

STL - 6주 1일

Wulong

6주 강의 2020. 4. 20 ~ 2020. 4. 24	
<p>지난 주 4.13 ~ 4.17</p>	<p>STL 컨테이너</p> <p>메모리 벡터의 메모리 관리</p> <p>벡터와 클래스 객체 컨테이너에서 객체의 복사와 이동 벡터의 원소 제거</p> <p>deque</p>
<p>이번 주 4.20 ~ 4.24</p>	<p>STL 컨테이너</p> <p>forward_list list list만의 멤버함수 원소 제거/ merge와 splice / sort와 unique</p> <p>순차 컨테이너를 사용하는 실습 문제</p>
<p>다음 주 4.27 ~ 5.1</p>	<p>STL 반복자</p> <p>반복자의 종류 iterator_traits 제네릭 함수 작성</p> <p>String을 표준 컨테이너로 만들기</p> <p>STL 컨테이너 map</p>

지난 주

안녕하세요? STL 수강생 여러분!

메모리 그림을 잘 이해하실 수 있었죠? 프로그램을 실행시켜 주소를 잘 관찰하였다면 바로 알 수 있었을 겁니다. 메모리를 이해하지 못하고 C++ 프로그램을 작성하면 안됩니다. 물론 정확하게 몰라도 실행이 되는 프로그램을 만들 수는 있지만 그것이 아무런 문제없이 실행되리라 기대하면 안됩니다. 잠깐 ctrl+F5 눌러 실행하고 바로 끝나는 프로그램이라면 사실 큰 문제가 될 것은 없습니다. 오랫동안 실행되는 프로그램에서 메모리에 문제가 있으면 언젠가 반드시 프로그램 꺾다 켜야 합니다.

지난 주 강의에서는 여러분들이 잘 이해하지 못했거나 의심나는 부분을 찾아 적극적으로 질문 해주셔서 재미가 있었습니다. 강의록에 설명 안하고 건너뛴 부분이 있는데 그걸 또 예리하게 찾아 질문 해주니 강의자료 만든 보람이 있더라고요. “이거 혹시 이상하게 생각하는 학생이 있을까?” 생각하며 적어 놓은걸 찾아 질문한 학생들 모두 칭찬합니다.

순차컨테이너를 공부하는 중입니다. C++20 표준 초안 문서에서는 22장에서 컨테이너 라이브러리라는 제목으로 모든 컨테이너를 서술하고 있습니다. 언어 표준은 간결하기 이를 데 없습니다. C++ 언어 표준문서는 컨테이너를 이렇게 정의합니다.

Containers are objects that store other objects.

컨테이너는 다른 객체를 담을 수 있는 객체이다. 진짜 간결합니다. 이 정의에 따르면 **vector<int>**는 객체인 int를 담을 수 있는 객체입니다. 따라서 **vector<vector<int>>**도 얼마든지 가능한 것입니다.

순차 컨테이너 마다 복잡도에 따른 특성이 다르므로 프로그래머는 다음 사항을 고려하여 컨테이너를 선택해야 합니다. 기본으로 vector를 사용합니다. 컴파일 시간에 크기(사용할 원소의 개수)를 알 수 있다면 array를 고려합니다. 데이터의 중간 부분에서 빈번한 삽입·삭제가 필요하다면 이번에 알아볼 forward_list와 list를 사용해야 합니다. 대부분의 삽입·삭제 동작이 데이터의 시작과 끝 부분에서 일어난다면 deque을 선택합니다. **vector가 최상의 컨테이너라는 것을 명심**하세요. 그래서 나도 vector를 설명하는데 많은 공을 들였습니다.

아직 C++20 표준이 완성되지는 않았지만 컨테이너 부분이 초안에서 크게 바뀔 것은 없을 겁니다. array, deque, vector까지 공부했는데 C++20 표준의 컨테이너 순서는 array, deque, forward_list, list 그리고 vector입니다. C++17에서도 같은 순서였는데 왜 vector를 제일 뒤에 배치했는가에 대한 설명이 표준에는 없습니다.

