20203324 게임공학과 최종현

```
#include <iostream>
#include <algorithm>

using namespace std;
struct singly_ll_node
{
    int data; // 이 칸에 저장될 실제 정수 데이터
    singly_ll_node* next; // 다음 노드을 가리키는 포인터.
    // 다음 칸이 없으면 nullptr이 된다.
};


class singly_ll // 단일 연결 리스트 전체를 관리하는 클래스
{
public:
    using node = singly_ll_node; // singly_ll_node 를 이제 'node' 라고 짧게 쓸 수 있다.
    using node_ptr = node*; // singly_ll_node* 를 이제 'node_ptr' 이라고 짧게 쓸 수 있다.


private:
    node_ptr head;
    // 리스트의 가장 첫 번째 노드를 가리키는 포인터. 리스트가 비어있으면 nullptr

public:
    // 리스트의 맨 앞에 새로운 값을 추가하는 함수
    void push_front(int val)
    {
        // 1.새로운 노드를 만든다. data에는 val 값을 넣고,
        // next는 현재 head가 가리키는 곳을 가리키게 한다.
        // 만약 리스트가 비어있었다면 head는 nullptr이었을 것이고,
        // new node->next도 nullptr이 된다.



        auto new_node = new node{ val,NULL };
        if (head != NULL)
        {
            new_node->next = head;
        }
        // 2. 리스트의 시작점(head)이 방금 만든 새 노드를 가리키도록 변경한다.
        head = new_node;
    }
    // 리스트의 맨 앞 노드를 제거하는 함수
    void pop_front()
    {
        // 1. 현재 맨 앞 노드(삭제될 노드)의 주소를 first라는 임시 포인터에 저장한다.
        auto first = head;
        // 2. 리스트가 비어있지 않은지 확인한다. (head가 nullptr이 아닌지)
        if (head)
        {
            head = head->next;
            // 3. head를 현재 두 번째 노드(first->next)로 변경하여, 리스트의 시작점을 다음 노드로 옮긴다.
            delete first;
            // 4. 아까 저장해둔 이전 head 노드(first)를 메모리에서 삭제한다.
        }
    }
```

```
// 리스트를 순회(하나씩 방문)하기 위한 이터레이터(반복자) 클래스이다. (for 루프 등에서 사용)
// 이터레이터는 리스트의 특정 노드를 가리키는 역할을 한다.
struct singly_LL_iterator
{
private:
    node_ptr ptr; // 현재 이터레이터가 가리키고 있는 노드의 주소를 저장하는 포인터

public:
    // 생성자: 특정 노드 포인터(p)를 받아서 이터레이터를 만든다.
    singly_LL_iterator(node_ptr p) :ptr(p) {}

    // 역참조 연산자 (*): 이터레이터가 가리키는 노드의 data 값에 접근하게 해준다. (값을 읽거나 변경)
    int& operator*() { return ptr->data; }
    // 현재 가리키는 노드의 포인터를 반환하는 함수
    node_ptr get() { return ptr; }

    // 전위 증가 연산자 (++it): 이터레이터를 다음 노드로 이동시키고, 이동된 '후'의 이터레이터를 반환한다.
    singly_LL_iterator& operator++() // 선행 증가
    {
        ptr = ptr->next; // 내부 포인터를 다음 노드로 이동
        return *this;   // 변경된 이터레이터 자신을 반환
    }

    // 후위 증가 연산자 (it++): 이터레이터를 다음 노드로 이동시키지만, 이동하기 '전'의 이터레이터를 반환한다.
    singly_LL_iterator operator++(int) // 후행 증가 (int는 전위와 구분하기 위한 더미 파라미터)
    {
        singly_LL_iterator result = *this; // 현재 상태(변경 전)를 복사해서 저장
        ++(*this); // 전위 증가 연산자를 호출하여 실제로 다음 노드로 이동
        return result; // 저장해둔 변경 전 상태를 반환
    }

        // 비교 연산자 (==): 두 이터레이터가 같은 노드를 가리키는지 확인한다.
