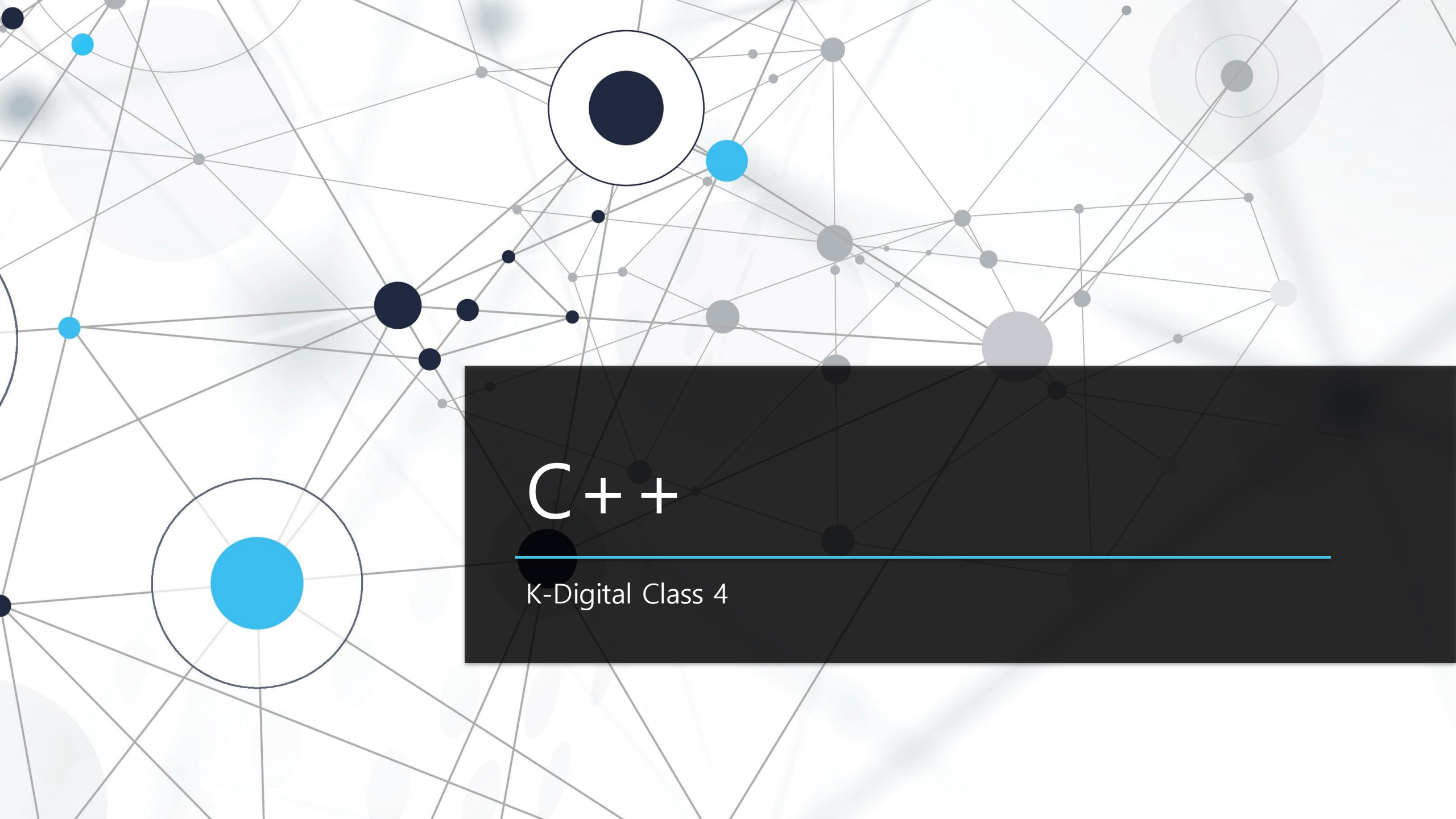


C++

K-Digital Class 4

C++ STL(STANDARD TEMPLATE LIBRARY)

