

# STL 1

STL & iterator

컴퓨터학부 20152445 강지훈

# Index

STL을 알기 전

STL이란?

Iterator

# STL을 알기 전

STL : Standard Template Library

사용자를 위해 자주 사용되는 자료구조와 알고리즘을 모아놓은 라이브러리

# STL을 알기 전

## 템플릿 Template

자료형은 다르지만 내부 작동은 같은 클래스나 함수를 한번만 작성이 가능


```
#include <stdio.h>

int MaxInt(int a, int b) {
    return a > b ? a : b;
}

float MaxFloat(float a, float b) {
    return a > b ? a : b;
}

char MaxChar(char a, char b) {
    return a > b ? a : b;
}

int main()
{
    printf("%d\n", MaxInt(1, 3)); // 3
    printf("%f\n", MaxFloat(3.2f, 2.3f)); // 3.2
    printf("%c\n", MaxChar('i', 'o')); // o
}
```



```
#include <stdio.h>

template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b;
}

int main()
{
    printf("%f\n", Max(5.9, 3.4)); // 5.9
}
```

# STL을 알기 전

## 템플릿 Template

자료형은 다르지만 내부 작동은 같은 클래스나 함수를 한번만 작성이 가능


```
#include <stdio.h>

int MaxInt(int a, int b) {
    return a > b ? a : b;
}

float MaxFloat(float a, float b) {
    return a > b ? a : b;
}

char MaxChar(char a, char b) {
    return a > b ? a : b;
}

int main()
{
    printf("%d\n", MaxInt(1, 3)); // 3
    printf("%f\n", MaxFloat(3.2f, 2.3f)); // 3.2
    printf("%c\n", MaxChar('i', 'o')); // o
}
```



```
#include <stdio.h>

template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b;
}

int main()
{
    printf("%f\n", Max(5.9, 3.4)); // 5.9
}
```

데이터 타입에 영향을 받지 않으므로, 하나의 템플릿으로 여러 데이터 타입에 적용 가능

**일반화 프로그래밍(Generic Programming)**

# STL을 알기 전

## 일반화 프로그래밍(Generic Programming)

위키백과, 우리 모두의 백과사전.

제네릭 프로그래밍(영어: Generic programming)은 데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있는 기술에 중점을 두어 **재사용성**을 높일 수 있는 프로그래밍 방식이다.

제네릭 프로그래밍은 여러가지 유용한 소프트웨어 컴포넌트들을 체계적으로 융합하는 방법을 연구하는 것으로 그 목적은 알고리즘, 데이터 구조, 메모리 할당 메커니즘, 그리고 기타 여러 소프트웨어적인 장치들을 발전시켜 이들의 재사용성, 모듈화, 사용 편의성을 보다 높은 수준으로 끌어올리고자 하는 것이다.

# STL이란?

STL : **S**tandard **T**emplate **L**ibrary  
재사용성을 높이기 위해 **Template**을 활용한 라이브러리

**A**개 데이터 타입에 따라(int, char, struct ...)  
**B**개 자료구조와(list, vector, stack ...)  
**C**개 알고리즘을 구현(search, sort, delete ..)

