13장 표준 템플릿 라이브러리

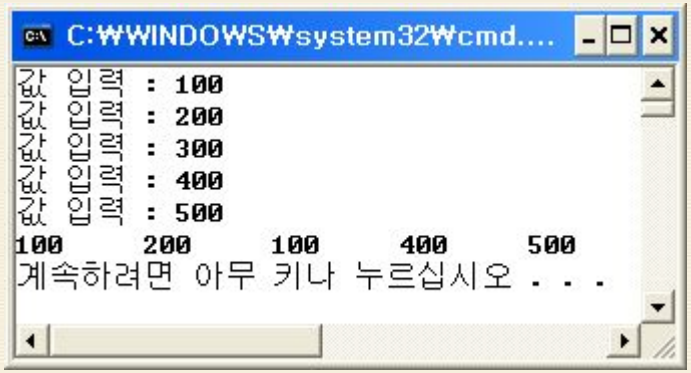
- STL 소개
- vector 클래스의 사용
- list 클래스의 사용
- 이터레이터의 이해
- 이터레이터의 사용
- 이터레이터의 종류
- 알고리즘의 이해
- 알고리즘의 종류
- 컨테이너 클래스의 종류

1. STL 소개

- 표준 템플릿 라이브러리(Standard Template Library)
  - 구성 요소
    - 컨테이너 클래스 : 자료구조 (stack, queue, vector, ...)
    - 이터레이터 : 모든 자료구조의 원소에 대한 동일한 접근 방법 제공
      - ✓ 포인터와 유사!
    - 알고리즘 : 함수 (sort, swap, find, ...)
  - 일반화 라이브러리(generic library)라고 불림
    - 기본적으로 객체지향 프로그래밍을 따르지 않음
    - sort라는 함수가 특정 자료 구조의 멤버 함수가 아니며 모든 자료구조에 대해 동일한 사용 방법을 제공함
- 설명 순서
  - vector 클래스와 list 클래스의 사용 방법
  - 이터레이터의 개념 및 필요성
  - vector 클래스와 list 클래스에 대한 알고리즘 적용 방법
  - 알고리즘과 컨테이너 클래스의 종류

2. vector 클래스의 사용

- vector
  - 1차원 배열
  - 수행 도중에 원소의 개수 변경 가능
  - <vector> 헤더 파일 포함
- 예 : 5개 원소를 가진 int형 배열




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

int main(void)
{
    int i;
    vector<int> intV(5);

    for (i = 0; i < 5; i++) {
        cout << "값 입력 : ";
        cin >> intV[i];
    }

    intV[2] = 100;

    for (i = 0; i < 5; i++) {
        cout << intV[i] << "₩t";
    }
    cout << endl;

    return 0;
}
```

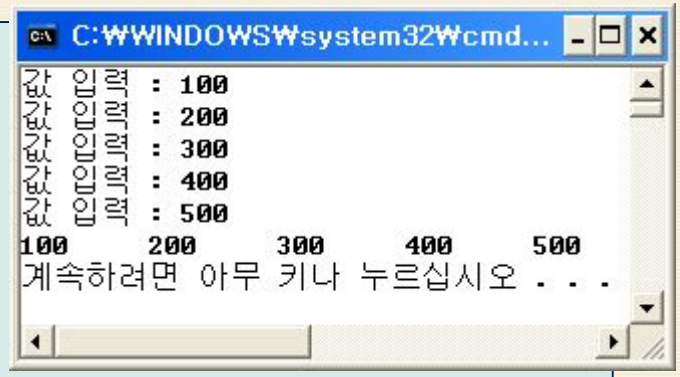
원소 5개인 int형 배열

일반 배열처럼 사용!

2. vector 클래스의 사용

- vector 클래스 객체 생성 방법
  - vector<int> intV(5); // 5개의 원소를 가진 int형 배열
  - vector<int> intV; // 0개의 원소를 가진 int형 배열???
  - vector<CPoint> pointV(3); // 3개의 원소를 가진 CPoint형 배열
- push_back 멤버 함수의 사용 : 마지막 원소 다음에 새로운 원소 추가

