P1.S12

작성일시	@2024년 11월 4일 오전 4:27
강의 번호	C++ 언리얼
유형	강의
복습	☐
날짜	@2024년 11월 4일

vector #1

```
#include <iostream>
using namespace std;
#include <vector>

// 오늘의 주제 : vector #1

int main()
{
    // STL (Standard Template Library)
    // 프로그래밍할 때 필요한 자료구조/알고리즘들을
    // 템플릿으로 제공하는 라이브러리


    // 컨테이너(Container) : 데이터를 저장하는 객체 (자료구조 Data St

    // vector (동적 배열)
    // - vector의 동작 원리 (size/capacity)
    // - 중간 삽입/삭제
    // - 처음/끝 삽입/삭제
    // - 임의 접근

    // 배열
    // 배열을 다 쓰면 추가적으로 늘리는 작업이 안됨
    const int MAX_SIZE = 10;
    int arr[MAX_SIZE] = {};
```

```
for (int i = 0; i < MAX_SIZE; i++)
    arr[i] = i;

//for (int i = 0; i < MAX_SIZE; i++)
//  cout << arr[i] << endl;


// 동적 배열
// 매우 매우 중요한 개념 -> 어떤 마법을 부렸길래 배열을 '유동적으로'

// 1) (여유분을 두고) 메모리를 할당한다
// 2) 여유분까지 꽉 찼으면, 메모리를 증설한다

// Question1) 여유분을 얼만큼이 적당할까?
// Question2) 증설을 얼만큼 해야 할까?
// Question3) 기존의 데이터를 어떻게 처리할까?


vector<int> v;
// 만들자 마자 크기와 초기값 지정도 가능
vector<int> v2(1000, 1);
vector<int> v3 = v2;

// size (실제 사용 데이터 개수)
// 1 2 3 4 5 6 7 ...
//
// capacity (여유분을 포함한 용량 개수)
// 1 2 3 4 6 9 13 19 28 ...  1.5배씩
// C++에서 capacity는 줄어들지 않는다

for (int i = 0; i < 1000; i++)
{
    v.push_back(100);
    cout << v.size() << " '" << v[i] << "' " << v.capacit
}

// reserve를 이용해서 처음부터 capacity를 정해줄 수 있음
```

```
    v.reserve(1000);
    cout << v.size() << " " << v.capacity() << endl;

    // resize를 이용해서 처음부터 데이터를 차지해줄수있음
    v.resize(1000);
    cout << v.size() << " " << v.capacity() << endl;

    // 데이터 날려주기
    v.clear();

    // 임시로 만들어준 vector<int>()는 swap을 통해 v를 vector가 가지
    // 깡통 vector<int>()의 정보를 v에 넣어준다
    vector<int>().swap(v);

    cout << v.size() << " " << v.capacity() << endl;


    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    // 마지막 데이터 빼기
    v.pop_back();
    // 맨 마지막 데이터 보기
    cout << v.back() << endl;
    // 맨 처음 데이터 보기
    cout << v.front() << endl;


    return 0;
}
```