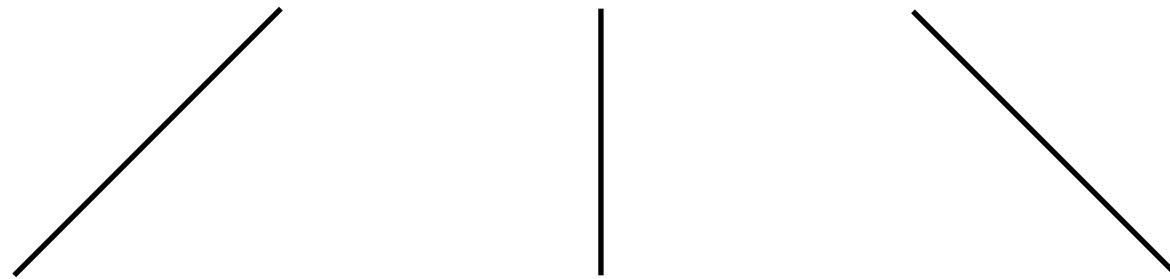


**A \* B \* C**개의 코드를 구현 필요!  
많은 비용을 요구  
이것을 일반화하기 위해 C++에서 STL을 만들어 제공

# STL이란?

STL : **S**tandard **T**emplate **L**ibrary

프로그램에 필요한 자료구조와 알고리즘을 **Template** 형태로 제공



**Container**

동일한 타입의 데이터를  
저장하고 관리

**Iterator**

컨테이너에 저장된 데이터를  
순회, 요소를 읽음

**Algorithm**

템플릿 함수로 구현된  
유용한 알고리즘 라이브러리



# STL이란?

STL : Standard Template Library



## Container

<vector>  
<list>  
<deque>  
  
<stack>, <queue>,  
<priority\_queue>  
  
<set>  
<map> ..

## Iterator

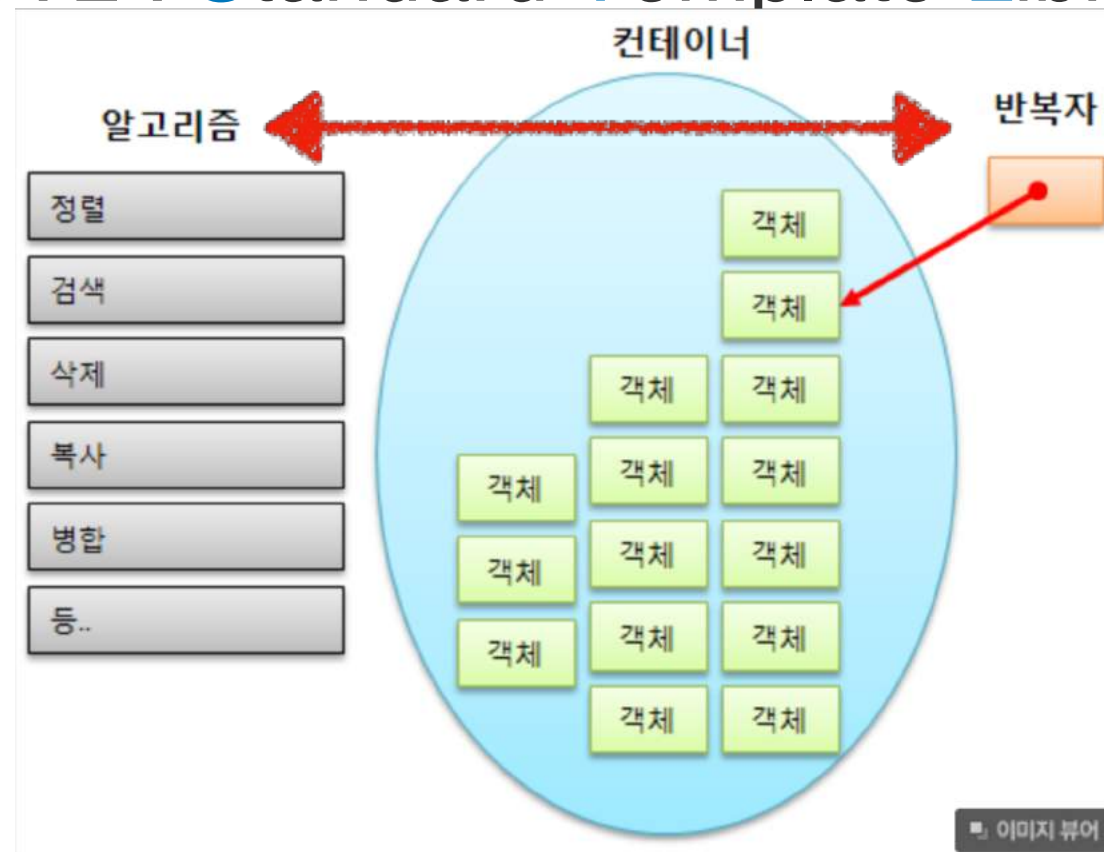
begin()  
end()  
  
rbegin()  
rend()  
  
cbegin()  
cend() ..

## Algorithm

<algorithm>  
sort()  
find()  
binary\_search()  
for\_each()  
transform()  
generate()

# STL의 구성

STL : Standard Template Library



$A * B * C$ 개의 코드를 구현 필요!

-> 컨테이너의 도입으로  $B * C$ 로 구현 가능!

-> 컨테이너에 구속받지 않고  
컨테이너마다 처리방식을 다르게한 반복자를 이용하여  
 $B + C$  개의 코드 사용!

