20203324 게임공학과 최종현
4주차

```cpp
#include <iostream>
#include <vector>

using namespace std;
struct Node
{
    int data;
    Node* next;
};

class IntaddNodeFirst
{
private:
    Node* head;
public:
    // 생성자
    IntaddNodeFirst() : head(nullptr)
    {

    }
    // 연결 리스트에 첫 번째에 새 노드를 추가하는 함수
    void add(int data)
    {
        // 새 노드 생성
        Node* newNode = new Node;
        newNode->data = data;

        // 새 노드를 첫 번째 노드로 설정하고 기존 첫 번째 노드를 새 노드의 다음 노드로 연결
        newNode->next = head;
        head = newNode;
    }
```

```
    void showprint()
    {
        Node* now = head;
        while (now != nullptr)
        {
            cout << now->data << endl;
            now = now->next;
        }
    }
};
```

```
// 연결 리스트의 첫 번째에 노드 추가
cout << "처음 부분에 Node를 추가하기" << endl;
IntaddNodeFirst List;
List.add(1);
List.add(2);
List.add(3);

List.showprint();
```

```
처음 부분에 Node를 추가하기
3
2
1
```

```
class IntaddNodeLast
{
private:
    Node* head;

public:
    IntaddNodeLast() : head(nullptr)
    {

    }

    void addNode(int data)
    {
        Node* newNode = new Node;
        newNode->data = data;
        // 새 노드는 마지막 노드이므로 다음 노드를 가리키는 포인터는 nullptr
        newNode->next = nullptr;

        // 연결 리스트가 비어있으면 새 노드를 첫 번째 노드로 설정
        if (head == nullptr)
        {
            head = newNode;
            return;
        }

        // 연결 리스트의 마지막 노드를 찾아서 새 노드를 연결
        Node* nowNode = head;
        while (nowNode->next != nullptr)
        {
            nowNode = nowNode->next;
        }
        // 마지막 노드의 다음 노드를 새 노드로 설정
        nowNode->next = newNode;

    }
```

```
void showprint()
{
    Node* nowNode = head;
    while (nowNode != nullptr)
    {
        cout << nowNode->data << endl;
        nowNode = nowNode->next;
    }
}
```

```
// 연결 리스트의 마지막에 노드 추가
cout << "마지막 부분에 Node를 추가하기" << endl;
List1.addNode(1);
List1.addNode(2);
List1.addNode(3);

List1.showprint();
```

```
마지막 부분에 Node를 추가하기
1
2
3
```

```
class DeleteFirstNode
{
private:
    Node* head;

public:
    DeleteFirstNode() : head(nullptr)
    {

    }

    // 연결 리스트의 첫 번째에 새 노드를 추가하는 함수
    void addNode2(int data)
    {
        Node* newNode = new Node;
        newNode->data = data;
        newNode->next = head;
        head = newNode;
    }

    // 연결 리스트의 첫 번째 노드를 삭제하는 함수
    void DeleteFirst()
    // 연결 리스트가 비어있으면 삭제할 노드가 없으므로 함수 종료
    {
        if (head == nullptr)
            return;
        // 삭제할 첫 번째 노드를 임시 포인터에 저장
        Node* temp = head;

        // 두 번째 노드를 첫 번째 노드로 설정
        head = head->next;

        delete temp;
    }
```

```
    void showNode()
    {
        Node* nowNode = head;
        while (nowNode != nullptr)
        {
            cout << nowNode->data << endl;
            nowNode = nowNode->next;
        }
    }
:
```

```
// 연결 리스트의 첫 번째 노드 삭제
cout << "첫 번째 Node 삭제" << endl;
DeleteFirstNode List2;

List2.addNode2(1);
List2.addNode2(2);
List2.addNode2(3);

List2.DeleteFirst();

List2.showNode();
```

```
첫 번째 Node 삭제
2
1
```

```
class DeleteLastNode
{
private:
    Node* head;
public:
    DeleteLastNode() : head(nullptr)
    {

    }

    // 연결 리스트 마지막에 새 노드를 추가하는 함수
    void addNode3(int data)