vector #2~3 (면접 단골)

```
#include <iostream>
using namespace std;
#include <vector>
```

```
// 오늘의 주제 : vector #2

// Iterator는 쉽게 생각하면 아래와 같은 구조
//class MyIterator
//{
//
//};
//
//class MyVector
//{
//
//public:
//  typedef MyIterator iterator;
//};


int main()
{
    // STL (Standard Template Library)
    // 프로그래밍할 때 필요한 자료구조/알고리즘들을
    // 템플릿으로 제공하는 라이브러리


    // 컨테이너(Container) : 데이터를 저장하는 객체 (자료구조 Data St

    // vector (동적 배열)
    // - vector의 동작 원리 (size/capacity)
    // - 중간 삽입/삭제
    // - 처음/끝 삽입/삭제
    // - 임의 접근

    // 반복자(Iterator) : 포인터와 유사한 개념. 컨테이너의 원소(데이터)
    vector<int> v(10);

    // v.size()는 unsigned int를 뱉어줘서, int i = 0; 이렇게 하면
    // 따라서 형태에 맞춰서 해주려고 vector<int>::size_type i = 0;를
    // using은 C++에 생긴 typedef와 유사한 문법
    for (vector<int>::size_type i = 0; i < v.size(); i++)
```

```
        v[i] = i;

    //vector<int>::iterator it;
    //MyVector::iterator it2;

    /*int* ptr;

    it = v.begin();
    ptr = &v[0];*/

    // 어떻게 포인터와 유사하게 사용하냐? -> * 연산자 오버로딩을 한것

    /*cout << (*it) << endl;
    cout << (*ptr) << endl;

    it++;
    ++it;
    ptr++;
    ++ptr;

    it--;
    --it;
    ptr--;
    --ptr;

    it += 2;
    it = it - 2;*/


    vector<int>::iterator itBegin = v.begin();
    vector<int>::iterator itEnd = v.end();
    // itEnd 같은 경우 컴파일해서 메모리를 보면 마지막꺼 다음 쓰레기값에
    // 따라서 저기서 뭘 조작하면 안됨

    // C++11에서는 vector<int>::iterator 이걸 auto로 퉁치는게 등장
    // 짤팁) ++it가 it++보다 조금더 성능이 좋다
    // 더 복잡해보이는데?
    // iterator는 vector뿐 아니라, 다른 컨테이너에도 공통적으로 있는 개
```

```
    // 다른 컨테이너는 v[i]와 같은 인덱스 접근이 안될 수 있음
    for (vector<int>::iterator it = v.begin(); it != v.end();
    {
        cout << (*it) << endl;
    }

    int* ptrBegin = &v[0]; // v.begin()._Ptr;
    int* ptrEnd = ptrBegin + 10; // v.end()._Ptr;
    for (int* ptr = ptrBegin; ptr != ptrEnd; ++ptr)
        cout << (*ptr) << endl;


    // const int*
    // 읽기만 하고 수정하지는 않으려면
    //vector<int>::const_iterator cit1 = v.cbegin();

    // 역방향
    for (vector<int>::reverse_iterator it = v.rbegin(); it !=
    {
        cout << (*it) << endl;
    }



    // - 중간 삽입/삭제        (효율 BAD)
    // - 처음/끝 삽입/삭제 (처음꺼는 BAD / 끝 GOOD)
    // - 임의 접근(Random Access)


    // 면접 단골 질문(C++ 기본기)

    // vector = 동적 배열 = 동적으로 커지는 배열 = 배열
    // 원소가 하나의 메모리 블록에 연속하게 저장된다!!

    // [                     ]
    // [0][1][2][3][4][ ][ ]    기본형태
    //
    // [0][1][5][2][3][4][ ]    5삽입하려면 뒤에것들 한칸씩 뒤로 이동
```

```
    //
    // [0][1][3][4][ ][ ][ ]    2를 삭제 후 뒤에것들 앞으로 이동시켜

    // 끝에 삽입 삭제하는것만 효율이 좋아서 v.push_back과 같이 back 관

    // i번째 데이터는 어디 있습니까?
    // 데이터들이 하나로 뭉쳐있어서 바로 접근해줄 수 있다는 장점
    v[2] = 3;

    // 중간에 삽입, 삭제하는 기능
    /*vector<int>::iterator insertIt = v.insert(v.begin() + 2
    vector<int>::iterator eraseIt1 = v.erase(v.begin() + 2);
    vector<int>::iterator eraseIt2 = v.erase(v.begin() + 2, v

    // 쭉~ 스캔을 하면서, 3이라는 데이터가 있으면 일괄 삭제하고 싶다
    // 삭제할때는 내부에서 어떤일이 일어나는지 정확하게 파악하고 코드를 짜
    // 주의가 필요함
    for (vector<int>::iterator it = v.begin(); it != v.end();
    {
        int data = *it;
        if (data == 3)
        {
            it = v.erase(it);
        }
        else
        {
            ++it;
        }
    }

    return 0;
}
```