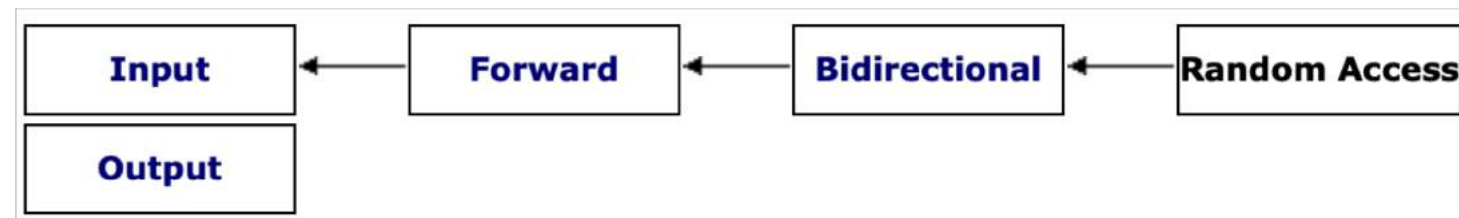
# Iterator

## Iterator 반복자란?

1. 컨테이너에 저장된 객체를 **순회**하고  
각각의 원소에 대한 **접근**을 제공
2. 일종의 포인터를 구현한 객체
3. 컨테이너마다 내부 구조가 다르므로 컨테이너에 맞게 구현 필요  
-> 컨테이너의 타입과는 상관없이 순회하는 과정을 일반화한 표현

# Iterator

입력 반복자(input iterator)	읽기만 가능, 순방향 이동
출력 반복자(output iterator)	쓰기만 가능, 순방향 이동
순방향 반복자(forward iterator)	읽기/쓰기 모두 가능, 순방향 이동
양방향 반복자(bidirectional iterator)	읽기/쓰기 모두 가능, 순방향/역방향 이동
임의 접근 반복자(random access iterator)	읽기/쓰기 모두 가능, 임의접근



컨테이너마다 선언되어있는 **Iterator**의 종류가 다르고  
알고리즘에서 사용하는 **Iterater**의 종류도 다르다!

# Iterator

class template

**std::vector**

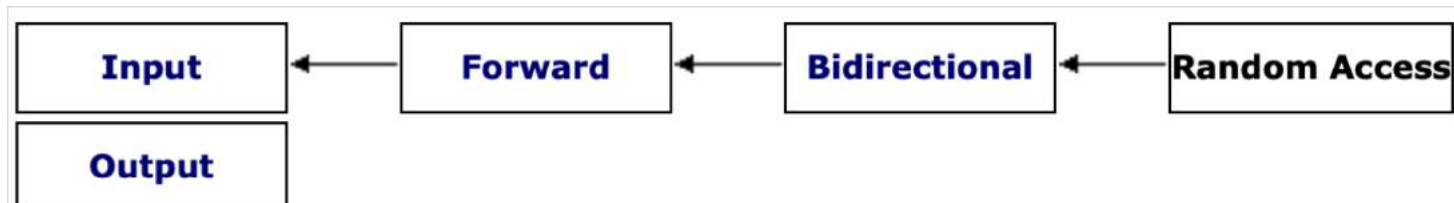
## Member types

C++98

C++11



member type	definition	notes
value_type	The first template parameter (T)	
allocator_type	The second template parameter (Alloc)	defaults to: <code>allocator&lt;value_type&gt;</code>
reference	<code>value_type&amp;</code>	
const_reference	<code>const value_type&amp;</code>	
pointer	<code>allocator_traits&lt;allocator_type&gt;::pointer</code>	for the default <code>allocator</code> : <code>value_type*</code>
const_pointer	<code>allocator_traits&lt;allocator_type&gt;::const_pointer</code>	for the default <code>allocator</code> : <code>const value_type*</code>
iterator	a random access iterator to <code>value_type</code>	convertible to <code>const_iterator</code>
const_iterator	a random access iterator to <code>const value_type</code>	
reverse_iterator	<code>reverse_iterator&lt;iterator&gt;</code>	
const_reverse_iterator	<code>reverse_iterator&lt;const_iterator&gt;</code>	
difference_type	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
size_type	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>