        friend bool operator==(const singly_LL_iterator& left,
            const singly_LL_iterator& right)
        {
            return left.ptr == right.ptr; // 내부 포인터가 같으면 같은 이터레이터로 취급
        }

        // 비교 연산자 (!=): 두 이터레이터가 다른 노드를 가리키는지 확인한다.
        friend bool operator!=(const singly_LL_iterator& left,
            const singly_LL_iterator& right)
        {
            // return !(left == right); 와 같음
            return left.ptr != right.ptr;
        }

};
    // 리스트의 시작(첫 번째 노드)을 가리키는 이터레이터를 반환하는 함수 (for 루프에서 사용)
    singly_LL_iterator begin() { return singly_LL_iterator(head); }
    // 리스트의 끝(마지막 노드 다음, 즉 nullptr)을 가리키는 이터레이터를 반환하는 함수 (for 루프에서 사용)
    singly_LL_iterator end() { return singly_LL_iterator(nullptr); } // 리스트의 끝은 nullptr로 표현

    // const 버전: 리스트의 내용을 변경하지 않는 경우에 사용되는 begin/end 함수
    singly_LL_iterator begin() const { return singly_LL_iterator(head); }
    singly_LL_iterator end() const { return singly_LL_iterator(nullptr); }

    // 기본 생성자: 컴파일러가 자동으로 만들어주는 것을 사용한다.
    singly_ll() = default;

    // 복사 생성자: 다른 리스트(other)를 받아서 똑같은 내용을 가진 새 리스트를 만든다. (깊은 복사)
    singly_ll(const singly_ll& other) : head(NULL) // 새 리스트의 head를 NULL로 초기화
```

```
singly_ll(const singly_ll& other) : head(NULL) // 새 리스트의 head를 NULL로 초기화
{
    // 1. 복사할 원본 리스트(other)가 비어있지 않은지 확인한다.
    if (other.head)
    {
        // 2. 새 리스트의 첫 번째 노드를 임시로 만든다 (데이터는 0, next는 nullptr).
        head = new node{ 0,NULL };
        // 3. 새 리스트에서 현재 마지막으로 추가된 노드를 가리킬 포인터 cur (처음엔 head)
        auto cur = head;
        // 4. 원본 리스트를 순회할 이터레이터 'it' (원본의 첫 번째 노드에서 시작)
        auto it = other.begin();
        // 5. 무한 루프 시작 (내부에서 break로 탈출)
        while (true)
        {
            // 6. 현재 원본 노드(it)의 데이터를 새 리스트의 현재 노드(cur)에 복사한다. (첫 노드의 0을 덮어씀)
            cur->data = *it;
            // 7. 원본 리스트의 다음 노드를 가리키는 임시 이터레이터 tmp를 만든다.
            auto tmp = it;
            ++tmp;
            // 8. 만약 원본 리스트의 다음 노드가 끝(nullptr)이라면, 복사가 끝난 것이므로 루프를 탈출한다.
            if (tmp == other.end())
                break;
            // 9. 새 리스트에 다음 노드를 만들 준비를 한다. 임시 노드(데이터 0)를 만들어 현재 노드(cur)의 next에 연결한다.
            cur->next = new node{ 0,NULL };
            // 10. 새 리스트의 현재 노드(cur)를 방금 만든 다음 노드로 이동한다.
            cur = cur->next;
            // 11. 원본 리스트의 이터레이터(it)도 다음 노드로 이동한다.
            it = tmp;
        }
    }
}
```

```
// 초기화 리스트 생성자: 중괄호 {1, 2, 3} 형태로 리스트를 만들 수 있게 해준다.
