17강. 표준 템플릿 라이브러리

C++ 프로그래밍

jhhwang@kumoh.ac.kr

컴퓨터공학과 황준하

목차

- 표준 템플릿 라이브러리 구성
- vector 클래스 기본 사용 방법
- list 클래스 기본 사용 방법
- 이터레이터
  - 이터레이터의 이해
  - 이터레이터를 활용한 멤버 함수 사용
  - 이터레이터의 종류
- 알고리즘
- 컨테이너 클래스의 종류

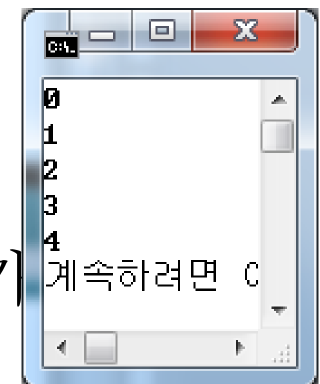
표준 템플릿 라이브러리 구성

- 컨테이너 클래스
  - stack, queue, vector, list 등 자료구조를 클래스화
  - 템플릿을 기반으로 하고 있어 모든 타입의 원소 수용 가능
- 이터레이터
  - 모든 자료구조의 특정 원소에 대한 동일한 접근 방법 제공
  - 포인터와 유사!
- 알고리즘
  - 모든 자료구조에 대해 적용 가능한 전역 함수들

vector 클래스 기본 사용 방법

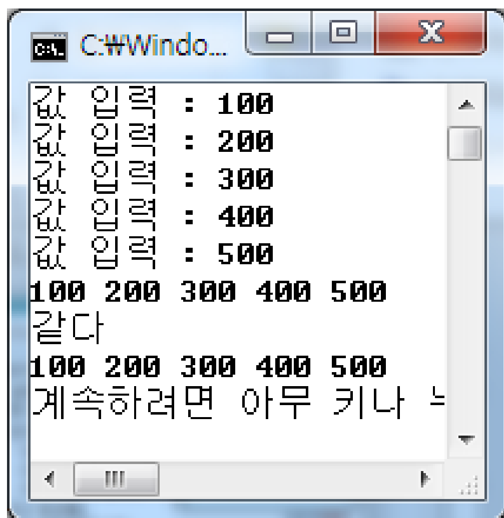
- vector
  - 사용하기 편한 1차원 배열
  - 헤더 파일 : <vector>
  - 생성자
    - vector<int> intV; // int 배열 객
    - vector<int> intV(5); // 5개 원소를
    - vector<CPoint> poV(3); // CPoint 객
  - 기본 사용 방법
    - 기존 1차원 배열과 동일
    - intV[2] = 5;
  - 멤버 함수와 알고리즘을 통해 다양한 기능 수행 가

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

void main(void)
{
	vector<int> intV(5);
	for (int i = 0; i < 5; i++)
		intV[i] = i;
	for (int i = 0; i < 5; i++)
		cout << intV[i] << endl;
}
```



vector 클래스 기본 사용 방법

- 멤버 함수 사용 예




```
void main(void)
{
    int i, value;
    vector<int> intV1, intV2;

    for (i = 0; i < 5; i++) {
        cout << "값 입력 : ";
        cin >> value;
        intV1.push_back(value);
    }

    for (i = 0; i < 5; i++)
        cout << intV1[i] << " ";
    cout << endl;

    intV2 = intV1;
    if (intV1 == intV2)
        cout << "같다" << endl;

    for (i = 0; i < intV2.size(); i++)
        cout << intV2[i] << " ";
    cout << endl;
}
```

마지막에 원소 추가 → intV1.push_back(value);

배열과 같이 사용 → cout << intV1[i] << " ";

대입 가능 → intV2 = intV1;

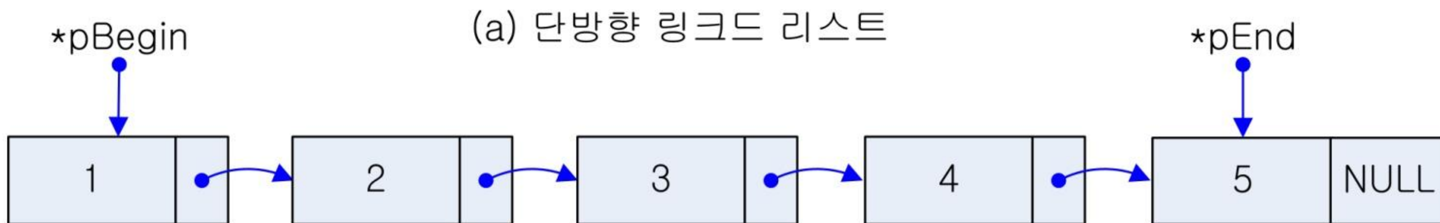
비교 가능 → if (intV1 == intV2)

현재 원소 개수 → intV2.size()

이터레이터 개념과 함께 더 많은 멤버 함수 사용 가능!

list 클래스 기본 사용 방법

- 단방향 링크드 리스트와 양방향 링크드 리스트



- list 클래스
  - 양방향 링크드 리스트를 구현해 놓은 클래스 템플릿
  - 내부 구조는 리스트이지만 vector와 유사하게 편리한 사용 가능
  - 헤더 파일 : <list>

list 클래스 기본 사용 방법

- list 클래스 사용 예