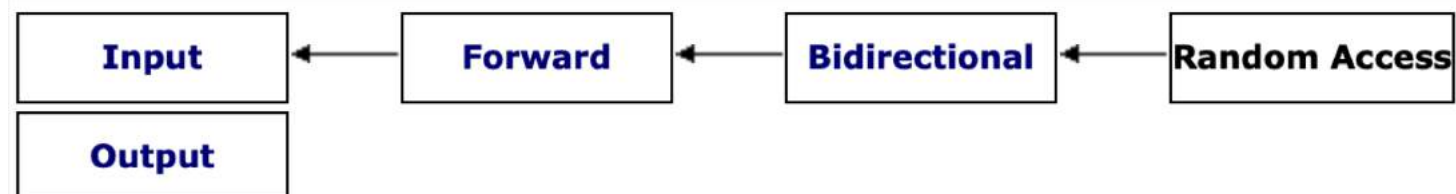
# Iterator

class template  
**std::list**

## Member types

C++98 C++11 ?

member type	definition	notes
value_type	The first template parameter (T)	
allocator_type	The second template parameter (Alloc)	defaults to: <code>allocator&lt;value_type&gt;</code>
reference	<code>value_type&amp;</code>	
const_reference	<code>const value_type&amp;</code>	
pointer	<code>allocator_traits&lt;allocator_type&gt;::pointer</code>	for the default <code>allocator</code> : <code>value_type*</code>
const_pointer	<code>allocator_traits&lt;allocator_type&gt;::const_pointer</code>	for the default <code>allocator</code> : <code>const value_type*</code>
iterator	a <code>bidirectional iterator</code> to <code>value_type</code>	convertible to <code>const_iterator</code>
const_iterator	a <code>bidirectional iterator</code> to <code>const value_type</code>	
reverse_iterator	<code>reverse_iterator&lt;iterator&gt;</code>	
const_reverse_iterator	<code>reverse_iterator&lt;const_iterator&gt;</code>	
difference_type	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
size_type	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>