vector #4

```
#include <iostream>
using namespace std;
#include <vector>

// 오늘의 주제 : vector

template<typename T>
class Iterator
{
public:
    Iterator() : _ptr(nullptr)
    {

    }

    Iterator(T* ptr) : _ptr(ptr)
    {

    }

    Iterator& operator++()
    {
        _ptr++;
        return *this;
    }

    // (a++)++ 과 같은 형태는 불가능함 따라서 복사라는 것을 알 수 있음
    // 그래서 &가 안붙음
    Iterator operator++(int)
    {
        Iterator temp = *this;
        _ptr++;
        return temp;
    }

    Iterator& operator--()
    {
        _ptr--;
```

```
        return *this;
    }

    Iterator operator--(int)
    {
        Iterator temp = *this;
        _ptr--;
        return temp;
    }

    Iterator operator+(const int count)
    {
        Iterator temp = *this;
        temp._ptr += count;
        return temp;
    }

    Iterator operator-(const int count)
    {
        Iterator temp = *this;
        temp._ptr -= count;
        return temp;
    }

    bool operator==(const Iterator& right)
    {
        return _ptr == right._ptr;
    }

    bool operator!=(const Iterator& right)
    {
        return !(*this == right);
    }

    T& operator*()
    {
        return *_ptr;
    }
```

```
public: // 멤버 변수가 public으로 열려있다
    T* _ptr;
};

template<typename T>
class Vector
{
public:

    Vector() : _data(nullptr), _size(0), _capacity(0)
    {

    }

    ~Vector()
    {
        if (_data)
            delete[] _data;
    }

    void push_back(const T& val)
    {
        if (_size == _capacity)
        {
            // 증설 작업
            int newCapacity = static_cast<int>(_capacity * 1.
            if (newCapacity == _capacity)
                newCapacity++;

            reserve(newCapacity);
        }

        // 데이터 저장
        // [0][1][2], _size = 3, val = 5
        // [0][1][2][5]
        _data[_size] = val;
```

```
        // 데이터 개수 증가
        _size++;
    }

    void reserve(int capacity)
    {
        _capacity = capacity;

        T* newData = new T[_capacity];

        // 데이터 복사
        for (int i = 0; i < _size; i++)
            newData[i] = _data[i];

        // 기존에 있던 데이터 날린다
        if (_data)
            delete[] _data;

        // 교체
        _data = newData;
    }

    // & : 레퍼런스로 뱉어주는 이유는 v[i] = 1; 이렇게 데이터를 밀어넣는
    // 레퍼런스를 이용해서 실제값을 건드릴 수 있어야해서임
    T& operator[](const int pos) { return _data[pos]; }

    int size() { return _size; }
    int capacity() { return _capacity; }

public: // iterator 관련 부분
    typedef Iterator<T> iterator;

    void clear() { _size = 0; }
    iterator begin() { return iterator(&_data[0]); }
    iterator end() { return begin() + _size; }

private:
    T* _data;
```

```
    int _size;
    int _capacity;
};

int main()
{
    Vector<int> v;

    v.reserve(100);

    for (int i = 0; i < 100; i++)
    {
        v.push_back(i);
        cout << v.size() << " " << v.capacity() << endl;
    }

    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i] << endl;
    }

    cout << "----------------------" << endl;

    for (Vector<int>::iterator it = v.begin(); it != v.end();
    {
        cout << (*it) << endl;
    }

    v.clear();

    return 0;
}
```