```
int main(void)
{
    int i, value;
    vector<int> intV;

    for (i = 0; i < 5; i++) {
        cout << "값 입력 : ";
        cin >> value;
        intV.push_back(value);   // 마지막 원소 뒤에 새로운 원소 추가
    }

    for (i = 0; i < 5; i++) {
        cout << intV[i] << "₩t"; // i번째 원소의 값 출력
    }
    cout << endl;

    return 0;
}
```



배열의 크기(원소 개수)를 알아내는 방법
size() 멤버 함수
intV.resize(intV.size() + 1); // 원소 개수를 1씩 증가

2. vector 클래스의 사용

- 예 : CPoint 클래스 객체를 원소로 갖는 vector 객체

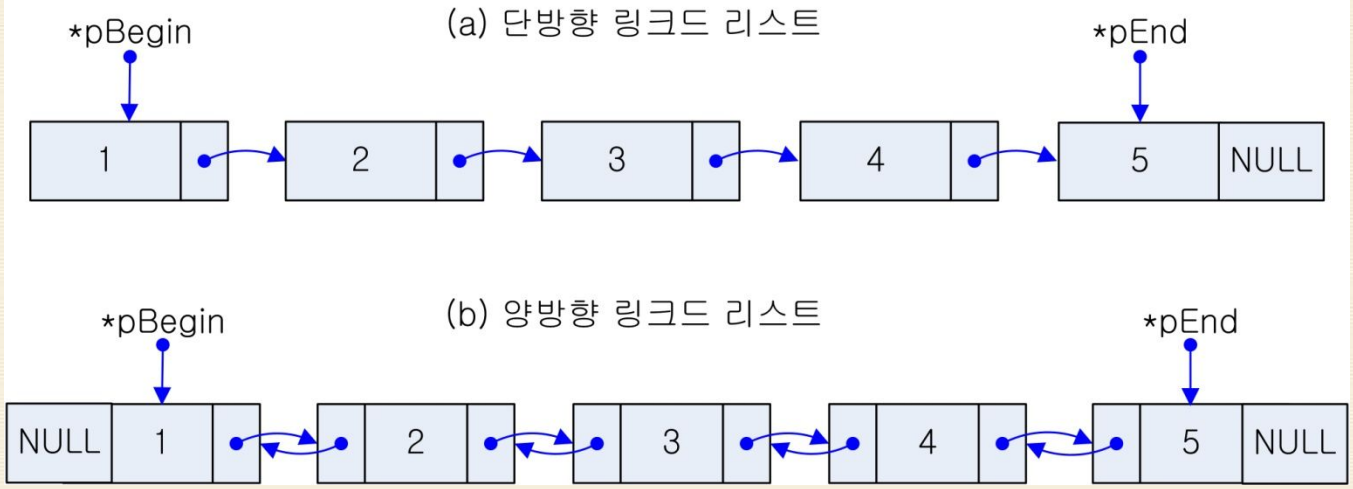
```
class CPoint{
private :
    int x, y;

public :
    CPoint(int a = 0, int b = 0) : x(a), y(b) { }
    void Print() { cout << "(" << x << ", " << y << ")" << endl;
};

int main(void)
{
    int i;
    vector<CPoint> cAry(5);      // 이 경우 디폴트 생성자 필요!

    for (i = 0; i < 5; i++)
        cAry.push_back(CPoint(i + 1, i + 1)); // 새로운 CPoint 객체 추가

    for (i = 0; i < 10; i++) // 총 10개의 원소 존재
        cAry[i].Print();

