C++ Iterator(반복자)

태그	
공개여부	☑
날짜	
작성일자	

https://eehoeskrap.tistory.com/263

C++ 반복자(Iterator)

C++ 라이브러리는 반복자를 제공하는데 이것을 사용하면 라이브러리의 방식대로 자료구조를 액세스 할 수 있다.

따라서 라이브러리가 효과적으로 동작한다는 것을 보장 할 수 있다는 장점이 있다.

즉, 포인터와 상당히 비슷하며, 컨테이너에 저장되어 있는 원소들을 참조할 때 사용한다.

추상적으로
말하자면, 반복자란 컨테이너에 저장되어 있는 모든 원소들을 전체적으로 훑어 나갈 때 사용하는, 일종의 포인터와 비슷한 객체라고 할 수 있다. 알고리즘 마다 각기 다른 방식으로 컨테이너를 훑어가기 때문에, 반복자에도 여러가지 종류가 있게 된다.

반복자의 성질

- 컨테이너와 컨테이너 안의 있는 요소를 구별
- 요소의 값 확인
- 컨테이너 안에 있는 요소들 간에 이동할 수 있는 연산 제공
- 컨테이너가 효과적으로 처리할 수 있는 방식으로 가용한 연산들을 한정

반복자 구간(range)

구간이란
컨테이너에 담긴 값들의 시퀀스를 나타낸다. 구간은 반복자 한 쌍으로 나타내며, 이 반복자들이 각각 시퀀스의 시작과 끝을 가리킨다. 반복자는 특정 값을 지정하는데 사용될 수 있으며, 연속적인 메모리 영역을 두 개의 포인터로 나타내듯이 한 쌍의 반복자는 값들의 구간을 설정하는데 사용될 수 있다.

그러나
반복자의 경우 설정되는 범위 내의 값들이 반드시 물리적으로 연속적이어야 할 필요가 없다 .단, 이 두개의 반복자가 같은 컨테이너로부터 생성된 것이어야 하며, 두번째 반복자는 첫번째 반복자 이후에 와야한다. 그리고 첫번째 반복자에서 두번째 반복자까지 차례로 컨테이너 내부의 원소들을 처리하게 되므로 논리적인 연속성을 지닐 수 있다.

또한 반복자 2개를 사용하여 컨테이너의 특정 구간에 속한 원소들을 나타내고자 한다면, 두번째 반복자가 첫번째 반복자에 도달가능 해야 한다.

그리고, 포인터는 널(NULL) 값을 가질 수 있는데 이는 아무것도 가리키지 않는다는 의미이다. 반복자 또한 어떤 값도 가리키지 않는 반복자를 참조하는 것은 에러를 발생시킨다.

- **end() 함수는 끝이 아니다**

컨테이너를
다룰 때 자주 쓰이는 end()라는 멤버함수는 컨테이너의 맨 마지막 원소를 가리키는게 아니다. end()가 가리키고 있는 것은 맨 마지막 원소의 바로 다음번 원소이다. 따라서 이러한 반복자를 past-the-end 반복자라고 부른다. (종점을 지나쳐버린 곳을 가리키는 반복자)

end() 멤버 함수를 통해 얻어지는 반복자는 결과적으로 아무 의미가 없는 것을 가리키고 있는 것이며, 이 반복자가 가리키는 것을 참조하면 예상치 못한 오류가 발생하게 된다.

또한 아무 원소도 없는 컨테이너의 begin()과 end()는 같다.

반복자의 종류

- **입력 반복자(input iterator)** : 읽기만 가능, 순방향 이동, 현 위치의 원소를 한 번만 읽을 수 있는 반복자
- **출력 반복자(output iterator)** : 쓰기만 가능, 순방향 이동, 현 위치의 원소를 한 번만 쓸 수 있는 반복자
- **순방향 반복자(forward iterator)** : 읽기/쓰기 모두 가능, 순방향 이동(++)이 가능한 재할당될 수 있는 반복자

- **양방향 반복자(bidirectional iterator)** : 읽기/쓰기 모두 가능, 순/역 방향 이동(--)이 가능한 반복자
- **임의 접근 반복자(random access iterator)** : 읽기/쓰기 모두 가능, 임의 접근,

  양방향 반복자 기능에 +, -, += , -=, [] 연산이 가능

  모든 컨테이너는 양방향 반복자 이상을 제공한다.