```
void main(void)
{
	int i, value;
	list<int> intL1, intL2;

	for (i = 0; i < 5; i++) {
		cout << "값 입력 : ";
		cin >> value;
		intL1.push_back(value);
	}

	cout << intL1.front() << " ";
	cout << intL1.back() << endl;

	intL2 = intL1;
	if (intL1 == intL2)
		cout << "같다" << endl;

	cout << intL1.front() << " ";
	cout << intL1.back() << endl;
}
```

```
값 입력 : 100
값 입력 : 200
값 입력 : 300
값 입력 : 400
값 입력 : 500
100 500
같다
100 500
계속하려면 아무 ㅋ
```

마지막에 원소 추가 → intL1.push_back(value);

첫 번째, 마자막 원소 출력 → cout << intL1.front() << " ";

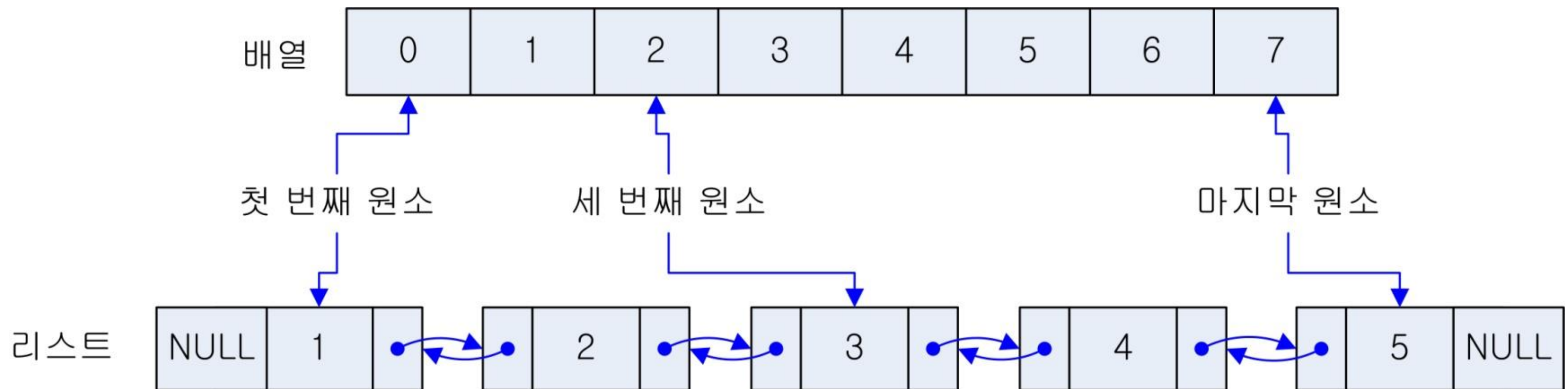
대입 가능 → intL2 = intL1;

비교 가능 → if (intL1 == intL2)

list는 intL1[3]과 같은 [] 연산자 없음
배열과 리스트는 용도가 서로 다름
배열 : 임의 접근, 리스트 : 빠른 삽입 삭제

이터레이터의 이해

- 배열과 리스트의 내부 구조



- 특정 원소를 가리키는 방법
  - 배열과 리스트에 동일한 방법 필요 → 멤버 함수뿐 아니라 여러 가지 알고리즘(전역 함수)에서 활용
- 이터레이터(iterator)
  - 포인터와 유사한 개념 → 포인터와 유사하게 사용 가능

이터레이터의 이해

- vector의 경우 실제 포인터를 통한 원소 접근 가능

```
void main(void)
{
    int i;
    vector<int> intV(5);
    int *pV = &intV[0];

    for (i = 0; i < 5; i++) {
        cout << "값 입력 : ";
        cin >> (*pV);
        pV++;
    }