    return 0;
}
```

```
C:₩...
(0, 0)
(0, 0)
(0, 0)
(0, 0)
(0, 0)
(1, 1)
(2, 2)
(3, 3)
(4, 4)
(5, 5)
계속하려면 아
```

vector 클래스의 또 다른 멤버 함수들
- insert, erase, clear => 대부분 이터레이터와 함께 동작
→ **4**절에서 설명

3. list 클래스의 사용

⌗ 단방향 링크드 리스트와 양방향 링크드 리스트



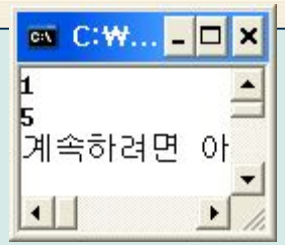
⌗ list 클래스

- 리스트를 쉽게 다룰 수 있도록 만든 컨테이너 클래스
- 마치 배열과 유사하게 사용 가능 (내부 구조는 리스트)
- <list> 헤더 파일 포함

3. list 클래스의 사용

list 클래스의 사용 예

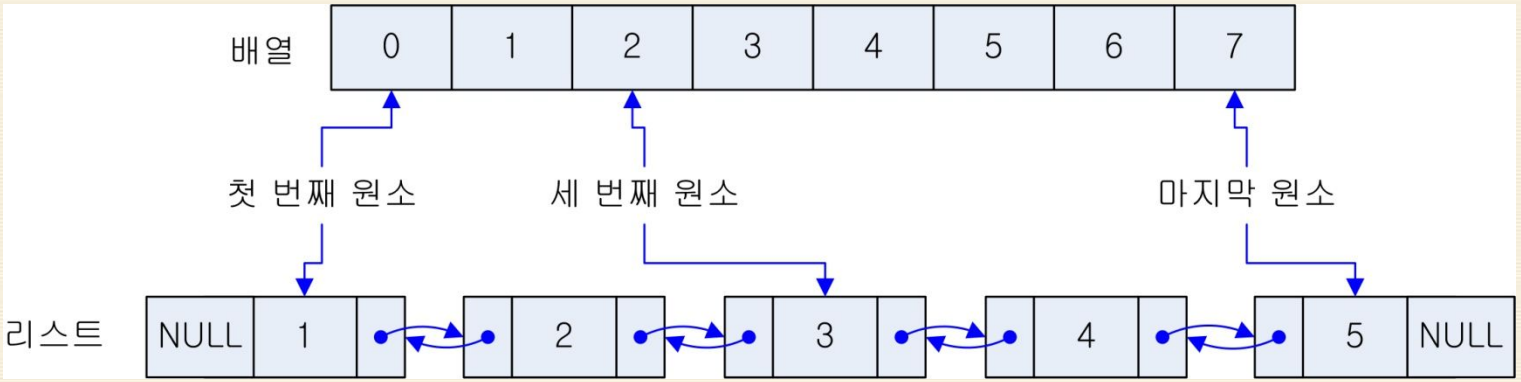
```
#include <iostream>
#include <list>
using namespace std;

int main(void)
{
    int i;
    list<int> MyList;                // int형 list 생성

    for (i = 0; i < 5; i++)
        MyList.push_back(i + 1);     // 마지막 원소 다음에 원소 삽입

    cout << MyList.front() << endl; // 첫 번째 원소
    cout << MyList.back() << endl;  // 마지막 원소

    return 0;
}
```



* list는 [] 연산자를 제공하고 있지 않음
  - 배열의 용도 : 임의 원소에 대한 빠른 접근 → [ ]
  - 리스트의 용도 : 임의 위치 사이의 빠른 삽입, 삭제
* 이터레이터를 사용한다면 유사하게 사용할 수 있음 → 다음 절

4. 이터레이터의 이해

⌗ 배열과 리스트의 모양



- 특정 원소를 가리키는 방법은?
  - ➢ 배열과 리스트에 동일한 방법 필요 ← 특정 알고리즘(함수)을 배열과 리스트에 동일한 방식으로 적용하기 위해서.

→ 이터레이터(iterator)

- ➢ 포인터와 유사
- ➢ vector의 경우 이터레이터와 포인터는 동일한 개념

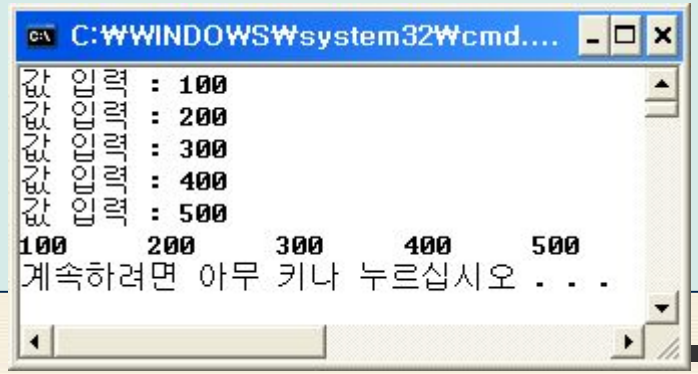
4. 이터레이터의 이해

- 예 : 포인터를 이용한 vector 원소의 값 처리

```
int main(void)
{
    int i;
    vector<int> intV(5);          // 5개 원소를 갖는 배열
    int *pV = &intV[0];           // 첫 번째 원소를 가리킴

    for (i = 0; i < 5; i++) {
        cout << "값입력: ";
        cin >> (*pV);
        pV++;                     // 다음 원소를 가리킴
    }

    pV = &intV[0];
    for (i = 0; i < 5; i++) {
        cout << *pV << "₩t";
        pV++;
    }
    cout << endl;

    return 0;
}
```

일반 포인터 변수를 사용하여 **vector** 객체의 원소 처리 가능

list 객체에 대해서는 일반 포인터로 다음 원소에 대한 접근 불가능 **(p++)**
→ **list**에 적합한 포인터 필요!



4. 이터레이터의 이해 : 원리 이해

- vector 클래스 및 vector 클래스의 이터레이터 클래스 구현
  - MyVector(편의상 5개 원소로 제한), VectorIterator

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

template <typename T>
class VectorIterator {
private :
    T *ptr;

public :
    VectorIterator(T *p = 0) : ptr(p) { }      // 초기화, 포인터로 주소 가리킴
    T &operator*() { return (*ptr); }          // 역참조 연산자
    void operator++(int) { ptr++; }            // 후위 증가 연산자
    void operator=(T *p) { ptr = p; }          // 대입 연산자
};

template <typename T>
class MyVector {
private :
    T ary[5];

public :
    typedef VectorIterator<T> iterator;        // iterator 이름으로 사용 가능
    T *begin() { return &ary[0]; }
};
```