- C++이 가지는 프로그래밍 언어 로서의 특징 중 하나로 일반화 프로그래밍(generic programming)을 들 수 있다.
- 이러한 일반화 프로그래밍은 데이터를 중시하는 객체 지향 프로그래밍과는 달리 프로그램의 알고리즘에 그 중점을 둔다.
- C++ 표준 템플릿 라이브러리인 STL도 이러한 일반화 프로그래밍 패러다임의 한 축을 담당하고 있다.
- STL은 알고리즘을 일반화한 표현을 제공하여, 데이터의 추상화와 코드를 재활용할 수 있게 한다.
- STL은 1994년 휴렛팩커드연구소의 알렉스 스테파노프(Alex Stepanov)와 멩 리(Meng Lee)가 처음으로 그 구현을 발표한다.
- 그 후 STL은 ISO/ANSI C++ 표준 위원회에 의해 C++ 표준 템플릿 라이브러리로 포함되게 된다.

C++ STL(STANDARD TEMPLATE LIBRARY)

STL Components

- 컨테이너(container)
  - 컨테이너는 특정 종류의 객체 컬렉션(Object collection)을 관리하는 데 사용된다.
  - Deque, list, vector, map 등과 같은 여러 유형의 컨테이너가 있다.
- 반복자(iterator)
  - 반복자는 객체 컬렉션(Object collection)의 구성요소를 단계별로 실행하는 데 사용된다.
  - 이러한 컬렉션은 컨테이너 또는 컨테이너의 하위 집합일 수 있다.
- 알고리즘(algorithm)
  - 알고리즘은 컨테이너에서 작동한다. 반복자를 이용해서 컨테이너 내용의 초기화, 정렬, 검색 및 변환을 수행하는 수단을 제공한다.

C++ STL(STANDARD TEMPLATE LIBRARY)

Container

- STL에서 컨테이너(container)는 같은 타입의 여러 객체를 저장하는 일종의 집합이라 할 수 있다.
- 컨테이너는 클래스 템플릿으로, 컨테이너 변수를 선언할 때 컨테이너에 포함할 요소의 타입을 명시할 수 있다.
- C++ STL 에서 컨테이너는 크게 두 가지 종류가 있다. 먼저 배열 처럼 객체들을 순차적으로 보관하는 시퀀스 컨테이너 (sequence container) 와 키(key)를 바탕으로 대응되는 값을 찾아주는 연관 컨테이너 (associative container)가 있다.

C++ STL(STANDARD TEMPLATE LIBRARY)

Sequence container

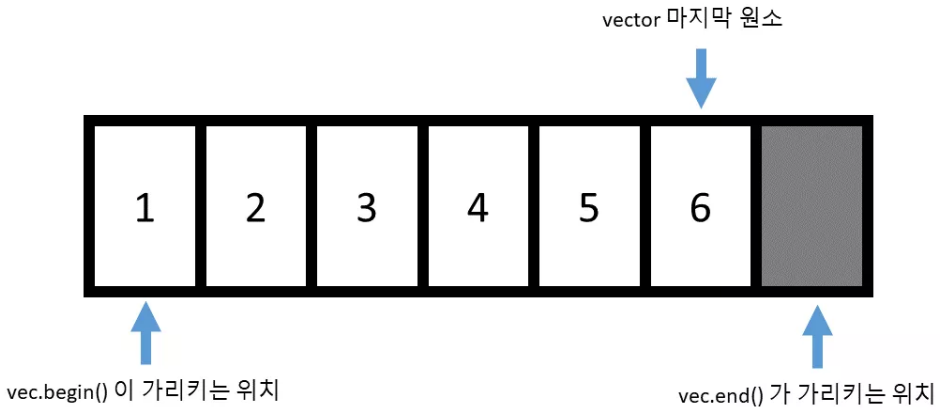
- Sequence container에는 vector, list, deque 이렇게 3개가 정의되어 있다. 먼저 벡터(vector) 의 경우, 쉽게 생각하면 가변길이 배열이라 생각하면 된다
- 벡터에는 원소들이 메모리 상에서 실제로 순차적으로 저장되어 있고, 따라서 임의의 위치에 있는 원소를 접근하는 것을 매우 빠르게 수행할 수 있다.
- vector 의 임의의 원소에 접근하는 것은 배열처럼 [] 를 이용하거나, at 함수를 이용하면 된다. 또한 맨 뒤에 원소를 추가하거나 제거하기 위해서는 push_back 혹은 pop_back 함수를 사용하면 된다.
- 벡터의 크기를 return하는 함수인 size 의 경우, return하는 값의 타입은 size_type 멤버 타입으로 정의되어 있다.

