함수가 아닌 LinkedList 클래스에 작성된 함수를 호출한다. 하지만 호출하려는 함수가 virtual 키워드로 인해 가상함수로 선언되어 있으면 포인터 p가 가리키는 실제 인스턴스의 클래스에 따라 해당 클래스에 재정의된 함수가 호출된다.

|  |
| --- |
| case 2:  if(p->Delete(tmpItem) == true)  … |

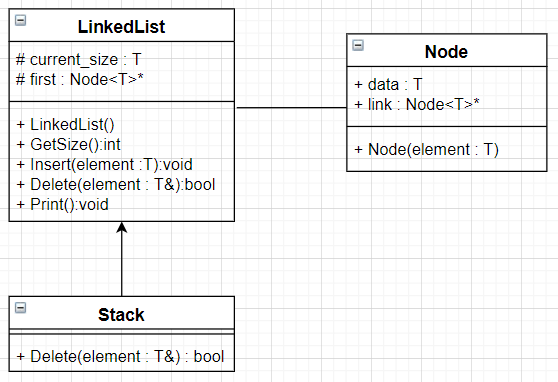
기반 클래스 LinkedList의 Delete 함수, 그리고 파생 클래스 Stack에 재정의된 Delete 함수 모두 가상 함수로 선언되어 있으므로 LinkedList 포인터 p로 Delete 함수를 호출할 경우 p가 LinkedList 인스턴스를 가리킬 경우 LinkedList 클래스에 작성된 Delete 함수를 호출하여 Queue처럼 FIFO 정책을 따라 가장 먼저 추가된 노드를 제거하고, p가 Stack 인스턴스를 가리킬 경우 Stack 클래스에 재정의된 Delete 함수를 호출하여 LIFO 정책을 따라 가장 최근에 추가된 노드를 제거한다.

같은 메시지(p->Delete(tmpItem))에 대해서도 p가 가리키는 인스턴스에 따라 다른 함수가 호출되는 서브타입 다형성이 구현되었다.

사용한 자료구조 및 알고리즘

연결리스트(LinkedList)를 구현하기 위해 노드(Node) 클래스를 정의했다. 노드 클래스는 저장할 데이터와 다른 노드 객체를 가리키는 포인터로 이루어져 있다. C에서 연결 리스트의 노드 구조체를 작성할 때 자기 자신을 참조하는 구조체(self-referential)로 작성한 것처럼 C++에서 구조체를 클래스로 바꾸고 생성자와 같은 멤버함수를 정의하여 노드 클래스를 구현했다. 노드 클래스를 이용하여 노드 클래스를 포함하는 연결 리스트 클래스를 구현했다. 연결 리스트의 노드 삽입, 삭제 방식은 큐(Queue)의 FIFO 정책을 따르도록 했고, 연결 리스트 클래스를 상속받고 삭제 방법만 재정의한 스택(Stack) 클래스를 작성하여 LIFO 정책을 따르게 구현했다. 다형성을 이용하여 기반 클래스의 포인터 변수로 기반 클래스 객체와 파생 클래스 객체를 모두 가리킬 수 있게 하고 함수를 호출 시 동적 타입을 확인하여 실제 가리키는 객체의 클래스에 작성된 함수를 호출한다.

프로그램의 구조도



Node, LinkedList, Stack 클래스의 관계를 나타낸 구조도이다. 각 클래스의 윗부분의 멤버는 멤버변수를 의미하고 아랫부분의 멤버는 멤버 함수를 의미한다. 멤버명의 #, +, -는 접근 제어자로 각각 protected, public, private를 의미한다. LinkedList클래스가 Node 클래스가 제공하는 특성 및 기능을 사용한다. Stack 클래스는 LinkedList 클래스를 상속한다. 그대로 상속받는 멤버는 따로 표시하지 않는다. 새로 추가되거나 재정의된 멤버를 표시하는데 Stack 클래스에서 LinkedList 클래스의 Delete 멤버 함수를 재정의하였으므로 표시한다. 파라미터적 다형성을 구현했으므로 특정 데이터형 대신 파라미터 T를 사용한다.

5. 과제

3주차 실습때 구현한 Array 클래스를 이용하여 Array 클래스를 상속받는 클래스인 GrowableArray 클래스를 구현한다. GrowableArray 클래스의 인스턴스는 배열의 범위 밖의 인덱스를 사용할 경우 접근하고자 하는 인덱스 크기의 2배 크기로 배열의 크기를 증가시켜 해당 인덱스에 접근이 가능하도록 한다. 기존 배열에서 추가된 공간의 배열은 0으로 초기화한 후 접근에 따라 값을 변경하도록 구현한다. 추가적으로 파라미터적 다형성을 구현하기 위해 기존의 Array 클래스를 템플릿 클래스로 변경하고 Array 클래스를 상속받는 GrowableArray 클래스도 템플릿 클래스로 작성한다.