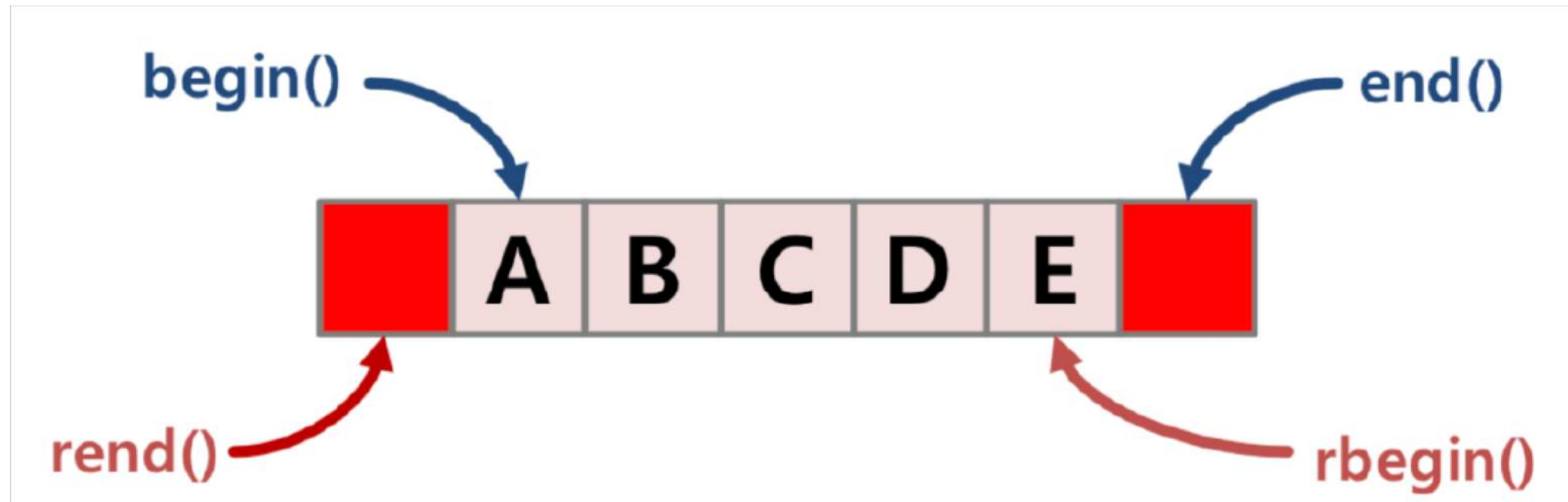


# Iterator

## Iterators:

<b>begin</b>	Return iterator to beginning (public member function )
<b>end</b>	Return iterator to end (public member function )
<b>rbegin</b>	Return reverse iterator to reverse beginning (public member function )
<b>rend</b>	Return reverse iterator to reverse end (public member function )
<b>cbegin</b> <small>C++11</small>	Return const_iterator to beginning (public member function )
<b>cend</b> <small>C++11</small>	Return const_iterator to end (public member function )
<b>crbegin</b> <small>C++11</small>	Return const_reverse_iterator to reverse beginning (public member function )
<b>crend</b> <small>C++11</small>	Return const_reverse_iterator to reverse end (public member function )

# Iterator



`begin()` : 첫 번째 원소를 가리킴  
`end()` : 마지막 원소 다음의 사용하지 않는 부분을 가리킴  
`rbegin()` : 마지막 원소를 가리킴  
`rend()` : 첫 번째 원소 이전의 사용하지 않는 부분을 가리킴

`cbegin()` `cend()` 반복자가 가리키는 값을  
`crbegin()` `crend()` 변경하고 싶지 않을 경우



# Iterator Example

## Iterator in Vector

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

int main(){
    vector<int> v;

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

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

    return 0;          //1 2 3 4
}
```

# Iterator Example

reverse\_iterator / rbegin / rend

```
vector<int>::reverse_iterator it;  
for(it=v.rbegin(); it!=v.rend(); it++){  
    cout<<*it<<" ";  
}
```

# Iterator Example

reverse\_iterator / rbegin / rend

```
vector<int>::reverse_iterator it;
for(it=v.rbegin(); it!=v.rend(); it++){
    cout<<*it<<" ";
}
```

const\_iterator / cbegin / cend

```
vector<int>::const_iterator it;
for(it=v.cbegin(); it!=v.cend(); it++){
    if(*it==4) *it=5;
    cout<<*it<<" ";
}
```



Cannot assign to return value because function 'operator\*' returns a const value

많은 경우 반복자가 가리키는 값을 바꾸지 않고 참조만 하는 경우가 많으므로,  
**const iterator** 를 적절히 이용하는 것을 지향!

# Iterator Example

## Iterator in List

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

int main(){
    list<int> li;

    li.push_back(1);    li.push_back(2);
    li.push_back(3);    li.push_back(4);

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

    return 0;           //1 2 3 4
}
```

# Iterator Example

## Iterator in Set

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

int main(){
    set<int> s;

    s.insert(1);    s.insert(2);
    s.insert(3);    s.insert(4);

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

    return 0;           //1 2 3 4
}
```

# Iterator Example

```
vector<int>::iterator it1;
for(it1=v.begin(); it1!=v.end(); it1++){
    cout<<*it1<<" ";
}

set<int>::iterator it2;
for(it2 = s.begin(); it2!=s.end(); it2++){
    cout<<*it2<<" ";
}

vector<int>::const_reverse_iterator it;
for(it=v.crbegin(); it!=v.crend(); it++){
    cout<<*it<<" ";
}
```

컨테이너, 반복자 의 종류마다 반복자를 새로 정의해줘야한다는 개발 비용 발생

# Iterator Example

```
vector<int>::iterator it1;
for(it1=v.begin(); it1!=v.end(); it1++){
    cout<<*it1<<" ";
}

set<int>::iterator it2;
for(it2 = s.begin(); it2!=s.end(); it2++){
    cout<<*it2<<" ";
}

vector<int>::const_reverse_iterator it;
for(it=v.crbegin(); it!=v.crend(); it++){
    cout<<*it<<" ";
}
```

컨테이너, 반복자 의 종류마다 반복자를 새로 정의해줘야한다는 개발 비용 발생



```
for(auto it=v.begin(); it!=v.end(); it++){
    cout<<*it<<" ";
}
```

**Auto** (c++ 11 부터 지원)

# STL의 장단점

## 장점

1. 일반화를 지원
2. 개발 비용의 단축
3. 실행 효율의 보장

## 단점

1. 숙련되기까지 많은 노력
2. 코드의 비대화



감사합니다