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> vec;
  vec.push_back(10);  // 맨 뒤에 10 추가
  vec.push_back(20);  // 맨 뒤에 20 추가
  vec.push_back(30);  // 맨 뒤에 30 추가
  vec.push_back(40);  // 맨 뒤에 40 추가

  for (std::vector<int>::size_type i = 0; i < vec.size(); i++) {
    std::cout << "vec 의 " << i + 1 << " 번째 원소 :: " << vec[i] << std::endl;
  }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- iterator는 container의 원소에 접근할 수 있는 포인터와 같은 객체라 할 수 있다. 물론 vector의 경우 [ ] 를 이용해서 정수형 변수로 마치 배열 처럼 임의의 위치에 접근할 수 있지만, 반복자를 사용해서도 마찬가지 작업을 수행할 수 있다. 특히 algorithm 라이브러리의 경우 대부분이 반복자를 인자로 받아서 algorithm을 수행한다.
- 반복자는 컨테이너에 iterator 멤버 타입으로 정의되어 있다. vector 의 경우 반복자를 얻기 위해서는 begin() 함수와 end() 함수를 사용할 수 있는데 이는 다음과 같은 위치를 반환한다.



C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- begin() 함수는 예상했던 대로, vector 의 첫번째 원소를 가리키는 반복자를 반환한다. 그런데, 흥미롭게도 end() 의 경우 vector 의 마지막 원소 한 칸 뒤를 가리키는 반복자를 반환하게 된다. 왜 end 의 경우 vector 의 마지막 원소를 가리키는 것이 아니라, 마지막 원소의 뒤를 가리키는 반복자를 반환할까?
- 이에 여러가지 이유가 있겠지만, 가장 중요한 점이 이를 통해 빈 벡터를 표현할 수 있다는 점이다. 만일 begin() == end() 라면 원소가 없는 벡터를 의미 하게 된다. 만약에 vec.end() 가 마지막 원소를 가리킨다면 비어 있는 벡터를 표현할 수 없게 된다.

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);
  vec.push_back(40);

  // 전체 벡터를 출력하기
  for (std::vector<int>::iterator itr = vec.begin(); itr != vec.end(); ++itr) {
    std::cout << *itr << std::endl;
  }

  // int arr[4] = {10, 20, 30, 40}
  // *(arr + 2) == arr[2] == 30;
  // *(itr + 2) == vec[2] == 30;

  std::vector<int>::iterator itr = vec.begin() + 2;
  std::cout << "3 번째 원소 :: " << *itr << std::endl;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 의 반복자의 타입은 std::vector<>::iterator 멤버 타입으로 정의되어 있고, vec.begin() 이나 vec.end() 함수가 이를 반환한다. End() 가 vector 의 마지막 원소 바로 뒤를 가리키기 때문에 for 문에서 vector 전체 원소를 보고 싶다면 vec.end() 가 아닐 때 까지 반복하면 된다.
- 반복자를 마치 포인터처럼 사용한다고 하였는데, 실제로 현재 반복자가 가리키는 원소의 값을 보고 싶다면…

```
std::cout << *itr << std::endl;
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 포인터로 * 를 해서 가리키는 주소 값의 값을 보았던 것처럼, * 연산자를 이용해서 itr 이 가리키는 원소를 볼 수 있다.
- 물론 itr 은 실제 포인터가 아니고 * 연산자를 오버로딩해서 마치 포인터 처럼 동작하게 만든 것이다.
- * 연산자는 itr 이 가리키는 원소의 레퍼런스를 반환한다.