list #1~2(면접 단골)

```
#include <iostream>
using namespace std;
#include <vector>
```

```
#include <list>

// 오늘의 주제 : list

// vector : 동적 배열
// [                  ]


// [ data(4) next(4/8) ]
class Node
{
public:
    Node* _next; // 노드안에 노드가 있어서 헷갈릴수있지만 주소값을 가지
    Node* _prev;
    int _data;
};


// 단일 / 이중 / 원형
// list : 연결 리스트

// [1]     ->   [2]    ->   [3]  ->      [4] ->  [5]
// [1]    <->   [2]   <->   [3] <->      [4]<->  [5] <->  [ _
// [1]    <->   [2]   <->   [3] <->      [4]<->  [5] <->

int main()
{
    // list (연결 리스트)
    // - list의 동작 원리
    // - 중간 삽입/삭제 (GOOD/GOOD)
    // - 처음/끝 삽입/삭제 (GOOD/GOOD)
    // - 임의 접근 (i번째 데이터는 어디 있습니까?) (NO)

    // 중간 데이터 찾는게 느린데 어떻게 중간 삽입삭제가 빠르냐?
    // 이터레이터로 기억하게 해뒀다가 나중에 삭제하게 된다면 중간 삽입삭제

    list<int> li;
```

```
    list<int>::iterator itRemember;

    for (int i = 0; i < 100; i++)
    {
        if (i == 50)
        {
            itRemember = li.insert(li.end(), i);
        }
        else
        {
            li.push_back(i);
        }
        li.push_back(i);
    }

    //li.push_front(10);
    int size = li.size();
    //li.capacity(); // 없음

    int first = li.front();
    int last = li.back();

    //li[3] = 10; // 없음

    list<int>::iterator itBegin = li.begin();
    list<int>::iterator itEnd = li.end();

    //list<int>::iterator it2 = itBegin + 10; // 지원 안함

    // 더미노드 확인용
    //list<int>::iterator ptrTest1 = --itBegin; // 크래시
    //list<int>::iterator ptrTest2 = --itEnd; // 가능
    //list<int>::iterator ptrTest3 = ++itEnd; // 크래시

    int* ptrBegin = &(li.front());
    int* ptrEnd = &(li.back());
```

```
    for (list<int>::iterator it = li.begin(); it != li.end();
    {
        cout << *it << endl;
    }

    //li.insert(itBegin, 100);
    //li.erase(li.begin()); // 중간값도 삭제시켜줄수 있는 erase
    //li.pop_front();
    //li.remove(10); // !!! value와 같은 것들 일괄 삭제


    // * 임의 접근이 안 된다
    // * 중간 삽입/삭제 빠르다 (?)
    // 50번 인덱스에 있는 데이터를 삭제!
    /*list<int>::iterator it = li.begin();
    for (int i = 0; i < 50; i++)
        ++it;

    li.erase(it);*/

    li.erase(itRemember);

    return 0;
}
```

list #3

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>

// 오늘의 주제 : list #3

template<typename T>
class Node
```

```
{
public:
    Node() : _next(nullptr), _prev(nullptr), _data(T())
    {

    }

    Node(const T& value) : _next(nullptr), _prev(nullptr), _d
    {

    }

public:
    Node*   _next;
    Node*   _prev;
    T       _data;
};

template<typename T>
class Iterator
{
public:
    Iterator() : _node(nullptr)
    {

    }

    Iterator(Node<T>* node) : _node(node)
    {

    }

    Iterator& operator++()
    {
        _node = _node->_next;
        return *this;
    }
```

```
    Iterator operator++(int)
    {
        Iterator<T> temp = *this;
        _node = _node->_next;
        return temp;
    }

    Iterator& operator--()
    {
        _node = _node->_prev;
        return *this;
    }

    Iterator operator--(int)
    {
        Iterator<T> temp = *this;
        _node = _node->_prev;
        return temp;
    }

    T& operator*()
    {
        return _node->_data;
    }

    bool operator==(const Iterator& right)
    {
        return _node == right._node;
    }

    bool operator!=(const Iterator& right)
    {
        return _node != right._node;
    }

public:
    Node<T>* _node;
};
```

```
// [1] <-> [2] <-> [3] <-> [ header ] <->
template<typename T>
class List
{
public:
    List() : _size(0)
    {
        _header = new Node<T>();
        _header->_next = _header;
        _header->_prev = _header;

    }

    ~List()
    {
        while (_size > 0)
            pop_back();

        delete _header;
    }

    void push_back(const T& value)
    {
        AddNode(_header, value);
    }

    void pop_back()
    {
        RemoveNode(_header->_prev);
    }

    // [1] <-> [2] <-> [before] <-> [4] <-> [ header ] <->
    // [1] <-> [2] <-> [node] <-> [before] <-> [4] <-> [ head
    Node<T>* AddNode(Node<T>* before, const T& value)
    {
        Node<T>* node = new Node<T>(value);
```