    {
        Node* newNode = new Node;
        newNode->data = data;
        newNode->next = nullptr;

        if (head == nullptr)
        {
            head = newNode;
            return;
        }

        Node* nowNode = head;
        while (nowNode->next != nullptr)
        {
            nowNode = nowNode->next;
        }
        nowNode->next = newNode;

    }
```

```
// 연결 리스트의 마지막 노드를 삭제하는 함수
void DeleteNode()
{
    // 연결 리스트가 비어있으면 삭제할 노드가 없으므로 함수 종료
    if (head == nullptr)
    {
        return;
    }

    // 연결 리스트에 노드가 하나만 있으면 첫 번째 노드를 삭제
    if (head->next == nullptr)
    {
        delete head;
        head = nullptr;
        return;
    }

    // 마지막 노드와 그 이전 노드를 찾음
    Node* nowNode = head;
    Node* previous = nullptr;
    while (nowNode->next != nullptr)
    {
        previous = nowNode;
        nowNode = nowNode->next;
    }

    // 마지막 노드 삭제(메모리 해제)
    delete nowNode;

    // 이전 노드의 다음 노드를 nullptr로 설정하여 연결 리스트의 끝을 표시
    previous->next = nullptr;
}
```

```
void showNode()
{
    Node* nowNode = head;
    while (nowNode != nullptr)
    {
        cout << nowNode->data << endl;
        nowNode = nowNode->next;
    }
}
```

```
// 연결 리스트의 마지막 노드 삭제
cout << "마지막 Node 삭제" << endl;

DeleteLastNode List3;
List3.addNode3(1);
List3.addNode3(2);
List3.addNode3(3);

List3.DeleteNode();

List3.showNode();
```

```
마지막 Node 삭제
1
2
```

```cpp
struct Node2 {
    int data; // 데이터
    Node2* left; // 왼쪽 가지
    Node2* right; // 오른쪽 가지
};

// 나무 탐색 클래스
class Traverser {
public:
    // 나무 만들기
    Node2* createTree() {
        // 루트 노드는 데이터 1을 가지고, 왼쪽 자식은 데이터 2, 오른쪽 자식은 데이터 3을 가진다.
        Node2* root = new Node2{ 1, new Node2{2, nullptr, nullptr}, new Node2{3, nullptr, nullptr} };
        return root; // 만들어진 나무의 루트 노드를 반환한다.
    }

    // 나무 탐색 함수 (중간부터 방문)
    // 중위 순회(inorder traversal)를 사용하여 나무를 탐색한다.
    // 왼쪽 자식 -> 현재 노드 -> 오른쪽 자식 순서로 방문한다.
    void traverse(Node2* node, vector<int>& result) {
        if (node == nullptr) {
            return; // 현재 노드가 비어있으면 탐색을 종료.
        }
        traverse(node->left, result); // 왼쪽 자식 노드를 재귀적으로 탐색한다
        result.push_back(node->data); // 현재 노드의 데이터를 결과 벡터에 저장한다.
        traverse(node->right, result); // 오른쪽 자식 노드를 재귀적으로 탐색한다.
    }
```

```cpp
// 나무 탐색 결과 확인 함수
// 실제 탐색 결과와 예상 결과를 비교하여 탐색이 제대로 되었는지 확인한다.
bool CheckTraverse(Node2* root, const vector<int>& expectedResult) {
    vector<int> realResult; // 실제 탐색 결과를 저장할 벡터
    traverse(root, realResult); // 실제 탐색을 수행하고 결과를 저장한다.

    // 실제 결과와 예상 결과의 크기가 다르면 탐색 실패
    if (realResult.size() != expectedResult.size()) {
        return false;
    }
    // 실제 결과와 예상 결과를 순서대로 비교
    for (size_t i = 0; i < realResult.size(); ++i) {
        if (realResult[i] != expectedResult[i]) {
            return false; // 하나라도 다르면 탐색 실패
        }
    }

    return true; // 모든 요소가 일치하면 탐색 성공
}
```

```cpp
// traverse 함수의 기능 구현
Traverser traverser;
Node2* tree = traverser.createTree(); // 나무 만들기

// 예상 탐색 결과 (중간부터 방문)
vector<int> expectedResult = { 2, 1, 3 };

// 탐색 결과 확인
if (traverser.CheckTraverse(tree,expectedResult)) {
    cout << "탐색 성공!" << endl;
}
else {
    cout << "탐색 실패!" << endl;
}
```

```
탐색 성공!