```
std::vector<int>::iterator itr = vec.begin() + 2;
std::cout << "3 번째 원소 :: " << *itr << std::endl;
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 반복자 역시 + 연산자를 통해서 그 만큼 떨어져 있는 원소를 가리키게 할 수 도 있다. (그냥 배열을 가리키는 포인터와 정확히 똑같이 동작한다)
- 반복자를 이용하면 예제와 같이 insert 와 erase 함수도 사용할 수 있다.

```cpp
#include <iostream>
#include <vector>


template <typename T>
void print_vector(std::vector<T>& vec) {
  // 전체 벡터를 출력하기
  for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
       ++itr) {
    std::cout << *itr << std::endl;
  }
}
int main() {
  std::vector<int> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);
  vec.push_back(40);

  std::cout << "처음 벡터 상태" << std::endl;
  print_vector(vec);
  std::cout << "----------------------------" << std::endl;

  // vec[2] 앞에 15 추가
  vec.insert(vec.begin() + 2, 15);
  print_vector(vec);

  std::cout << "----------------------------" << std::endl;
  // vec[3] 제거
  vec.erase(vec.begin() + 3);
  print_vector(vec);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

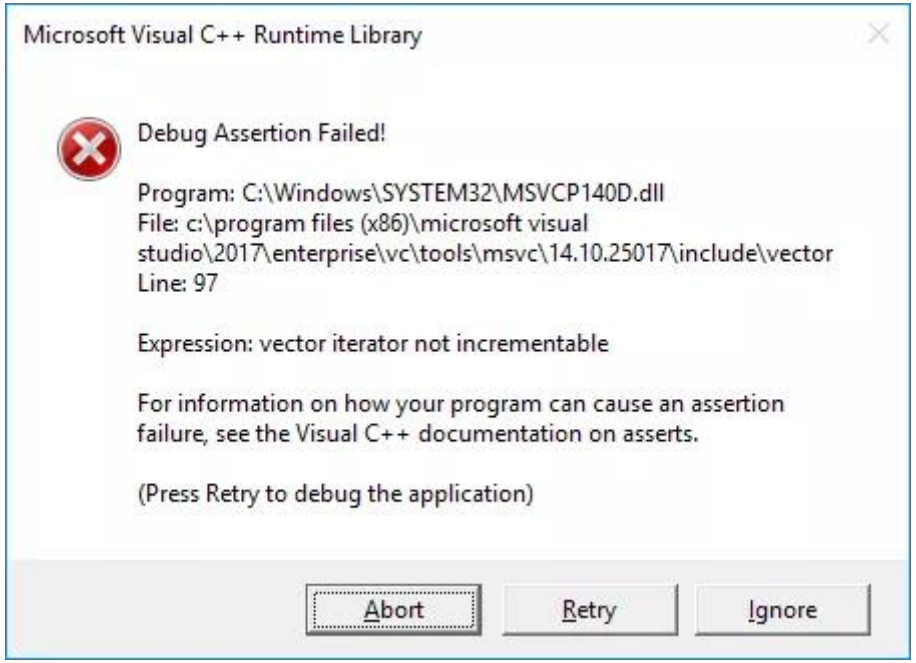
- vector 에서 반복자로 erase 나 insert 함수를 사용할 때 주의해야할 점이 있다.