```
        Node<T>* prevNode = before->_prev;
        prevNode->_next = node;
        node->_prev = prevNode;

        node->_next = before;
        before->_prev = node;

        _size++;

        return node;
    }

    // [1] <-> [2] <-> [node] <-> [4] <-> [ header ] <->
    // [1] <-> [2] <-> [4] <-> [ header ] <->
    Node<T>* RemoveNode(Node<T>* node)
    {
        Node<T>* prevNode = node->_prev;
        Node<T>* nextNode = node->_next;

        prevNode->_next = nextNode;
        nextNode->_prev = prevNode;

        delete node;

        _size--;

        return nextNode;
    }

    int size() { return _size; }

public:
    typedef Iterator<T> iterator;

    iterator begin() { return iterator(_header->_next); }
    iterator end() { return iterator(_header); }

    iterator insert(iterator it, const T& value)
```

```
    {
        Node<T>* node = AddNode(it._node, value);
        return iterator(node);
    }

    iterator erase(iterator it)
    {
        Node<T>* node = RemoveNode(it._node);
        return iterator(node);
    }

public:
    Node<T>* _header;
    int _size;
};


int main()
{
    List<int> li;

    List<int>::iterator eraseIt;

    for (int i = 0; i < 10; i++)
    {
        if (i == 5)
        {
            eraseIt = li.insert(li.end(), i);
        }
        else
        {
            li.push_back(i);
        }
    }

    li.pop_back();

    li.erase(eraseIt);
```

```
    for (List<int>::iterator it = li.begin(); it != li.end();
    {
        cout << (*it) << endl;
    }


    return 0;
}
```

deque

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>

// 오늘의 주제 : deque


// vector : 동적 배열
// [          ]

// list : 이중 연결 리스트
// [] <-> [] <-> []

// deque : double-ended queue 데크
// [     ]
// [     ]
// [     ]


int main()
{
    // 시퀀스 컨테이너 (Sequence Container)
    // 데이터가 삽입 순서대로 나열되는 형태
    // vector list deque
```

```
    // vector와 마찬가지로 배열 기반으로 동작
    // 다만 메모리 할당 정책이 다르다

    // vector
    // [1 1 1]

    // deque
    // [    3 3]    1동
    // [1 1 1 2]    2동
    // [2      ]    3동

    vector<int> v(3, 1);
    deque<int> dq(3, 1);

    v.push_back(2);
    v.push_back(2);

    dq.push_back(2);
    dq.push_back(2);

    dq.push_front(3);
    dq.push_front(3);

    // - deque의 동작 원리
    // - 중간 삽입/ 삭제 (BAD / BAD)
    // - 처음/끝 삽입/삭제 (GOOD / GOOD)
    // - 임의 접근 (GOOD)

    // 중간에 데이터가 비어있으면 안된다~

    return 0;
}
```

map

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>

// 오늘의 주제 : map

class Player
{
public:
    Player() : _playerId(0) {}
    Player(int playerId) : _playerId(playerId) {}


public:
    int _playerId;
};


int main()
{
    // vector, list의 치명적 단점
    // -> 원하는 조건에 해당하는 데이터를 빠르게 찾을 수 [없다]


    // 연관 컨테이너

    // map : 균형 이진 트리 (AVL)
    // - 노드 기반

    class Node
    {
    public:
        Node* _left;
        Node* _right;
        // DATA
```

```
    pair<int, Player*> _data;
    //int       _key;
    //Player* _value;
};

srand(static_cast<unsigned int>(time(nullptr)));

// (Key, Value)
map<int, int> m;

// insert 두번 연속으로 해보기
// pair<map<int, int>::iterator, bool> 이런 타입을 뱉어서 성공
// 혹시라도 동일한 키에 넣으려고 하면 무시된다
/*pair<map<int, int>::iterator, bool> ok;
ok = m.insert(make_pair(1, 100));
ok = m.insert(make_pair(1, 200));*/