내부적으로 포인터 포함

4. 이터레이터의 이해 : 원리 이해

- vector 클래스 및 vector 클래스의 이터레이터 클래스 구현 (계속)

```
int main(void)
{
    int i;
    MyVector<int> intV;
    MyVector<int>::iterator vIter(intV.begin()); // VectorIterator<int> 객체

    for (i = 0; i < 5; i++) {
        *vIter = i;
        vIter++;
    }

    vIter = intV.begin();
    for (i = 0; i < 5; i++) {
        cout << *vIter << endl;
        vIter++;
    }

    return 0;
}
```

기존 포인터의 사용 방법과 동일

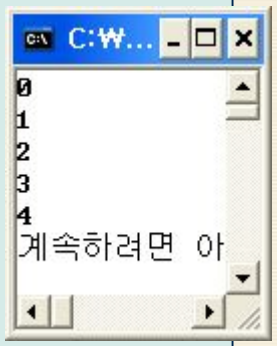
4. 이터레이터의 이해 : 원리 이해

- list 클래스 및 list 클래스의 이터레이터 클래스 구현
  - MyList(단방향 링크드 리스트로 가정), ListIterator

```
template <typename T>
struct Node {      // 하나의 노드 : 데이터 + 다음 노드에 대한 포인터
    T data;
    Node<T> *next;
    Node(T d, Node<T> *n = NULL) : data(d), next(n) { }
};

template <typename T>
class ListIterator {
private :
    Node<T> *ptr;  // 특정 노드에 대한 포인터

public :
    ListIterator(Node<T> *p = 0) : ptr(p) { }
    void operator++(int) { ptr = ptr->next; }
    T &operator*() { return ptr->data; }
    void operator=(Node<T> *p) { ptr = p; }
};
```

다음 원소

현재 원소의 값



4. 이터레이터의 이해 : 원리 이해

- list 클래스 및 list 클래스의 이터레이터 클래스 구현 (계속)

```
template <typename T>
class MyList {
private :
    Node<T> *start;
    Node<T> *last;

public :
    MyList() : start(NULL), last(NULL) { }
    typedef ListIterator<T> iterator;
    Node<T> *begin() { return start; }
    void push(T d) {                    // 새로운 노드 추가
        Node<T> *temp = new Node<T>(d, NULL);
        if (start == NULL) {
            start = temp;
            last = temp;
        }
        else {
            last->next = temp;
            last = temp;
        }
    }
};
```

첫 번째 원소와 마지막 원소

4. 이터레이터의 이해 : 원리 이해

- list 클래스 및 list 클래스의 이터레이터 클래스 구현 (계속)

```
int main(void)
{
    int i;
    MyList<int> intL;
    MyList<int>::iterator Iter; // 원소 중 하나를 가리킬 이터레이터 생성

    for (i = 0; i < 5; i++)
        intL.push(i);

    Iter = intL.begin();        // 첫 번째 원소를 가리킴
    for (i = 0; i < 5; i++) {
        cout << *Iter << endl;
        Iter++;
    }

    return 0;
}
```

5. 이터레이터의 사용

⌗ vector와 list 클래스의 (이터레이터 관련) 멤버 함수들

멤버 함수	설명
begin	첫 번째 원소의 이터레이터 반환
end	마지막 원소의 바로 다음을 의미하는 이터레이터 반환. 이로부터 마지막 원소를 지나쳤음을 감지할 수 있음
rbegin	마지막 원소의 이터레이터 반환
rend	첫 번째 원소의 바로 앞을 의미하는 이터레이터 반환
insert	이터레이터가 가리키는 특정 위치에 원소 삽입. 이 때 특정 위치란 이터레이터가 가리키는 원소와 바로 이전 원소의 사이를 의미함
erase(iterator pos)	pos가 가리키는 원소 제거
erase(iterator first, iterator last)	first 이터레이터가 가리키는 원소부터 last 이터레이터가 가리키는 원소의 바로 이전 원소까지의 모든 원소 제거
비교 연산자	두 개의 컨테이너 클래스 객체가 서로 같은지(==), 다른지(!=), 또는 대소를 비교할 수 있는 비교 연산자가 준비되어 있음
대입 연산자	obj1 = obj2와 같이 하나의 컨테이너 클래스 객체를 다른 컨테이너 클래스 객체로 대입. obj1은 obj2와 원소의 개수 및 각 원소의 값이 같아짐