```cpp
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
  // 전체 벡터를 출력하기
  std::cout << "[ ";
  for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
       ++itr) {
    std::cout << *itr << " ";
  }
  std::cout << "]";
}
int main() {
  std::vector<int> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);
  vec.push_back(40);
  vec.push_back(20);

  std::cout << "처음 벡터 상태" << std::endl;
  print_vector(vec);

  std::vector<int>::iterator itr = vec.begin();
  std::vector<int>::iterator end_itr = vec.end();

  for (; itr != end_itr; ++itr) {
    if (*itr == 20) {
      vec.erase(itr);
    }
  }

  std::cout << "값이 20 인 원소를 지운다!" << std::endl;
  print_vector(vec);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 컴파일 후 실행하였다면 그림과 같은 오류가 발생한다.



C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 왜 이런 오류가 발생하는 것일까?
- 문제는 바로 위 코드에서 발생한다. 컨테이너에 원소를 추가하거나 제거하게 되면 기존에 사용하였던 모든 반복자들을 사용할 수 없게 된다. 다시 말해 위 경우 vec.erase(itr) 을 수행하게 되면 더 이상 itr 은 유효한 반복자가 아니게 되는 것이다. 또한 end_itr 역시 무효화 된다.
- 따라서 itr != end_itr 이 영원히 성립되며 무한 루프에 빠져 위와 같은 오류가 발생한다.
- 결과적으로 코드를 제대로 고치려면 다음과 같이 해야 한다.

```
std::vector<int>::iterator itr = vec.begin();

for (; itr != vec.end(); ++itr) {
  if (*itr == 20) {
    vec.erase(itr);
    itr = vec.begin();
  }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 사실 생각해보면 위 바뀐 코드는 꽤나 비효율적임을 알 수 있다. 왜냐하면 20 인 원소를 지우고, 다시 처음으로 돌아가서 원소들을 찾고 있기 때문이다. 그냥 20 인 원소 바로 다음 위치 부터 찾으면 될 것이기 때문이다.
- 그렇다면 아예 위처럼 굳이 반복자를 쓰지 않고 erase 함수에만 반복자를 바로 만들어서 전달하면 된다.

```
for (std::vector<int>::size_type i = 0; i != vec.size(); i++) {
  if (vec[i] == 20) {
    vec.erase(vec.begin() + i);
    i--;
  }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 사실 생각해보면 위 바뀐 코드는 꽤나 비효율적임을 알 수 있다. 왜냐하면 20 인 원소를 지우고, 다시 처음으로 돌아가서 원소들을 찾고 있기 때문이다. 그냥 20 인 원소 바로 다음 위치 부터 찾으면 될 것이기 때문이다.
- 그렇다면 아예 위처럼 굳이 반복자를 쓰지 않고 erase 함수에만 반복자를 바로 만들어서 전달하면 된다.
- 사실 이방법은 그리 권장하는 방법은 아니다. 기껏 원소에 접근하는 방식은 반복자를 사용하는 것으로 통일하였는데, 위 방법은 이를 모두 깨 버리고 그냥 기존의 배열 처럼 정수형 변수 i 로 원소에 접근하는 것이기 때문이다.
- Algorithm library에서 다시 하도록 한다.

```
for (std::vector<int>::size_type i = 0; i != vec.size(); i++) {
  if (vec[i] == 20) {
    vec.erase(vec.begin() + i);
    i--;
  }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 에서 지원하는 반복자로 const_iterator 가 있다. 이는 마치 const pointer를 생각하면 된다.
- 즉, const_iterator 의 경우 가리키고 있는 원소의 값을 바꿀 수 없다.

```cpp
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
  // 전체 벡터를 출력하기
  for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
       ++itr) {
    std::cout << *itr << std::endl;
  }
}
int main() {
  std::vector<int> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);
  vec.push_back(40);

  std::cout << "초기 vec 상태" << std::endl;
  print_vector(vec);

  // itr 은 vec[2] 를 가리킨다.
  std::vector<int>::iterator itr = vec.begin() + 2;

  // vec[2] 의 값을 50으로 바꾼다.
  *itr = 50;

  std::cout << "---------------" << std::endl;
  print_vector(vec);

  std::vector<int>::const_iterator citr = vec.cbegin() + 2;