GrowableArray 클래스 선언 부분

|  |
| --- |
| Template <class T>  Class GrowableArray : public Array<T> {  private:  void doubleArray(int idx);  public:  GrowableArray(int size);  T& operator [](int i);  T operator [](int i) const;  }; |

GrowableArray 클래스는 Array 클래스로부터 상속받은 멤버 변수만을 사용한다. 추가적으로 사용하는 멤버 변수는 없다. GrowableArray 클래스는 GrowableArray 클래스 내에서만 자체적으로 호출되어 사용되는 doubleArray 함수가 따로 있다. 또한 GrowableArray 인스턴스를 생성할 수 있는 생성자를 정의하고 Array 클래스에서 다중정의된 [] 연산자들을 GrowableArray에서 재정의한다.

생성자 및 멤버 함수

(1) GrowableArray<T>::GrowableArray(int size)

GrowableArray 클래스의 인스턴스를 생성하기 위한 생성자로 size 크기만큼의 GrowableArray 객체를 생성한다. 배열을 생성하기 위해 기반 클래스인 Array 클래스에 선언된 T\* 변수 data를 사용해야 한다. 따라서 Array<T>::Array(size) 형태로 기반 클래스의 생성자를 호출한다.

input : GrowableArray 인스턴스의 배열의 크기 size

output : 없음

(2) void GrowableArray<T>::doubleArray(int idx)

생성된 GrowableArray 인스턴스의 범위를 넘어가는 인덱스 idx에 접근했을 시 기존 배열의 크기를 idx의 두배로 늘려주는 함수이다. 해당 함수를 호출하면 크기가 증가한 배열의 인덱스는 0~2 \* idx – 1 까지이므로 인덱스 idx에 접근할 수 있게 된다. doubleArray 함수가 호출되면 기존 배열 data의 시작 인덱스에서 끝 인덱스까지 새로 동적할당된 배열 temp에 그대로 복사되고 temp에 추가된 공간의 원소는 모두 0으로 초기화한다. 그 후 data가 가리키던 기존의 배열은 delete 명령어로 메모리가 해제되고 data 변수가 새로 할당된 배열(temp가 가리키는 배열)을 가리키게 한다. 마지막으로 배열의 길이를 나타내는 Array 클래스의 멤버변수 len의 크기를 idx \* 2로 업데이트한다.

해당 함수는 범위를 벗어나는 인덱스에 접근했을 때 배열의 크기를 증가시키기 위해 클래스 내부적으로 호출되는 함수로 클래스 외부에서 호출할 수 없도록 접근 제어자를 private로 선언한다.

재정의된 연산자 다중정의

(1) T& GrowableArray<T>::operator [](int i)

입력 인자로 전달받은 인덱스 i가 기존 배열의 인덱스 범위 내에 있는 지 확인한다. 기존 배열의 인덱스 범위를 벗어날 경우 위에서 설명한 GrowableArray 함수에 인덱스 i를 전달해 호출하여 해당 인덱스에 접근 가능하도록 배열의 크기를 증가시킨다. 마지막으로 Array 클래스에 정의된 [] 연산자 다중정의를 호출하여 인덱스 i의 원소의 참조자(left-value)를 반환하여 값을 대입할 수 있게 한다.

input : GrowableArray 인스턴스의 접근할 인덱스를 나타내는 정수형 변수 i

output : 접근한 해당 인덱스의 원소의 참조자(left-value)

(2) T GrowableArray<T>::operator [](int i) const

입력 인자로 전달받은 인덱스 i가 기존 배열의 인덱스 범위 내에 있는 지 확인한다. 기존 배열의 인덱스 범위를 벗어날 경우 위에서 설명한 GrowableArray 함수에 인덱스 i를 전달해 호출하여 해당 인덱스에 접근 가능하도록 배열의 크기를 증가시킨다. 마지막으로 Array 클래스에 정의된 [] 연산자 다중정의를 호출하여 인덱스 i의 원소의 값을 반환한다.(right-value) (1)의 연산자 다중정의와는 const 선언으로 구분된다.

input : GrowableArray 인스턴스의 접근할 인덱스를 나타내는 정수형 변수 i

output : 접근한 해당 인덱스의 원소의 값(right-value)

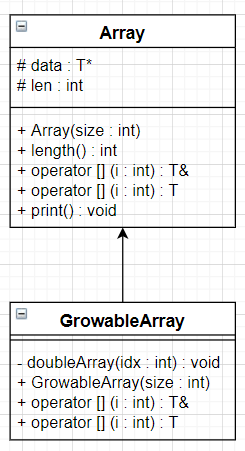
doubleArray 구현 부분(GrowableArray.h 파일의 일부분)

|  |
| --- |
| template <class T>  void GrowableArray<T>::doubleArray(int idx) {  T\* temp = new T[idx \* 2];  for(int i = 0; i < Array<T>::len; i++)  temp[i] = Array<T>::data[i];  for(int i = Array<T>::len; i < idx \* 2; i++)  temp[i] = 0;  delete[] Array<T>::data;  Array<T>::data = temp;  Array<T>::len = idx \* 2;  } |