// 10 만명
for (int i = 0; i < 100000; i++)
{
    m.insert(pair<int, int>(i, i * 100));
}

// 5만명 퇴장
for (int i = 0; i < 50000; i++)
{
    int randomValue = rand() % 50000;

    // Erase By Key
    m.erase(randomValue);
}

// Q) ID = 1만인 Player 찾고 싶다!
// A) 매우 빠르게 찾을 수 있음

// erase를 두번 연속 해보기
// 삭제 성공했으면 1, 성공 못했으면 0을 뱉어준다
/*unsigned int count = 0;
```

```
    count = m.erase(10000);
    count = m.erase(10000);*/

    // map 순회

    for (map<int, int>::iterator it = m.begin(); it != m.end(
    {
        pair<const int, int>& p = (*it);
        int key = it->first;
        int value = it->second;

        cout << key << " " << value << endl;
    }

    // 없으면 추가, 있으면 수정

    map<int, int>::iterator findIt = m.find(10000);
    if (findIt != m.end())
    {
        findIt->second = 200;
    }
    else
    {
        m.insert(make_pair(10000, 200));
    }

    // 없으면 추가, 있으면 수정 v2
    m[5] = 500;

    m.clear();
    // [] 연산자 사용할 때 주의
    // 대입을 하지 않더라도 (Key/Value) 형태의 데이터가 추가된다!
    for (int i = 0; i < 10; i++)
    {
        cout << m[i] << endl; // 이렇게 사용하는 순간 데이터가 없었다
    }
```

```
    // 넣고      (insert, [])
    // 빼고      (erase)
    // 찾고      (find, [])
    // 반복자    (map::iterator) (*it)pair<key, value>&

    return 0;
}
```

set, multimap, multiset

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>


// 오늘의 주제 : set, multimap, multiset

int main()
{
    // (Key = Value)
    set<int> s;

    // 넣고
    // 빼고
    // 찾고
    // 순회하고

    // 넣고
    s.insert(10);
    s.insert(20);
    s.insert(30);
    s.insert(40);
    s.insert(50);
    s.insert(60);
```

```
    s.insert(70);
    s.insert(80);
    s.insert(90);
    s.insert(100);

    // 빼고
    s.erase(40);

    // 찾고
    set<int>::iterator findIt = s.find(50);
    if (findIt == s.end())
    {
        cout << "못 찾음" << endl;
    }
    else
    {
        cout << "찾음" << endl;
    }

    // 순회하고
    for (set<int>::iterator it = s.begin(); it != s.end(); ++
    {
        cout << (*it) << endl;
    }

    cout << "----------------------------" << endl;

    multimap<int, int> mm;

    // 넣고
    mm.insert(make_pair(1, 100));
    mm.insert(make_pair(1, 200));
    mm.insert(make_pair(1, 300));
    mm.insert(make_pair(2, 400));
    mm.insert(make_pair(2, 500));

    //mm[1] = 500; 이게 멀티맵에서는 안됨
```

```
    // 빼고
    //unsigned int count = mm.erase(1); // 3개 삭제했다고 알려줌

    // 찾고
    /*multimap<int, int>::iterator itFind = mm.find(1);
    if (itFind != mm.end())
        mm.erase(itFind);*/
    // 찾은거 처음꺼만 삭제해줌

    pair<multimap<int, int>::iterator, multimap<int, int>::it
    itPair = mm.equal_range(1); // 1의 시작 끝 둘다 알려줌

    for (multimap<int, int>::iterator it = itPair.first; it !=
    {
        cout << it->first << " " << it->second << endl;
    }

    multimap<int, int>::iterator itBegin = mm.lower_bound(1);
    multimap<int, int>::iterator itEnd = mm.upper_bound(1); /

    for (multimap<int, int>::iterator it = itBegin; it != itE
    {
        cout << it->first << " " << it->second << endl;
    }

    cout << "----------------------------" << endl;

    multiset<int> ms;

    // 넣고
    ms.insert(100);
    ms.insert(100);
    ms.insert(100);
    ms.insert(200);
    ms.insert(200);

    // 찾고
    multiset<int>::iterator findIt2 = ms.find(100);
```

```
    pair<multiset<int, int>::iterator, multiset<int, int>::it
    itPair2 = ms.equal_range(100);

    for (multiset<int, int>::iterator it = itPair2.first; it
    {
        cout << *it << endl;
    }

    multiset<int, int>::iterator itBegin2 = ms.lower_bound(10
    multiset<int, int>::iterator itEnd2 = ms.upper_bound(100)

    for (multiset<int, int>::iterator it = itBegin2; it != it
    {
        cout << *it << endl;
    }

    return 0;
}
```