  배열 기반 컨테이너인 vector와 deque는 임의 접근 반복자를 제공한다.

반복자 종류	사용 방식	읽기	접근	쓰기	증감
입력 반복자(input iterator)	istream_iterator	=*p	->		++
출력 반복자 (output iterator)	ostream_iteratorinserterfront_inserterback_inserter			*p=	++
순방향 반복자 (forward iterator)		=*p	->	*p=	++
양방향 반복자 (bidirectional iterator)	listset 과 multisetmap 과 multimap	=*p	->	*p=	++ --
임의접근 반복자 (random access iterator)	일반 포인터vectordeque	=*p	->[]	*p=	++ --+ -+= -=

추가적인 자료 : http://mayple.tistory.com/

반복자 개념을 포인터로 적용한 예제

정수형 배열을 가리키는 포인터 it를
선언하고 배열 첫 번째 요소의 번지에서 시작하여 끝 다음점 요소 직전까지 순회하면서 *it를 출력하면 배열 요소 전체가 출력된다. 여기서 사용된 it 포인터는 배열의 한 요소를 가리키며 증가하고 비교되며 *연산자로 요소를 읽기도 하므로 반복자의 요구 조건을 모두 만족한다.

```
#include <iostream>  using namespace std;  void main() {       int ari[]=;       int *it;       for (it=&ari[0];it!=&ari[5];it++) {
printf("%d\n",*it);       } }
```

벡터를 이용한 반복자

정수형 벡터에 1부터 5까지 정수 값을 채워넣었다. 벡터의 한 요소를 가리키는 반복자는 다음과 같이 선언한다.

vector<T>::iterator it;

vector<T>가 클래스
이름이고 이 클래스 안에 iterator라는 타입이 typedef로 정의되어 있으므로 이 타입으로 변수를 하나 선언하면 벡터의 한 요소를 가리키는 반복자가 된다. for 루프에서는 반복자를 begin으로 초기화 하고 end 직전까지 반복자를 증가시키며 벡터의 매 요소를 순회하였다.

```
#include <iostream>  #include <vector>  using namespace std;  void main() {       int ari[]=;       vector<int> vi(&ari[0],&ari
[5]);       vector<int>::iterator it;       for (it=vi.begin();it!=vi.end();it++) {            printf("%d\n",*it);       } }
```

리스트를 이용한 반복자

이는 위 예제의 vector를 list로 바꾸었다. 벡터와 순회하는 방법이 완전 동일하다. begin ~ end 사이를 반복자가 순회하여 *it 표현식으로 순회중의 요소를 액세스 할 수 있다.

```
#include <iostream>  #include <list>  using namespace std;  void main() {       int ari[]=;       list<int> li(&ari[0],&ari[5]);
list<int>::iterator it;       for (it=li.begin();it!=li.end();it++) {           printf("%d\n",*it);       } }
```

벡터와 리스트의 차이

반복자가 가리키는 요소를 삭제 할 경우 그 반복자는 무효화 된다.

vector에 대해 erase를 호출하면 방금 삭제된 요소 다음에 있는 요소들을 가리키는 모든 반복자는 무효화 된다.

또한 push_back을 사용하여 vector에 요소를 추가해도 해당 vector를 가리키고 있던 모든 반복자는 무효화 된다.

vector에 한 요소를 삭제하면 그 다음 요소들이 이동되고, 한 요소를 추가하면 새로운 요소를 위한 공간을 확보하기 위해 전체 vector가 재할당되기 때문이다.

하지만 list에서는 erase나 push_back이 다른 요소들에 대한 반복자를 무효화 시키지 않고 실제로 삭제된 요소를 가리키는 반복자만 무효화 된다. 왜냐하면 그 요소는 더 이상 존재하지 않기 때문이다.