함수에서 이터레이터 관련 범위 지정
func(first, last)인 경우
[first 이상, **last** 미만**)**을 의미

5. 이터레이터의 사용

⌗ vector 클래스에 대한 이터레이터 사용 예

```
C:₩WINDOWS₩system...
>> intV1 : 0 1 2 3 4
>> intV2 :
>> intV1 : 0 1 100 2 3 4 101
>> intV1 : 0 1 2 3 4 101
>> intV2 : 0 1 2 3 4 101
intV1과 intV2는 동일하다
계속하려면 아무 키나 누르십시오
```

```
void PrintVector(vector<int> intV, char *name)
{
    vector<int>::iterator iter;

    cout << ">> " << name << " : ";
    for (iter = intV.begin(); iter != intV.end(); iter++) // 각 원소에 접근
        cout << *iter << " ";
    cout << endl;
}
```

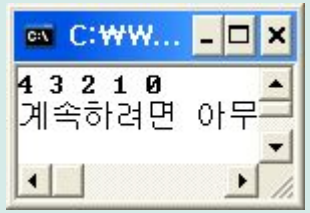
```
int main(void)
{
    int i;
    vector<int> intV1(5);
    vector<int> intV2;
    vector<int>::iterator iter
        = intV1.begin();

    for (i = 0; i < 5; i++) {
        *iter = i;
        iter++;
    }

    PrintVector(intV1, "intV1");
    PrintVector(intV2, "intV2");
```

```
    intV1.insert(intV1.begin() + 2, 100);
    intV1.insert(intV1.end(), 101);
    PrintVector(intV1, "intV1");

    intV1.erase(intV1.begin() + 2);
    PrintVector(intV1, "intV1");

    intV2 = intV1;
    PrintVector(intV2, "intV2");
    if (intV1 == intV2) {
        cout << "intV1과 intV2는 동일하다" << endl;
    }

    return 0;
}
```

5. 이터레이터의 사용

- reverse_iterator의 사용
  - rbegin, rend와 함께 사용 : iter++의 경우 이전 원소로 이동
  - 예 : 역순으로 원소의 값 출력

```
int main(void)
{
    int i;
    vector<int> intV(5);
    vector<int>::iterator f_iter = intV.begin();
    vector<int>::reverse_iterator r_iter;

    for (i = 0; i < 5; i++) {
        *f_iter = i;
        f_iter++;
    }

    for (r_iter = intV.rbegin(); r_iter != intV.rend(); r_iter++) // 역순 접근
        cout << *r_iter << " ";
    cout << endl;

    return 0;
}
```



5. 이터레이터의 사용

- list 클래스에 대한 이터레이터 사용 예 (p.15 vector와 동일한지 확인)

```
void PrintVector(list<int> intV, char *name)
{
    list<int>::iterator iter;

    cout << ">> " << name << " : ";
    for (iter = intV.begin(); iter != intV.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

int main(void)
{
    int i;
    list<int> intV1(5);
    list<int> intV2;
    list<int>::iterator iter = intV1.begin();

    for (i = 0; i < 5; i++) {
        *iter = i;
        iter++;
    }
```

5. 이터레이터의 사용

- list 클래스에 대한 이터레이터 사용 예 (계속)