singly_ll(const initializer_list<int>& ilist) :head(NULL) // 새 리스트의 head를 nullptr로 초기화
{
    // 1. 초기화 리스트({1, 2, 3})를 뒤에서부터(rbegin) 거꾸로(rend) 순회한다. (즉, 3, 2, 1 순서)
    for (auto it = rbegin(ilist); it != rend(ilist); it++)
        // 2. 각 원소를 리스트의 맨 앞에 추가(push_front)한다,
        // (3 추가 -> 2 추가 -> 1 추가 => 결과적으로 1 -> 2 -> 3 순서의 리스트가 만들어짐)
        push_front(*it);
}
```

```cpp
// 초기화 리스트 생성자를 이용해 sll 리스트 생성: 1 -> 2 -> 3
singly_ll sll = { 1, 2, 3 };
// push_front 함수를 이용해 맨 앞에 0 추가: 0 -> 1 -> 2 -> 3
sll.push_front(0);
sll.push_front(1);
sll.push_front(2);
cout << "첫 번째 리스트 : ";
// 범위 기반 for 루프: sll 리스트의 처음(begin)부터 끝(end)까지 순회
// 내부적으로 begin(), end(), ++, * 연산자를 사용
for (auto i : sll)
    // 각 노드의 데이터(i)를 출력하고 한 칸 띄움
    cout << i << " "; // 출력 0 1 2 3
cout << endl;

// 복사 생성자를 이용해 sll 리스트를 sll2 리스트로 깊은 복사
// sll2는 sll과 똑같은 내용(0 -> 1 -> 2 -> 3)을 가지지만, 서로 다른 메모리 공간을 사용
auto sll2 = sll;
// sll2 리스트의 맨 앞에 -1 추가: -1 -> 0 -> 1 -> 2 -> 3
sll2.push_front(-1);
sll2.push_front(-2);
sll2.push_front(-3);
cout << "첫 번째 리스트를 복사한 후, 맨 앞에 -1을 추가";
for (auto i : sll2)
    cout << i << ' '; // 출력 -1 0 1 2 3
cout << endl;

cout << "깊은 복사 후 첫 번째 리스트 : ";
// sll2를 변경한 것이 sll에 영향을 주지 않았는지 확인 (깊은 복사 확인)
for (auto i : sll)
    cout << i << ' '; // 출력 0 1 2 3
```

```
첫 번째 리스트 : 2 1 0 1 2 3
첫 번째 리스트를 복사한 후, 맨 앞에 -1을 추가-3 -2 -1 2 1 0 1 2 3
깊은 복사 후 첫 번째 리스트 : 2 1 0 1 2 3
```

```cpp
#include<iostream>
#include<list>
using namespace std;
int main()
{
    // int(정수) 타입의 데이터를 저장하는 표준 list 객체 list1을 생성.
    // list는 내부적으로 '이중 연결 리스트'로 구현되어 있어, 데이터의 삽입/삭제가 특정 위치에서 효율적임.
    // 중괄호 {} 를 사용하여 초기값 {1, 2, 3, 4, 5} 로 리스트를 초기화한다.
    list<int> list1 = { 1,2,3,4,5 };
    // list1의 맨 뒤(back)에 값 6을 추가(push)한다.
    list1.push_back(6); // 1 2 3 4 5 6
    // list1의 특정 위치에 값을 삽입(insert)한다,
    // list1.begin()은 리스트의 가장 첫 번째 원소(1)를 가리키는 위치(이터레이터)이다.
    // next(list1.begin())은 list1.begin()의 바로 다음 위치, 즉 두 번째 원소(2)를 가리킨다.
    // insert 함수는 지정된 위치 바로 앞에 값을 삽입한다.
    // 따라서 두 번째 원소 2 앞에 0을 삽입한다.
    list1.insert(next(list1.begin()), 0); // 1 0 2 3 4 5 6

    // list1의 맨 끝 바로 앞에 값 7을 삽입한다.
    // list1.end()는 마지막 원소 다음 위치를 가리키는 특별한 이터레이터이다.