```

```
class addAfterN
{

};

class deleteAfterN
{

};

class getsize
{

};

class findNumber
{

};
```

연습문제 3

```cpp
#include <string> // string 클래스를 사용하기 위한 헤더 파일 포함
#include <iostream> // cout, endl 등 입출력 기능을 사용하기 위한 헤더 파일 포함
#include <forward_list> // forward_list 컨테이너를 사용하기 위한 헤더 파일 포함

using namespace std;
// 시민 정보를 저장하는 구조체 정의
struct citizen
{
    string name; // 시민의 이름을 저장하는 문자열 변수
    int age; // 시민의 나이를 저장하는 정수형 변수
};

// citizen 구조체 객체를 출력 스트림에 출력하는 연산자 오버로딩
// citizen 객체를 "[이름,나이]" 형식으로 출력하도록 정의
ostream& operator<<(ostream& os, const citizen& c)
{
    return(os << "[" << c.name << "," << c.age << "]");
}
// 투표권이 있는 시민 정보(19세 이상)를 출력
cout << "투표권이 있는 시민들: ";
// for 루프를 사용하여 citizens의 모든 요소에 접근
for (const auto& c : citizens)
    cout << c << " "; // 각 시민 정보를 출력
cout << endl;

// 추가 코드 작성 16세는 내년에도 투표권이 없기 때문에 완전 제외
citizens.remove_if([](const citizen &c)
    {
        return (c.age == 16);

    });


// remove_if 함수를 사용하여 나이가 18세가 아닌 시민을 리스트에서 제거
citizens.remove_if([](const citizen &c)
    {
        return (c.age == 18);
        // 나이가 18세이면 true를 반환하여 해당 요소를 제거
    });
```

```cpp
int main()
{
    // citizen 구조체를 요소로 가지는 forward_list 컨테이너 생성 및 초기화
    // forward_list는 단방향 연결 리스트로, 요소의 삽입/삭제가 효율적임
    forward_list<citizen> citizens =
    {
        {"Kim",22},{"Lee",25},{"Park",19},{"Jin",16}
    };
    // citizens forward_list를 복사하여 citizen_copy forward_list 생성 (깊은 복사)
    // forward_list의 복사 생성자는 깊은 복사를 수행하여 요소의 내용까지 복사함
    auto citizen_copy = citizens; // 깊은 복사

    // 전체 시민 정보를 출력
    cout << "전체 시민들 : ";
    // for 루프를 사용하여 citizens의 모든 요소에 접근
    for (const auto& c : citizens)
        cout << c << " "; // 각 시민 정보를 출력
    cout << endl;

    // remove_if 함수를 사용하여 나이가 19세 미만인 시민을 리스트에서 제거
    // 람다 표현식을 사용하여 제거 조건을 정의
    citizens.remove_if([](const citizen &c)
        {
            // 나이가 19세보다 작으면 리스트에서 제거한다.
            return (c.age < 19);
        });
// remove_if 함수를 사용하여 나이가 18세가 아닌 시민을 리스트에서 제거
citizens.remove_if([](const citizen &c)
    {
        return (c.age == 18);
        // 나이가 18세이면 true를 반환하여 해당 요소를 제거
    });

// 내년에 투표권이 생기는 시민 정보(19세)를 출력
// citizen_copy는 citizens의 복사본이므로, citizens의 변경 사항이 반영되지 않음
cout << "내년에 투표권이 생기는 시민들 : ";
// for 루프를 사용하여 citizen_copy의 모든 요소에 접근
// 추가 주석 작성 16세가 내년에 투표권이 생기는 시민들에 출력이 되는것이 이상하여
// 16세이면 내년에도 투표권이 없기 때문에 제외를 해버리고
// citizen_copy를 통해서 원본을 불러오는게 아닌 삭제된 정보를 출력하게 함.