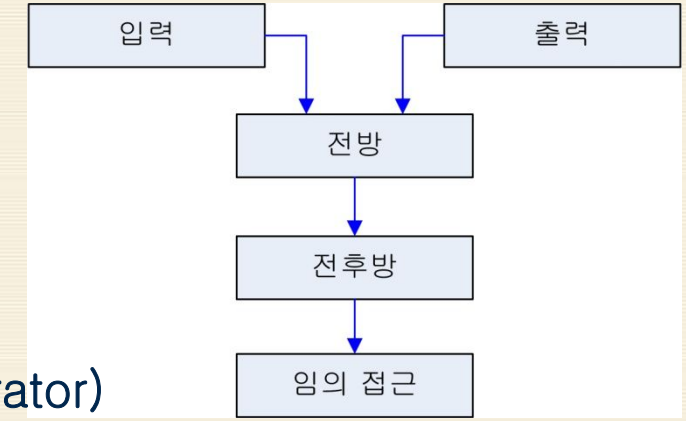
```
    PrintVector(intV1, "intV1");
    PrintVector(intV2, "intV2");

    iter = intV1.begin();
    iter++; iter++;
    intV1.insert(iter, 100);          // 두 번째와 세 번째 원소 사이
    intV1.insert(intV1.end(), 101);   // 마지막 원소 다음
    PrintVector(intV1, "intV1");

    iter = intV1.begin();
    iter++; iter++;
    intV1.erase(iter);                // 세 번째 원소
    PrintVector(intV1, "intV1");

    intV2 = intV1;                    // 대입
    PrintVector(intV2, "intV2");
    if (intV1 == intV2) {
        cout << "intV1과 intV2는 동일하다" << endl;
    }

    return 0;
}
```

vector 이터레이터와의 차이점
list 이터레이터는 vector와 달리
(intV.begin() + 2)와 같은 + 연산 제공하지 않음
list의 특성 : 순차 접근 (비교-vector : 임의접근)

이터레이터에도 종류가 있음!!!

6. 이터레이터의 종류

- 허용된 기능에 따른 이터레이터의 종류
  - 입력 이터레이터(InputIterator)
  - 출력 이터레이터(OutputIterator)
  - 전방 이터레이터(ForwardIterator)
  - 전후방 이터레이터(BidirectionalIterator)
  - 임의 접근 이터레이터(RandomAccessIterator)



- 이터레이터별 기능 및 사용 가능 연산자

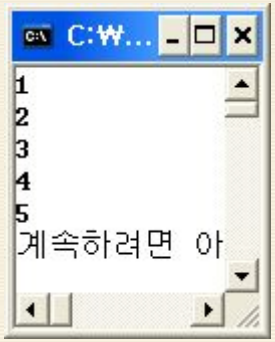
이터레이터 / 기능	입력	출력	전방	전후방	임의 접근
읽기	= *p		= *p	= *p	= *p
쓰기		*p =	*p =	*p =	*p =
1씩 증가, 감소	++	++	++	++, --	++, --
임의 접근					+, -, +=, -+, []

  - vector : 임의 접근 이터레이터
  - list : 전후방 이터레이터

6. 이터레이터의 종류

- 출력 이터레이터인 ostream_iterator 클래스의 사용 예
  - <iterator> 헤더 파일 포함

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

int main(void)
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);
    values.push_back(5);

    ostream_iterator<int> output(cout, "₩n");

    for (int i = 0; i < 5; i++)
        *output = values[i];

    return 0;
}
```



데이터 출력 후 "\n" 자동 출력

일반 포인터를 통해 값을 대입하듯이 화면 출력 가능

알고리즘 별로 요구하는 이터레이터의 종류가 다름!!!

7. 알고리즘의 이해

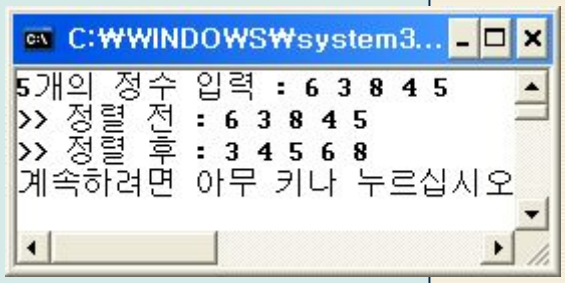
- sort 알고리즘(함수)의 적용
  - 프로토타입
    - void sort(RandomAccessIterator first, RandomAccessIterator last);
    - void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
  - first, last : 적용 범위 [first 이상, last 미만)
  - comp : 정렬 기준 함수
  - 임의 접근 이터레이터에만 적용 가능
  - <algorithm> 헤더 파일 포함

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

void PrintVector(vector<int> intV, char *name)
{
    vector<int>::iterator iter;

    cout << ">> " << name << " : ";
    for (iter = intV.begin(); iter != intV.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

7. 알고리즘의 이해

⌗ sort 알고리즘(함수)의 적용 (계속)

```
int main(void)
{
    int i;
    vector<int> intV(5);

    cout << "5개의 정수 입력 : ";
    for (i = 0; i < 5; i++)
        cin >> intV[i];
    PrintVector(intV, "정렬 전");

    sort(intV.begin(), intV.end());  // 모든 원소들에 대해 정렬 수행
    PrintVector(intV, "정렬 후");

    return 0;
}
```



⌗ 내림차순 정렬

```
bool IntCompare(int a, int b)
{
    return (a > b) ? true : false;
}

sort(intV.begin(), intV.end(), IntCompare);
```

7. 알고리즘의 이해

- vector 객체의 원소가 클래스 객체인 경우의 sort 함수 적용
  - 대소 기준에 대한 일반 함수 사용 또는 < 연산자 오버로딩 사용