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

C:\Windows\system32\cmd.exe

```
값 입력: 100
값 입력: 200
값 입력: 300
값 입력: 400
값 입력: 500
100     200     300     400     500
```

- 포인터로 첫 번째 원소 가리킴 (`int *pV = &intV[0];`)
- 역참조 연산자를 통해 원소의 값 저장 (`cin >> (*pV);`)
- ++ 연산자로 다음 원소 가리킴 (`pV++;`)

list 객체에 대해서도 포인터로 각 원소를 가리킬 수 있을까? ++ 연산자 사용 등

→ No! → 이터레이터 도입

이터레이터의 내부 구조 이해

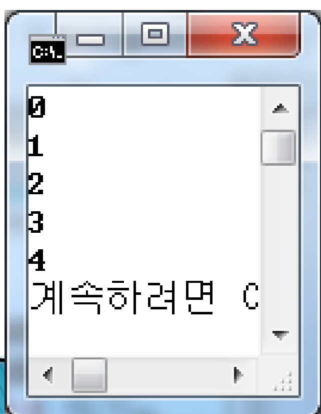
- vector 원소를 가리키는 vector 이터레이터 내부 구조
  - 다음과 같이 포인터 역할을 하는 이터레이터 클래스를 직접 만든다면?

```
void main(void)
{
    int i;
    MyVector<int> intV;
    VectorIterator<int> vIter = intV.begin();

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

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

기본적으로 5개의 원소 생성 가정

intV.begin() : 첫 번째 원소의 주소 반환
vIter : 포인터 역할의 이터레이터 객체

포인터와 동일하게 사용 가능



이터레이터의 내부 구조 이해

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

public :
	VectorIterator(T *p = 0) : ptr(p) { }
	T &operator*() { return (*ptr); }
	void operator++(int) { ptr++; }
	VectorIterator &operator=(VectorIterator<T> Iter)
		{ ptr = Iter.ptr; return (*this); }
};
```

- 생성자 : 해당 주소 가리킴
- 역참조 연산자
- ++ 증가 연산자
- 대입 연산자

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

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

- Iterator라는 이름이 타입 사용 가능
  MyVector<int>::iterator ...
- 첫 번째 원소의 주소로 Iterator를 만들어 반환

이터레이터의 내부 구조 이해

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

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

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

- 첫 번째 주소를 가리킴 → `MyVector<int>::iterator vIter = intV.begin();`
- 이터레이터가 가리키는 원소 → `*vIter = i;`
- 다음 원소를 가리킴 → `vIter++;`
- 다시 첫 번째 원소를 가리킴 → `vIter = intV.begin();`

list 또한 동일한 원리로 ListIterator를 만들어 VectorIterator와 동일한 방식으로 사용 가능

이터레이터를 활용한 멤버 함수 사용

vector와 list 클래스의 멤버 함수

멤버 함수	기능
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 사이 의미
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 연산자
대입 연산자	obj1 = obj2와 같이 하나의 컨테이너 클래스 객체를 다른 컨테이너 클래스 객체로 대입

범위 : first 이상, last 미만을 의미

이터레이터를 활용한 멤버 함수 사용

- vector 멤버 함수 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;
}
```

이터레이터를 이용해 모든 원소의 값을 출력

이터레이터를 활용한 멤버 함수 사용

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

	for (i = 0; i < 5; i++) {
		*iter = i;
		iter++;
	}
	PrintVector(intV, "intV");

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

	intV.erase(intV.begin() + 1, intV.begin() + 4);
	PrintVector(intV, "intV");
}
```

```
C:\Windows\system32...
>> intV : 0 1 2 3 4
>> intV : 0 1 100 2 3 4 101
>> intV : 0 3 4 101
계속하려면 아무 키나 누르십시오
```

- 세 번째 원소 앞에 100 추가
- 마지막 앞(즉, 마지막)에 101 추가
- 두 번째 원소부터 다섯 번째 원소 앞(네 번째 원소)까지 삭제

이터레이터를 활용한 멤버 함수 사용

- list 멤버 함수 사용 예

```
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;
}
```