  // 상수 반복자가 가리키는 값은 바꿀수 없다. 불가능!
  *citr = 30;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 에서 지원하는 반복자로 const_iterator 가 있다. 이는 마치 const pointer를 생각하면 된다.
- 즉, const_iterator 의 경우 가리키고 있는 원소의 값을 바꿀 수 없다.

```
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
  // 전체 벡터를 출력하기
  for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
       ++itr) {
    std::cout << *itr << std::endl;
  }
}
int main() {
  std::vector<int> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);
  vec.push_back(40);

  std::cout << "초기 vec 상태" << std::endl;
  print_vector(vec);

  // itr 은 vec[2] 를 가리킨다.
  std::vector<int>::iterator itr = vec.begin() + 2;

  // vec[2] 의 값을 50으로 바꾼다.
  *itr = 50;

  std::cout << "---------------" << std::endl;
  print_vector(vec);

  std::vector<int>::const_iterator citr = vec.cbegin() + 2;

  // 상수 반복자가 가리키는 값은 바꿀수 없다. 불가능!
  *citr = 30;
}
```

⚠ 컴파일 오류

```
'citr': you cannot assign to a variable that is const
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- const 반복자가 가리키고 있는 값은 바꿀 수 없다고 오류가 발생한다. 주의할 점은, const 반복자의 경우 cbegin() 과 cend() 함수를 이용하여 얻을 수 있다.
- 많은 경우 반복자의 값을 바꾸지 않고 참조만 하는 경우가 많으므로, const iterator 를 적절히 이용하는 것이 좋다.

```
std::vector<int>::const_iterator citr = vec.cbegin() + 2;
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