```
class CPoint {
private :
    int x, y;

public :
    CPoint(int a = 0, int b = 0) : x(a), y(b) { }
    friend ostream &operator<<(ostream &out, CPoint &Po);
    friend bool Compare(CPoint &Po1, CPoint &Po2);
};

ostream &operator<<(ostream &out, CPoint &Po)
{
    out << "(" << Po.x << ", " << Po.y << ")";
    return out;
}

bool Compare(CPoint &Po1, CPoint &Po2)   // x, y의 합이 작은 객체가 앞에 위치
{
    if (Po1.x + Po1.y < Po2.x + Po2.y)
        return true;
    else
        return false;
}
```

7. 알고리즘의 이해

- vector 객체의 원소가 클래스 객체인 경우의 sort 함수 적용 (계속)

```
void PrintVector(vector<CPoint> intV, char *name)
{
    vector<CPoint>::iterator iter;

    cout << ">> " << name << " : ";
    for (iter = intV.begin(); iter != intV.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}


int main(void)
{
    vector<CPoint> intV(5);

    intV[0] = CPoint(5, 3);
    intV[1] = CPoint(2, 9);
    intV[2] = CPoint(1, 1);
    intV[3] = CPoint(2, 5);
    intV[4] = CPoint(3, 7);
    PrintVector(intV, "정렬 전");

    sort(intV.begin(), intV.end(), Compare);  // Compare 기준 모든 원소 정렬
    PrintVector(intV, "정렬 후");

    return 0;
}
```

7. 알고리즘의 이해

- vector 객체의 원소가 클래스 객체인 경우의 sort 함수 적용 (계속)
  - < 연산자 오버로딩을 CPoint 클래스에 포함시키는 경우 별도로 Compare 함수를 작성하지 않아도 됨

```
bool operator<(CPoint &Po) {
    if (x + y < Po.x + Po.y)
        return true;
    else
        return false;
}
```

8. 알고리즘의 종류

알고리즘의 종류 및 대표적인 알고리즘

종류	알고리즘	이터레이터	설명
변경 불가 시퀀스 연산	for_each	입력	범위 내의 원소에 대해 지정한 함수 수행
	find	입력	특정 값을 가진 첫 번째 원소의 이터레이터 반환
	count	입력	특정값을 가진 원소의 개수 반환
변경 가능 시퀀스 연산	rotate	전방	원소들을 왼쪽으로 이동 (circular)
	random_shuffle	전방	범위 내의 원소들을 임의의 순서로 재정렬
	reverse	전후방	역순으로 재정렬
정렬 및 관련 연산	sort	임의 접근	정렬

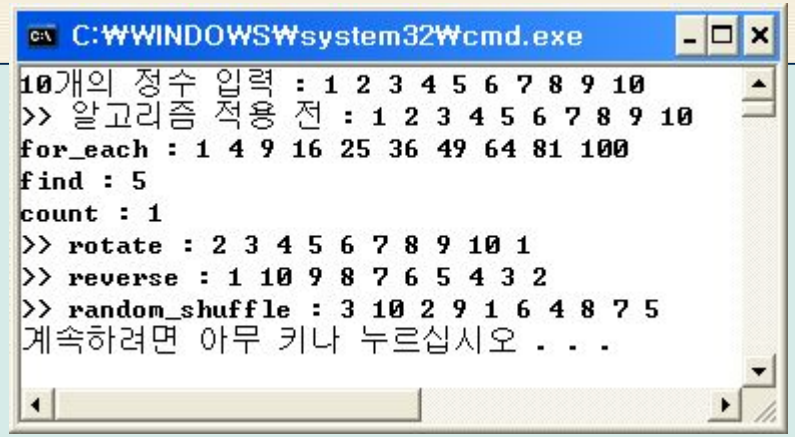
알고리즘 사용 예

```
void PrintVector(vector<int> intV, char *name)
{
    vector<int>::iterator iter;

    cout << ">> " << name << " : ";
    for (iter = intV.begin(); iter != intV.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