연습문제

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>


// 오늘의 주제 : 연습 문제

int main()
{
    srand(static_cast<unsigned int>(time(nullptr)));

    vector<int> v;
```

```
    for (int i = 0; i < 100; i++)
    {
        int num = rand() % 100;
        v.push_back(num);
    }

    // Q1) number라는 숫자가 벡터에 체크하는 기능 (bool, 첫 등장 iter

    {
        int number = 50;

        bool found = false;
        vector<int>::iterator it;

        for (it = v.begin(); it != v.end(); ++it)
        {
            if (*it == number)
            {
                cout << "찾았음" << endl;
                cout << (*it) << endl;
                found = true;
                break;
            }

        }

        if (it == v.end())
        {
            cout << "못 찾았음" << endl;
        }
    }

    // Q2) 11로 나뉘는 숫자가 벡터에 있는지 체크하는 기능 (bool, 첫 등

    {
        bool found = false;
        vector<int>::iterator it;
```

```
    for (it = v.begin(); it != v.end(); ++it)
    {
        if ((*it) != 0 && (*it % 11) == 0)
        {
            cout << "찾았음" << endl;
            cout << (*it) << endl;
            found = true;
            break;
        }

    }

    if (it == v.end())
    {
        cout << "못 찾았음" << endl;
    }
}

// Q3) 홀수인 숫자의 개수는? (count)

{
    int count = 0;

    for (int i = 0; i < v.size(); i++)
    {
        if ((v[i] % 2) == 1)
        {
            count++;
        }
    }

    cout << count << "개" << endl;
}

// Q4) 벡터에 들어가 있는 모든 숫자들에 3을 곱해주세요!

{
```

```
        vector<int>::iterator it;

        for (it = v.begin(); it != v.end(); ++it)
        {
            (*it) *= 3;

            cout << (*it) << endl;
        }
    }

    return 0;
}
```

알고리즘

```
#include <iostream>
using namespace std;
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
#include <algorithm>

// 오늘의 주제 : 알고리즘

int main()
{
    // 자료구조 (데이터를 저장하는 구조)
    // 알고리즘 (데이터를 어떻게 사용할 것인가?)

    // find
    // find_if

    // count
    // count_if
    // all_of
    // any_of
```

```
    // none_of
    // for_each
    // remove
    // remove_if

    srand(static_cast<unsigned int>(time(nullptr)));

    vector<int> v;

    for (int i = 0; i < 100; i++)
    {
        int num = rand() % 100;
        v.push_back(num);
    }

    // Q1) number라는 숫자가 벡터에 체크하는 기능 (bool, 첫 등장 iter

    {
        cout << "Q1) number라는 숫자가 벡터에 체크하는 기능 (bool,

        int number = 50;

        bool found = false;
        vector<int>::iterator it;

        for (it = v.begin(); it != v.end(); ++it)
        {
            if (*it == number)
            {
                cout << "찾았음" << endl;
                cout << (*it) << endl;
                found = true;
                break;
            }

        }

        if (it == v.end())
```

```
        {
            cout << "못 찾았음" << endl;
        }

        vector<int>::iterator itFind = find(v.begin(), v.end(
        if (itFind == v.end())
        {
            // 못찾았음
            cout << "못 찾았음" << endl;
        }
        else
        {
            cout << "찾았음" << endl;
        }
    }