for (const auto& c : citizens) // 각 시민 정보를 출력
    cout << c << " ";
cout << endl;
```

```
전체 시민들 : [Kim,22] [Lee,25] [Park,19] [Jin,16]
투표권이 있는 시민들: [Kim,22] [Lee,25] [Park,19]
내년에 투표권이 생기는 시민들 : [Kim,22] [Lee,25] [Park,19]
```

연습문제 4

```cpp
#include <iostream> // 표준 입출력 스트림을 사용하기 위한 헤더 파일 포함
#include <forward_list> // 단방향 연결 리스트인 forward_list 컨테이너를 사용하기 위한 헤더 파일 포함
#include <vector> // 동적 배열인 vector 컨테이너를 사용하기 위한 헤더 파일 포함

using namespace std;
int main()
{
    // 문자열을 요소로 가지는 vector 컨테이너 vec 생성 및 초기화
    vector<string> vec =
    {
        "Lewis Hamilton","Lewis Hamilton","Nico Roseberg",
        "Sebastian Vettel","Lewis Hamilton","Sebastian Vettel",
        "Sebastian Vettel","Sebastian Vettel","Fernando Alonso"
    };

    // vector의 시작 반복자(iterator)를 it 변수에 할당 (상수 시간 복잡도)
    // 반복자는 컨테이너의 요소를 가리키는 포인터와 유사한 객체
    auto it = vec.begin(); // 상수 시간
    // 가장 최근 우승자 출력 (vector의 첫 번째 요소)
    cout << "가장 최근 우승자 : " << *it << endl;

    it += 8; // 상수 시간
    // 반복자를 8칸 뒤로 이동 (상수 시간 복잡도)
    // vector는 임의 접근 반복자를 제공하므로, 상수 시간 내에 임의의 위치로 이동 가능
    // 8년 전 우승자 출력 (vector의 8번째 요소)
    cout << "8년 전 우승자 : " << *it << endl;
```

```cpp
it += 8; // 상수 시간
// 반복자를 8칸 뒤로 이동 (상수 시간 복잡도)
// vector는 임의 접근 반복자를 제공하므로, 상수 시간 내에 임의의 위치로 이동 가능
// 8년 전 우승자 출력 (vector의 8번째 요소)
cout << "8년 전 우승자 : " << *it << endl;

advance(it, -3); // 상수 시간
// 반복자를 3칸 앞으로 이동 (상수 시간 복잡도)
// vector는 양방향 이동을 지원하므로, 음수 값을 사용하여 앞으로 이동 가능
// 3년 뒤 우승자 출력 (vector의 5번째 요소)
cout << "그후 3년 뒤 우승자 : " << *it << endl;


// vector의 모든 요소를 복사하여 forward_list 컨테이너 fwd 생성 및 초기화
// forward_list는 단방향 연결 리스트이므로, 순차적인 접근만 가능
forward_list<string> fwd(vec.begin(), vec.end());

// forward_list의 시작 반복자를 it1 변수에 할당
auto it1 = fwd.begin();

// 가장 최근 우승자 출력 (forward_list의 첫 번째 요소)
cout << "가장 최근 우승자 : " << *it1 << endl;

// 반복자를 5칸 뒤로 이동 (선형 시간 복잡도)
// forward_list는 단방향 연결 리스트이므로, 순차적으로 이동해야 함
advance(it1, 5); // 선형 시간
// 5년 전 우승자 출력 (forward_list의 5번째 요소)
cout << "5년 전 우승자 : " << *it1 << endl;
```

```cpp
advance(it1, 2);
// 7년 전 우승자 출력(forward_list의 7번째 요소)
cout << "7년 전 우승자 : " << *it1 << endl;
cout << endl;
// forward_list는 순방향으로만 이동할 수 있으므로
// advance(it1,-2)는 에러가 발생한다.
// vector는 양방향으로 접근이 가능
// forward_list는 단방향이라서 순차적인 접근이 가능

// vector의 모든 우승자를 순차적으로 출력
cout << "vector의 우승자 목록:" << endl;
for (const auto& winner : vec) {
    cout << winner << endl;
}
cout << endl;
// forward_list의 모든 우승자를 순차적으로 출력
cout << "forward_list의 우승자 목록:" << endl;
for (const auto& winner : fwd) {
    cout << winner << endl;
}

return 0;
```

```
가장 최근 우승자 : Lewis Hamilton
8년 전 우승자 : Fernando Alonso
그후 3년 뒤 우승자 : Sebastian Vettel
가장 최근 우승자 : Lewis Hamilton
5년 전 우승자 : Sebastian Vettel
7년 전 우승자 : Sebastian Vettel

vector의 우승자 목록:
Lewis Hamilton
Lewis Hamilton
Nico Roseberg
Sebastian Vettel
Lewis Hamilton
Sebastian Vettel
Sebastian Vettel
Sebastian Vettel
Fernando Alonso

forward_list의 우승자 목록:
Lewis Hamilton
Lewis Hamilton
Nico Roseberg
Sebastian Vettel
Lewis Hamilton
Sebastian Vettel
Sebastian Vettel
Sebastian Vettel
Fernando Alonso
```