사용한 자료구조 및 알고리즘

파라미터적 다형성을 구현하여 임의의 자료형 T의 1차원 배열 역할을 하는 Array를 구현한다. T형 포인터 변수 data를 이용해 배열을 동적 할당받고, [] 연산자를 다중정의하여 배열의 특정 인덱스에 값을 대입하고 값을 얻어오는 것을 가능하게 한다. 구현된 Array 클래스를 상속받는 GrowableArray 클래스를 작성한다. 이미 정해진 배열의 길이와 인덱스의 범위가 있을 때, 접근하려는 인덱스가 범위를 초과할 경우, 접근하려는 인덱스의 접근을 보장하기 위해서 배열의 최대 길이를 접근하려는 인덱스의 두 배로 증가시킨다. 기존 배열의 인덱스에 해당하는 부분은 새로 증가한 배열에 그대로 복사하고, 새로 추가된 인덱스의 원소는 0으로 초기화 후 접근하여 사용한다. GrowableArray 인스턴스의 접근하려는 인덱스가 기존 배열의 인덱스 범위에 해당할 경우 배열을 접근 인덱스의 두 배로 증가하는 함수 doubleArray를 호출한다. 해당 함수에서 기존 배열의 접근 방식을 그대로 사용하고 범위를 넘어갈 경우 T형 포인터 변수 temp로 해당 인덱스 크기의 2배 배열을 선언 후 기존 배열 data를 temp에 복사 후 data를 해제하여 메모리 누수를 방지한다. 마지막으로 포인터 변수 data에 temp를 대입하면 data가 새로 증가한 배열을 가리킨다.

프로그램의 구조도



Array 클래스와 GrowableArray 클래스의 관계를 나타낸 구조도이다. GrowableArray 클래스는 Array 클래스를 상속하고 기본적인 멤버 변수와 length() 함수, print() 함수는 상속받아 그대로 사용하지만 [] 연산자 다중 정의를 자체적으로 재정의해서 사용하고 자체적인 생성자와 자체적인 멤버 함수 doubleArray를 가진다. 생성자 GrowableArray는 기반 클래스 Array의 생성자를 호출한다. 재정의된 [] 연산자 다중정의는 접근하려는 인덱스가 기존 배열의 범위를 벗어날 경우 에러 메시지를 출력하는 Array 클래스와는 다른 처리를 하기 위해 재정의한다. 기존 배열의 범위를 벗어날 경우 GrowableArray 클래스 자체적으로 가지고 있는 doubleArray 멤버함수를 호출하여 배열의 크기를 충분히 증가시킨 후 Array 클래스의 연산자 다중정의를 호출하여 어떤 음이 아닌 정수 인덱스에도 접근 가능하도록 한다. 배열의 크기를 증가시켜주는 doubleArray 함수는 클래스 내부에서만 사용되며 외부에서 호출할 필요가 없고 호출해서도 안되므로 private로 선언한다.

6. 결론

파라미터적 다형성을 이용하면 같은 기능을 가졌지만 대상 자료형이 다른 비슷한 클래스를 여러 번 작성할 필요 없이 템플릿 클래스로 작성할 수 있다. 템플릿 클래스를 이용하면 중복되는 코드를 작성할 필요가 없어 코드 작성 시간이 획기적으로 줄어들고 코드의 재사용과 유지보수가 쉬워진다.

파라미터적 다형성을 구현한 템플릿 클래스를 작성할 경우 그렇지 않은 클래스를 작성할 때와는 다르게 클래스의 선언과 정의를 하나의 파일에 같이 포함시켜야 한다. 위에서 작성한 실습의 LinkedList 클래스와 Stack 클래스는 각각 linkedlist.h 파일과 stack.h 파일에 선언과 정의가 모두 들어있으며 과제의 Array 클래스와 GrowableArray 클래스는 각각 Array.h 파일과 GrowableArray.h 파일에 선언과 정의가 모두 들어 있다.

파라미터적 다형성을 구현한 두 클래스가 상속관계를 맺을 때 파생 클래스에서 기반 클래스의 멤버를 호출하려면 기반 클래스를 명시한 후 해당 멤버를 호출해야 한다.

기반 클래스의 포인터 변수와 기반 클래스에 정의된 가상 함수, 파생 클래스에 재정의된 가상 함수를 이용하여 서브타입 다형성을 구현할 수 있다. 기반 클래스의 포인터 변수는 기반 클래스의 인스턴스는 물론 파생 클래스의 인스턴스도 가리킬 수 있다. 기반 클래스에 선언된 함수가 파생 클래스에도 그대로 적용 가능할 경우 상속 받은 그대로 사용할 수 있고, 파생 클래스에 맞게 수정이 필요할 경우, 기반 클래스에 해당 함수를 가상 함수로 선언하고 파생 클래스에서 해당 클래스를 재정의한다. 기반 클래스 포인터 변수로 해당 함수를 호출하면 포인터 변수가 가리키는 인스턴스에 따라 해당 클래스에 정의 또는 재정의된 가상함수가 호출되어 같은 메시지에 대해서도 다른 함수가 호출되는 서브타입 다형성이 구현되었다.