    // Q2) 11로 나뉘는 숫자가 벡터에 있는지 체크하는 기능 (bool, 첫 등

    {
        cout << "Q2) 11로 나뉘는 숫자가 벡터에 있는지 체크하는 기능 (b

        bool found = false;
        vector<int>::iterator it;

        for (it = v.begin(); it != v.end(); ++it)
        {
            if ((*it) != 0 && (*it % 11) == 0)
            {
                cout << "찾았음" << endl;
                cout << (*it) << endl;
                found = true;
                break;
            }

        }

        if (it == v.end())
        {
```

```
            cout << "못 찾았음" << endl;
        }

        struct CanDivideBy11
        {
            bool operator()(int n)
            {
                return(n % 11) == 0;
            }
        };

        vector<int>::iterator itFind = std::find_if(v.begin()
        if (itFind == v.end())
        {
            cout << "못 찾았음" << endl;
        }
        else
        {
            cout << "찾았음" << endl;
        }
    }

    // Q3) 홀수인 숫자의 개수는? (count)

    {
        cout << "Q3) 홀수인 숫자의 개수는? (count)" << endl;

        int count = 0;

        for (int i = 0; i < v.size(); i++)
        {
            if ((v[i] % 2) == 1)
            {
                count++;
            }
        }

        cout << count << "개" << endl;
```

```
    struct IsOdd
    {
        bool operator()(int n)
        {
            return (n % 2) != 0;
        }
    };

    int n = std::count_if(v.begin(), v.end(), IsOdd());

    // 모든 데이터가 홀수 입니까?
    int b1 = std::all_of(v.begin(), v.end(), IsOdd());
    // 홀수인 데이터가 하나라도 있습니까?
    int b2 = std::any_of(v.begin(), v.end(), IsOdd());
    // 모든 데이터가 홀수가 아닙니까?
    int b3 = std::none_of(v.begin(), v.end(), IsOdd());
}

// Q4) 벡터에 들어가 있는 모든 숫자들에 3을 곱해주세요!

{
    cout << "Q4) 벡터에 들어가 있는 모든 숫자들에 3을 곱해주세요!"

    vector<int>::iterator it;

    for (it = v.begin(); it != v.end(); ++it)
    {
        (*it) *= 3;

        cout << (*it) << endl;
    }

    struct MultiplyBy3
    {
        void operator()(int& n)
        {
            n = n * 3;
```

```
            }
        };

        std::for_each(v.begin(), v.end(), MultiplyBy3());
    }

    // 홀수인 데이터를 일괄 삭제
    {
        //for (vector<int>::iterator it = v.begin(); it != v.
        //{ // 데이터를 건드릴때는 ++it를 아래서 넣는게 낫다
        //  if ((*it % 2) != 0)
        //      it = v.erase(it);
        //  else
        //      ++it;
        //}

        v.clear();

        v.push_back(1);
        v.push_back(4);
        v.push_back(3);
        v.push_back(5);
        v.push_back(8);
        v.push_back(2);

        // 1 4 3 5 8 2

        //std::remove(v.begin(), v.end(), 4);

        struct IsOdd
        {
            bool operator()(int n)
            {
                return (n % 2) != 0;
            }
        };

        //vector<int>::iterator it = std::remove_if(v.begin()
```

```
        // 4 8 2 5 8 2

        /*template<class ForwardIt, class T = typename std::i
        ForwardIt remove(ForwardIt first, ForwardIt last, con
        {
            first = std::find(first, last, value);
            if (first != last)
                for (ForwardIt i = first; ++i != last;)
                    if (!(*i == value))
                        *first++ = std::move(*i);
            return first;
        }*/

        // 1 4 3 5 8 2
        // 4 4 3 5 8 2
        // 1 8 3 5 8 2
        // 1 8 2 5 8 2

        // 데이터를 날려주지는 않고, 쓸 데이터만 앞에 모아줌
        // 날릴 데이터의 시작 위치를 반환해준다

        //v.erase(it, v.end());

        // 따라서 한방에
        v.erase(remove_if(v.begin(), v.end(), IsOdd()), v.end
    }

    return 0;
}
```