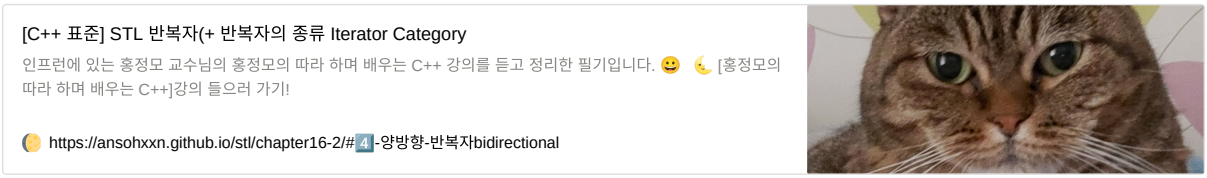
참고자료 1 : http://egloos.zum.com/printf/v/1824410

참고자료 2 : http://hyeonstorage.tistory.com/318

참고자료 3 : http://mayple.tistory.com

참고자료 4 : http://egloos.zum.com/lsrsp/v/13053

반복자



- 컨테이너에 저장되있는 원소들을 공통적인 방법으로
  하나씩 접근할 수 있게 해줌.
  - 모든 컨테이너들이 다 같은 방법으로 반복자 사용 가능.
- 각 타입에 `::iterator` 또는 `::const_iterator` 를 뒤에 붙여주면 사용이 가능하다.
  - vector 컨테이너의 반복자 itr
    - vector<int>**::iterator itr**;
  - vector 컨테이너의 const한 반복자 citr
    - vector<int>**::const_iterator citr**;
- **포인터**와 비슷하게 사용한다.
  - 간접 참조 가능
    - `itr = v.begin() + 2` 에서 `itr` 간접 참조를 하면 세번째 원소 값이 리턴된다.
- iterator 와 const_iterator 의 차이
  - const_iterator 는 반복자가 가리키는 원소의 값을 변경하지 못한다.
    - 반복자 값이 변경되지 못하는게 아니고 반복자가 가리키는 원소의 값이 변경될 수 없는 것!
  - 일반 반복자는 포인터와 비슷하고 const 반복자는 const 포인터와 비슷하다. (간접참조로 값을 변경하지 못하는)
- 초기화 하는 방법
  - itr = container.begin()
    - 컨테이너이름.begin() 하면 첫번째 원소의 iterator를 리턴

반복자의 종류Permalink

```
unordered_map<string, set<int>> db;
```

`set` 컨테이너는 자동으로 정렬이 된다는 것에 착안하여 처음에는 이렇게 Value를 `vector`가 아닌 `set`으로 했었다.

```
set<int>::iterator itr = lower_bound(score.begin(), score.end(), stoi(condition[4]));
answer.push_back(score.end() - itr); // ❌❌ 에러 발생!
```

근데 `set`의 반복자끼리를 사칙연산 하려고 하니 자꾸 불가능하다고 에러가 뜨는 것이였다. `vector` 반복자는 잘만 되던데 뭐지 싶었다. (근데 set이 만약에 반복자 연산에 문제 없었다 하더라도 `info` for문 안에서 매번 자동 정렬이 된단 소리니까 set을 사용하면 시간초과가

될 수도 있었을 것 같다.)

> 반복자도 종류가 있다. 산술 연산이 가능한건 “임의 접근 반복자”를 가지는 vector, deque 밖에 없다.

STL 의 모든 컨테이너 혹은 함수들은 아래와 같은 종류의 반복자들을 지원하는데, 아래 반복자들 중 어느 것들까지 지원을 하느냐에 따라 사용할 수 있는 연산, 함수들이 달라진다.

컨테이너마다 지원하는 반복자의 종류가 다르다.