8. 알고리즘의 종류

⌗ 알고리즘 사용 예 (계속)

```
void Print(int val)
{
    cout << val * val << " ";
}

int main(void)
{
    int i;
    vector<int> intV(10);
    vector<int>::iterator iter;

    cout << "10개의 정수 입력 : ";
    for (i = 0; i < 10; i++)
        cin >> intV[i];
    PrintVector(intV, "알고리즘 적용 전");

    cout << "for_each : ";
    for_each(intV.begin(), intV.end(), Print); // 모든 원소에 대해 Print 적용
    cout << endl;
```



8. 알고리즘의 종류

⌗ 알고리즘 사용 예 (계속)

```
C:₩WINDOWS₩system32₩cmd.exe
10개의 정수 입력 : 1 2 3 4 5 6 7 8 9 10
>> 알고리즘 적용 전 : 1 2 3 4 5 6 7 8 9 10
for_each : 1 4 9 16 25 36 49 64 81 100
find : 5
count : 1
>> rotate : 2 3 4 5 6 7 8 9 10 1
>> reverse : 1 10 9 8 7 6 5 4 3 2
>> random_shuffle : 3 10 2 9 1 6 4 8 7 5
계속하려면 아무 키나 누르십시오 . . .
```

```
    cout << "find : ";
    iter = find(intV.begin(), intV.end(), 5); // 값이 5인 원소의 위치 반환
    cout << *iter << endl;

    cout << "count : ";
    cout << count(intV.begin(), intV.end(), 5) << endl;// 값이 5인 원소 개수

    rotate(intV.begin(), intV.begin() + 1, intV.end());// 왼쪽으로 1씩 이동
    PrintVector(intV, "rotate");

    reverse(intV.begin(), intV.end());        // 역순으로 재정렬
    PrintVector(intV, "reverse");

    random_shuffle(intV.begin(), intV.end()); // 임의 순서로 재정렬
    PrintVector(intV, "random_shuffle");

    return 0;
}
```

9. 컨테이너 클래스의 종류

- 컨테이너 클래스의 종류
  - 시퀀스 컨테이너 : 원소들 사이의 순서 개념 포함
  - 컨테이너 어댑터 : 순서 개념, 시퀀스 컨테이너를 기반으로 만듦
    - stack, queue ← deque, priority_queue ← vector
  - 결합 컨테이너 : (값 + 키)를 원소로 저장

종류	컨테이너 클래스	설명	기능
시퀀스 컨테이너	vector	배열	후미 신속 삽입, 삭제
	deque	double_ended 큐	선두 또는 후미에 신속 삽입, 삭제
	list	doubly-linked 리스트	임의 위치에 신속 삽입, 삭제
컨테이너 어댑터	stack	LIFO 구조의 스택	스택
	queue	FIFO 구조의 큐	큐
	priority_queue	우선 순위를 가진 큐	우선 순위 큐
결합 컨테이너	set	집합	신속 검색, 이중 요소 불허
	multiset	이중 요소 허용 집합	신속 검색, 이중 요소 허용
	map	키-값 연결	신속 검색, 이중 요소 불허
	multimap	키-값들 연결	신속 검색, 이중 요소 허용

9. 컨테이너 클래스의 종류

- multimap 클래스의 사용 예
  - 방에 있는 물건 저장 : (방번호, 물건)의 쌍을 저장
  - 키(방번호) 하나에 값(물건)들이 존재 → multimap으로 표현

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main(void)
{
    multimap<int, string> Room;

    Room.insert(pair<int, string>(101, "chair"));      // 원소 추가
    Room.insert(pair<int, string>(102, "computer"));
    Room.insert(pair<int, string>(101, "desk"));
    Room.insert(pair<int, string>(101, "book"));
    Room.insert(pair<int, string>(102, "notebook"));
```

방번호,물건

9. 컨테이너 클래스의 종류

multimap 클래스의 사용 예 (계속)

```
    cout << "모든 방과 물건: ";
    multimap<int, string>::iterator iter;
    for (iter = Room.begin(); iter != Room.end(); iter++)
        cout << (*iter).first << "-" << (*iter).second << " ";
    cout << endl;

    cout << "101호에 있는 물건 개수: " << Room.count(101) << endl;
    pair<multimap<int, string>::iterator,
        multimap<int, string>::iterator> range = Room.equal_range(101);
    for (iter = range.first; iter != range.second; iter++)
        cout << (*iter).second << " ";
    cout << endl;

    return 0;
}
```

특정 키에 해당하는 값들의 범위

```
C:₩WINDOWS₩system32₩cmd.exe
모든 방과 물건 : 101-chair 101-desk 101-book 102-computer 102-notebook
101호에 있는 물건 개수 : 3
chair desk book
계속하려면 아무 키나 누르십시오 . . .
```