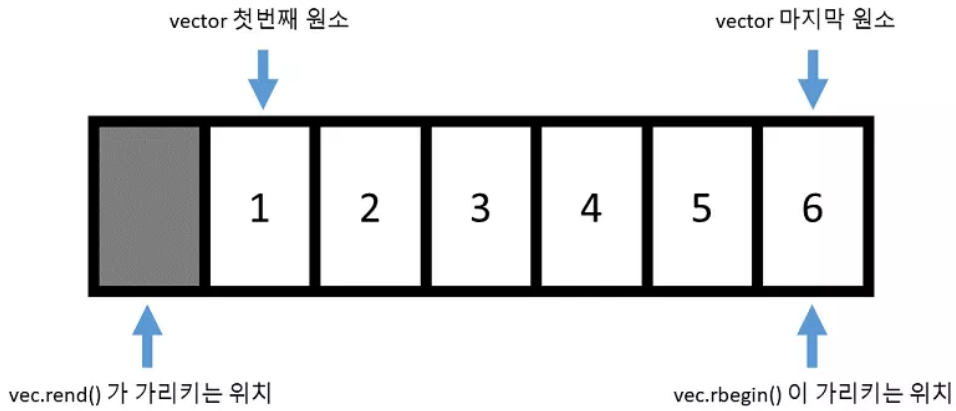
- vector 에서 지원하는 반복자 중 마지막 종류로 역반복자 (reverse iterator) 가 있다.
- 이는 반복자와 똑같지만 벡터 뒤에서 부터 앞으로 거꾸로 간다는 특징이 있다.

```cpp
#include <iostream>
#include <vector>

template <typename T>
void print_vector(std::vector<T>& vec) {
  // 전체 벡터를 출력하기
  for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
       ++itr) {
    std::cout << *itr << std::endl;
  }
}
int main() {
  std::vector<int> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);
  vec.push_back(40);

  std::cout << "초기 vec 상태" << std::endl;
  print_vector(vec);

  std::cout << "역으로 vec 출력하기!" << std::endl;
  // itr 은 vec[2] 를 가리킨다.
  std::vector<int>::reverse_iterator r_iter = vec.rbegin();
  for (; r_iter != vec.rend(); r_iter++) {
    std::cout << *r_iter << std::endl;
  }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 앞에서 반복자의 end() 가 맨 마지막 원소의 바로 뒤를 가리켰던 것처럼, 역반복자의 rend() 역시 맨 앞 원소의 바로 앞을 가리키게 된다. 또한 반복자의 경우 값이 증가하면 뒤쪽 원소로 가는 것처럼, 역반복자의 경우 값이 증가하면 앞쪽 원소로 가게 된다.
- 또 반복자가 상수 반복자가 있는 것처럼 역반복자 역시 상수 역반복자가 있다. 그 타입은 const_reverse_iterator 타입이고, crbegin(), crend() 로 얻을 수 있다.



C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 역반복자를 사용하는 것은 매우 중요하다.

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> vec;
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);

  // 끝에서 부터 출력하기
  for (std::vector<int>::size_type i = vec.size() - 1; i >= 0; i--) {
    std::cout << vec[i] << std::endl;
  }

  return 0;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 그림과 오류가 발생하게 된다. 맨 뒤의 원소 부터 제대로 출력하는 코드 같은데 왜 이런 문제가 발생하였을까? 그 이유는 vector 의 index 를 담당하는 타입이 부호 없는 정수 이기 때문이다. 따라서 i 가 0 일 때 i – 를 하게 된다면 -1 이 되는 것이 아니라, 해당 타입에서 가장 큰 정수가 되는 것이다.
- 따라서 for 문이 영원히 종료할 수 없게 되므로, 이 문제를 해결하기 위해서는 부호 있는 정수로 선언해야 하는데, 이 경우 vector 의 index 타입과 일치하지 않아서 타입 캐스팅을 해야 한다는 문제가 발생하게 된다.
- 따라서 가장 현명한 선택으로는 역으로 원소를 참조하고 싶다면, 역반복자를 사용하는 것이다.

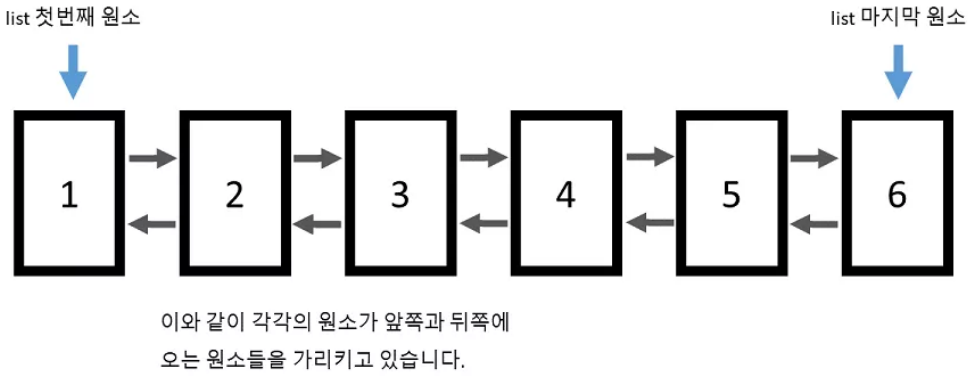
실행 결과

```
3
2
1
// ... (생략) ...
0
0
0
1
0
593
0
0
[1]    22180 segmentation fault (core dumped)  ./test
```

C++ STL(STANDARD TEMPLATE LIBRARY)

list

- 리스트(list) 의 경우 양방향 연결 구조를 가진 자료형이라 볼 수 있다.
- vector 와는 달리 임의의 위치에 있는 원소에 접근을 바로 할 수 없다. list 컨테이너 자체에서는 시작 원소와 마지막 원소의 위치만을 기억하기 때문에, 임의의 위치에 있는 원소에 접근하기 위해서는 하나씩 링크를 따라가야 한다.
- 그래서 리스트에는 아예 [] 나 at 함수가 아예 정의되어 있지 않다.



C++ STL(STANDARD TEMPLATE LIBRARY)

list

- 리스트(list) 의 경우 양방향 연결 구조를 가진 자료형이라 볼 수 있다.
- vector 와는 달리 임의의 위치에 있는 원소에 접근을 바로 할 수 없다. list 컨테이너 자체에서는 시작 원소와 마지막 원소의 위치만을 기억하기 때문에, 임의의 위치에 있는 원소에 접근하기 위해서는 하나씩 링크를 따라가야 한다.
- 그래서 리스트에는 아예 [] 나 at 함수가 아예 정의되어 있지 않다.