반복자의 종류Permalink

<table>
<tr><th colspan="5">Iterator category</th><th>Defined operations</th></tr>
<tr><td rowspan="5">LegacyContiguousIterator</td><td rowspan="4">LegacyRandomAccessIterator</td><td rowspan="3">LegacyBidirectionalIterator</td><td rowspan="2">LegacyForwardIterator</td><td>LegacyInputIterator</td><td>• read
• increment (without multiple passes)</td></tr>
<tr><td></td><td>• increment (with multiple passes)</td></tr>
<tr><td colspan="2"></td><td>• decrement</td></tr>
<tr><td colspan="3"></td><td>• random access</td></tr>
<tr><td colspan="4"></td><td>• contiguous storage</td></tr>
<tr><td colspan="6">Iterators that fall into one of the above categories and also meet the requirements of LegacyOutputIterator are called mutable iterators.</td></tr>
<tr><td colspan="5">LegacyOutputIterator</td><td>• write
• increment (without multiple passes)</td></tr>
</table>

<table>
<tr><th colspan="5">category</th><th>properties</th><th>valid expressions</th></tr>
<tr><td colspan="5">all categories</td><td>copy-constructible, copy-assignable and destructible</td><td>X b(a);
b = a;</td></tr>
<tr><td colspan="5"></td><td>Can be incremented</td><td>++a
a++</td></tr>
<tr><td rowspan="7">Random Access</td><td rowspan="5">Bidirectional</td><td rowspan="4">Forward</td><td rowspan="2">Input</td><td>Supports equality/inequality comparisons</td><td>a == b
a != b</td></tr>
<tr><td>Can be dereferenced as an rvalue</td><td>*a
a->m</td></tr>
<tr><td rowspan="3">Output</td><td>Can be dereferenced as an lvalue
(only for mutable iterator types)</td><td>*a = t
*a++ = t</td></tr>
<tr><td>default-constructible</td><td>X a;
X()</td></tr>
<tr><td>Multi-pass: neither dereferencing nor incrementing affects dereferenceability</td><td>{ b=a; *a++;
*b; }</td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td>--a
a--
*a--</td></tr>
<tr><td colspan="3"></td><td>Supports arithmetic operators + and -</td><td>a + n
n + a
a - n
a - b</td></tr>
<tr><td colspan="3"></td><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td>a < b
a > b
a <= b
a >= b</td></tr>
<tr><td colspan="3"></td><td>Supports compound assignment operations += and -=</td><td>a += n
a -= n</td></tr>
<tr><td colspan="3"></td><td>Supports offset dereference operator ([])</td><td>a[n]</td></tr>
</table>

- 출처
  - cppreference
  - cplusplus

1️⃣ 입력 반복자(Input)Permalink

- Read 및 접근 만 가능.
- Write 불가능
- 증감은 `++` 연산만 가능.
- 비교는 `==`, `!=` 연산만 가능.

```
cout << itr->size() << endl; // 접근 ⭕
뫄뫄 = *itr;  // 읽기 가능 ⭕ (간접 참조처럼 * 연산자)
itr++; // ++ 연산 가능 ⭕
if (itr1 == itr2); // ⭕
*itr = 뫄뫄;  // 쓰기 불가능 ❌ (간접 참조 수정 불가)
```

2️⃣ 출력 반복자(Output)Permalink

- Write 만 가능.
- Read 및 접근 불가능
- `++` 연산만 가능. (`itr++`)
- 비교 연산자 불가능

```
cout << itr->size() << endl; // 접근 불가능 ❌
뫄뫄 = *itr;  // 읽기 불가능 ❌
itr++; // ++ 연산 가능 ⭕
if (itr1 == itr2); // 비교 연산 불가능 ❌
*itr = 뫄뫄;  // 쓰기 가능 ⭕
```

3️⃣ 순방향 반복자(Forward)Permalink

- 읽기 쓰기 접근 다된다.
- 산술 연산 👉 `++`만 가능
- 비교 연산 👉 `==`, `!=`만 가능

4️⃣ 양방향 반복자(Bidirectional)Permalink

- 읽기 쓰기 접근 다된다.
- 산술 연산 👉 `++`, `-` 가능
- 비교 연산 👉 `==`, `!=`만 가능

list, set, map 은 이 반복자를 지원한다. **(그래서 set 의 반복자끼리 뺄셈을 하려 했을 때 불가능했던 것이다.)**

이 양방향 반복자를 지원하지 않는 컨테이너는 알고리즘 헤더의 *reverse()* 함수를 사용할 수 없다. *reverse* 함수는 이 양방향 반복자를 사용하기 때문이다. **이처럼 함수, 연산에 따라 사용할 수 있는 반복자가 정해져있다는 것을 꼭 알아두자!**