vector에서의 내용과 100% 동일

이터레이터를 활용한 멤버 함수 사용

```
void main(void)
{
	list<int> intV(5);
	list<int>::iterator iter = intV.begin(), iter2;

	for (int i = 0; i < 5; i++) {
		*iter = i;
		iter++;
	}
	PrintVector(intV, "intV");

	iter = intV.begin();
	iter++; iter++;
	intV.insert(iter, 100);
	intV.insert(intV.end(), 101);
	PrintVector(intV, "intV");

	iter = intV.begin();
	iter++;
	iter2 = iter;
	iter2++; iter2++; iter2++;
	intV.erase(iter, iter2);
	PrintVector(intV, "intV");
}
```

```
C:\Windows\system32...
>> intV : 0 1 2 3 4
>> intV : 0 1 100 2 3 4 101
>> intV : 0 3 4 101
계속하려면 아무 키나 누르십시오
```

list의 이터레이터는 임의 원소 접근을 허용하지 않음 → 순차적으로 접근

두 번째 원소에서 다섯 번째 원소로 이

이터레이터의 종류

이터레이터의 종류 및 기능

이터레이터	읽기 = *iter	쓰기 *iter =	다음 원소 이동 iter++	이전 원소 이동 iter--	임의 원소 이동 iter[4] iter + 3
입력	O		O		
출력		O	O		
전방	O	O	O		
전후방	O	O	O	O	
임의 접근	O	O	O	O	O

- vector 이터레이터 : 임의 접근 이터레이터
- list 이터레이터 : 전후방 이터레이터

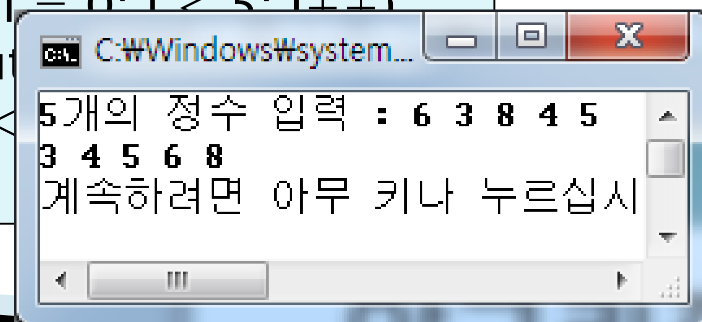
이터레이터의 종류에 따라 적용 가능한 알고리즘(전역 함수)이 달라짐!

알고리즘

- sort 알고리즘(전역 함수) 사용 방법
  - 임의 접근 이터레이터에 적용 가능 → vector 가능
  - 헤더 파일 : ‹algorithm›
  - void sort(임의접근이터레이터 iter1, 임의접근이터레이터 iter2);
    - iter1 원소부터 iter2 원소 앞까지 정렬

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

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

```
    cout << "5개의 정수 입력 : ";
    for (int i = 0; i < 5; i++)
        cin >> intV[i];

    sort(intV.begin(), intV.end());
    for (int i
        cout
    cout <<
}
```

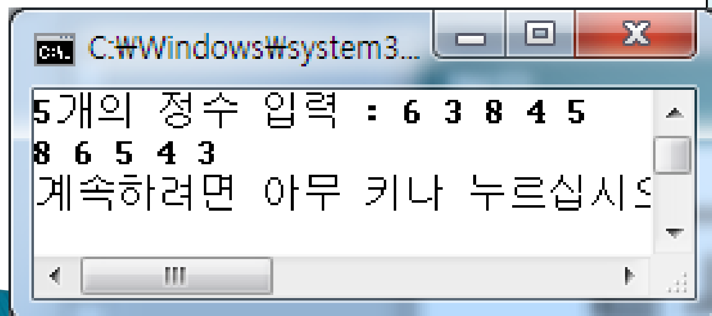


알고리즘

- sort 알고리즘 : 정렬 기준을 함수로 추가 가능
  - 디폴트 오름차순 → 내림차순으로 정렬하려면?
  - void sort(iter1, iter2, 정렬기준함수명);

```
bool Decrease(int a, int b)
{
	if (a > b)
		return true;
	else
		return false;
}
```

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

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

	sort(intV.begin(), intV.end(), Decrease);
	for (int i = 0; i < 5; i++)
		cout << intV[i] << " ";
	cout << endl;
}
```



알고리즘의 종류

- 대표적인 알고리즘 종류

알고리즘	이터레이터	기능
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
rotate	전방	원소들을 왼쪽으로 이동 (circular)
random_shuffle	전방	범위 내의 원소들을 임의의 순서로 재정렬
reverse	전후방	역순으로 재정렬
sort	임의 접근	정렬

컨테이너 클래스의 종류

- 컨테이너 클래스의 종류

클래스	설명	기능
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
stack	LIFO 구조의 스택	스택
queue	FIFO 구조의 큐	큐
priority_queue	우선 순위를 가진 큐	우선 순위 큐
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