```cpp
#include <iostream>
#include <list>

int main() {
  std::list<int> lst;

  lst.push_back(10);
  lst.push_back(20);
  lst.push_back(30);
  lst.push_back(40);

  for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
    std::cout << *itr << std::endl;
  }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

list

- 리스트의 반복자의 경우 다음과 같은 연산밖에 수행할 수 없다.
  - itr++, itr—
  - itr * 5 같은 연산은 불가능하다. 즉 임의의 위치에 있는 원소를 가리킬 수 없다. 반복자는 오직 하나씩 밖에 움직일 수 없다.
  - 이와 같은 이유는 list 의 구조를 생각해보면 알 수 있습니다. 앞서 말했듯이 리스트는 왼쪽 혹은 오른쪽을 가리키고 있는 원소들의 모임으로 이루어져 있기 때문에, 한 번에 한 칸 씩 밖에 이동할 수 없다. 즉, 메모리 상에서 원소들이 연속적으로 존재하지 않을 수 있다는 뜻이다.
  - 반면에 벡터의 경우 메모리 상에서 연속적으로 존재하기 때문에 쉽게 임의의 위치에 있는 원소를 참조할 수 있다.

```cpp
#include <iostream>
#include <list>

int main() {
  std::list<int> lst;

  lst.push_back(10);
  lst.push_back(20);
  lst.push_back(30);
  lst.push_back(40);

  for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
    std::cout << *itr << std::endl;
  }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

list

- list 에서 정의되는 반복자의 타입은 BidirectionalIterator 타입이다. 이름에서도 알 수 있듯이 양방향으로 이동할 수 있되, 한 칸 씩 밖에 이동할 수 없다. 반면에 벡터에서 정의되는 반복자의 타입은 RandomAccessIterator 타입이다.
- 즉, 임의의 위치에 접근할 수 있는 반복자 이다. (참고로 RandomAccessIterator 는 BidirectionalIterator 를 상속받고 있다)
- 리스트의 반복자는 BidirectionalIterator 이기 때문에 ++ 과 -- 연산만 사용 가능하다. 따라서 for 문으로 하나, 하나 원소를 확인 해보는 것은 가능하다.
- 마찬가지로 erase 함수를 이용하여 원하는 위치에 있는 원소를 지울 수 도 있다. 리스트의 경우는 벡터와는 다르게, 원소를 지워도 반복자가 무효화 되지 않는다. 왜냐하면, 각 원소들의 주소 값들은 바뀌지 않기 때문이다.

```cpp
#include <iostream>
#include <list>

template <typename T>
void print_list(std::list<T>& lst) {
  std::cout << "[ ";
  // 전체 리스트를 출력하기 (이 역시 범위 기반 for 문을 쓸 수 있습니다)
  for (const auto& elem : lst) {
    std::cout << elem << " ";
  }
  std::cout << "]" << std::endl;
}
int main() {
  std::list<int> lst;

  lst.push_back(10);
  lst.push_back(20);
  lst.push_back(30);
  lst.push_back(40);

  std::cout << "처음 리스트의 상태 " << std::endl;
  print_list(lst);

  for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
    // 만일 현재 원소가 20 이라면
    // 그 앞에 50 을 집어넣는다.
    if (*itr == 20) {
      lst.insert(itr, 50);
    }
  }

  std::cout << "값이 20 인 원소 앞에 50 을 추가 " << std::endl;
  print_list(lst);

  for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
    // 값이 30 인 원소를 삭제한다.
    if (*itr == 30) {
      lst.erase(itr);
      break;
    }
  }

  std::cout << "값이 30 인 원소를 제거한다" << std::endl;
  print_list(lst);
}
```