3️⃣ 임의접근 반복자(Random Access))Permalink

- 읽기 쓰기 접근 다된다.
- 산술 연산 👉 `++`, `-`, ✨`+`, `,` `+=`, `=` 가능✨
- 비교 연산 👉 `==`, `!=`, ✨`>`, `<`, `>=`, `<=` 가능✨
- 첨자 연산자 사용 가능 👉 `[]`
  - `itr[n]`은 곧 `(itr + n)`과도 같다.

    ```
    vector<int>::iterator itr;
    itr[4];  // itr 에서 4 칸 더 간 곳의 반복자!
    ```

vector, deque 는 이 반복자를 지원한다. 그래서 반복자끼리 사칙연산을 해야한다면 `vector`를 사용해야 한다.

vector에서의 사용 예시Permalink

예시 1Permalink

- `itr = container.begin();`

    - 첫번째 주소를 받아
- `while (itr != container.end())`
    - 반복자가 끝까지 다 돌때까지.
- `itr`
    - 반복자가 가리키는 원소를 이용한 후 (간접참조같이)
- `++itr` 다음 반복자.

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <map>

using namespace std;

int main()
{
  vector<int> container;

  for (int i = 0; i < 10; ++i)
    container.push_back(i);

  vector<int>::const_iterator itr;
  itr = container.begin();

  while (itr != container.end())
  {
    cout << *itr << "  ";
    ++itr;
  }
}
```

예시 2Permalink

- for문 사용

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <map>

using namespace std;

int main()
{
  vector<int> container;

  for (int i = 0; i < 10; ++i)
    container.push_back(i);

  vector<int>::const_iterator itr;

  for (auto itr = container.begin(); itr != container.end(); ++itr)
    cout << *itr << "  ";
  cout << endl;
}
```

예시 3Permalink

- for-each 문 이용
- `for (auto& itr : container)`

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <map>

using namespace std;

int main()
{
  vector<int> container;

  for (int i = 0; i < 10; ++i)
    container.push_back(i);
```

```
  vector<int>::const_iterator itr;

  for (auto& itr : container)
    cout << itr << "  ";
  cout << endl;
}
```

list에서의 사용 예시Permalink

- `list<int> container`;
  - `push_back` 사용
- 벡터와 동일한 반복자 문법.

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <map>

using namespace std;

int main()
{
  list<int> container;

  for (int i = 0; i < 10; ++i)
    container.push_back(i);

  vector<int>::const_iterator itr;

  for (auto& itr : container)
    cout << itr << "  ";
  cout << endl;
}
```

set에서의 사용 예시Permalink

- `set<int> container;`
  - `insert` 사용
- 벡터와 동일한 반복자 문법.

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <map>

using namespace std;

int main()
{
  set<int> container;

  for (int i = 0; i < 10; ++i)
    container.insert(i);

  vector<int>::const_iterator itr;

  for (auto& itr : container)
    cout << itr << "  ";
  cout << endl;
}
```

map에서의 사용 예시Permalink

- `map<int, char> container;`
  - `insert` 사용
    - `make_pair` 이용하여 key와 value 를 함께 `insert` 해주기
- itr → first : *Key*
- itr → second : *Value*

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <map>

using namespace std;

int main()
{
  map<int, char> container;

  for (int i = 0; i < 10; ++i)
    container.insert(make_pair(i, char(i + 65)));

  vector<int>::const_iterator itr;

  for (auto& itr : container)
    cout << itr->first << "  " << itr->second << endl;
  cout << endl;
}
```

💎출력💎

```
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
8 I
9 J
```