    // end() 앞에 삽입하는 것은 결국 리스트의 맨 뒤에 추가하는 것과 같다.
    list1.insert(list1.end(), 7); // 1 0 2 3 4 5 6 7

    // list1의 맨 뒤(back) 원소를 제거(pop)합니다.가장 마지막의 7이 제거된다.
    list1.pop_back(); // 1 0 2 3 4 5 6

    cout << "삽입 & 삭제 후 리스트 : ";
    for (auto i : list1)
        cout << i << " ";
    return 0;
}
```

```
삽입 & 삭제 후 리스트 : 1 0 2 3 4 5 6
```

```cpp
#include <iostream>
#include <vector>
#include <initializer_list> // 중괄호 초기화 {1, 2, 3} 를 사용하기 위해 필요
using namespace std;
class intList {

private:
    // vector<int> 를 사용하여 정수들을 저장
    // vector는 필요에 따라 자동으로 크기가 조절되는 편리한 배열
    vector<int> data;

public:
    // 1. 생성자 (객체가 처음 만들어질 때 호출되는 함수들)
    // 기본 생성자: 아무런 초기값 없이 빈 리스트를 만든다.
    // 특별한 코드가 없어도 내부의 vector가 자동으로 비어있는 상태로 초기화한다.
    intList() {};

    // 초기화 리스트 생성자: 중괄호 { } 안에 값들을 넣어 리스트를 생성할 수 있게 한다.
    intList(initializer_list<int> init_list) : data(init_list)
    {
        // 멤버 초기화 리스트(: data(init_list))를 사용해서
        // vector를 전달받은 초기값들로 바로 초기화한다.
    }

    // 2. 멤버 함수 (객체가 할 수 있는 기능들)

    // 리스트의 맨 뒤에 정수 값을 추가하는 함수
    void add_element(int value)
    {
        // vector의 push_back 함수를 사용하여 맨 뒤에 요소를 추가.
        data.push_back(value);
    }
```

```
// 리스트에 저장된 모든 정수를 출력하는 함수
// 함수 뒤에 'const'를 붙이면 이 함수는 멤버 변수의 내용을
// 변경하지 않는다는 약속이다.
void print_list() const
{
    if (data.empty())
    { // vector가 비어있는지 확인
        cout << "리스트가 비어있습니다.";
    }
    else
    {
        // 범위 기반 for 루프를 사용하여 vector의 모든 요소를 출력.
        for (int element : data)
        {
            cout << element << " ";
        }
    }
    cout << endl; // 줄바꿈
}

// 리스트에 저장된 정수의 개수를 반환하는 함수
size_t get_size() const {
    // vector의 size 함수를 사용하여 저장된 요소의 개수를 반환.
    // size_t는 보통 부호 없는 정수(unsigned int) 타입.
    return data.size();
}

// 3. 소멸자 (객체가 메모리에서 사라지기 직전에 호출되는 함수)
// 이 클래스는 vector를 사용하고 vector가 메모리 관리를 해주므로,
// 직접 new/delete를 쓰지 않았다면 특별히 작성할 코드는 없다.
~intList() {
```

```cpp
int main() {

    // 1. 초기화 리스트 생성자를 사용하여 List1 객체 생성
    intList List1 = { 10, 20, 30 };
    List1.print_list(); // 생성된 리스트 내용 출력
    cout << "List1의 크기: " << List1.get_size() << endl;

    // 2. add_element 멤버 함수를 사용하여 요소 추가
    List1.add_element(40);
    List1.add_element(50);
    List1.print_list(); // 변경된 리스트 내용 출력
    cout << "List1의 크기: " << List1.get_size() << endl;

    // 3. 기본 생성자를 사용하여 비어있는 List2 객체 생성
    intList List2;
    List2.print_list(); // 비어있는 리스트 출력
    cout << "List2의 크기: " << List2.get_size() << endl;

    // 4. 비어있는 myList2에 요소 추가

    List2.add_element(5);
    List2.add_element(15);
    List2.add_element(25);

    List2.print_list(); // 변경된 리스트 내용 출력
    cout << "myList2의 크기: " << List2.get_size() << endl;
    return 0;
}
```

```
10 20 30
List1의 크기: 3
10 20 30 40 50
List1의 크기: 5
리스트가 비어있습니다.
List2의 크기: 0
5 15 25
myList2의 크기: 3
intList 객체가 소멸되었습니다.
intList 객체가 소멸되었습니다.
```