C++17부터는 **contiguous container(연속 컨테이너)**라는 새로운 용어를 도입하여 컨테이너를 새롭게 분류하는데 **array와 vector는 contiguous 컨테이너**이지만 deque은 그렇지 않습니다. 지난 주 강의에서 deque의 메모리가 연속되어 있지 않았다는 것을 확인해 봤습니다.

이번 시간에는 `forward_list`와 `list`를 알아봅니다. 우리 책에는 `list`를 먼저 설명하고 있지만 `forward_list`를 먼저 설명하겠습니다. 그런데 어떤 문제를 해결하는데 `forward_list`를 사용한 본 적은 없었습니다. 컨테이너의 특성을 잘 나타낼 예제를 중심으로 설명하니 꼭 책 보고 같이 공부하시기 바랍니다.

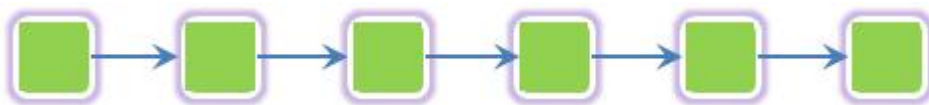
forward_list

컨테이너의 원소들을 순회할 수 있는 반복자는 특성에 따라 다섯 가지로 나뉩니다. 연속 컨테이너(contiguous container)의 원소들은 랜덤하게 순회([] 연산자를 제공)할 수 있으므로 연속 컨테이너가 제공하는 반복자는 랜덤반복자(random access iterator)라고 합니다. 연속 컨테이너는 아니지만 `deque`의 반복자도 랜덤반복자입니다. `deque`에서도 [] 연산자를 사용할 수 있으니까요. 물론 클래스 내부에서는 몇 단계를 거쳐 실제 원소에 접근해야 할 때도 있습니다.

`forward_list`는 각 원소를 노드에 저장하고 한 쪽으로만 원소를 순회하는 구조이기 때문에 순방향 반복자(forward iterator)를 제공합니다. `forward_list`는 저장하고 있는 원소의 어느 곳에서든 $O(1)$ 복잡도로 원소의 삽입과 삭제를 할 수 있습니다. 표준에서는 `forward_list`를 이렇게 설명합니다.

직접 작성한 C 스타일의 단일 연결 리스트(singly linked list)와 비교했을 때 공간이나 시간 면에서 조금의 군더더기도 없다.

이 설명과 어긋나는 기능은 아예 제공하지 않습니다. `forward_list`는 `list`에 비하여 메모리를 덜 사용하면서도 속도는 빠릅니다.



`forward_list`는 컨테이너의 원소들을 순방향(전진 방향)으로만 순회할 수 있습니다. 원소들의 메모리가 서로 떨어져 있으므로 특정한 원소에 접근하기 위해서는 한 번에 하나씩 전진하면서 훑어 나아가야 합니다. `forward_list`는 원소의 개수가 몇 개인지 알려주는 멤버가 없습니다. 원소의 개수가 필요하다면, 제일 처음의 원소를 가리키는 반복자를 `begin()`으로 얻어 제일 마지막 원소의 다음 위치를 가리키는 `end()`에 도달할 때까지 하나씩 전진하며 원소의 개수를 세는 수밖에 없습니다. 반복자가 앞으로만 가기 때문에 항상 현재 반복자 위치 다음에 원소의 삽입과 같은 동작들이 일어납니다. 그래서 멤버함수의 이름에 `_after`가 붙습니다.

다른 모든 컨테이너와 같이 `forward_list`에서도 `begin()` 멤버 함수가 제일 앞 원소를 가리키기 때문에 어떤 동작이 일어날 위치가 제일 앞이 되어야 한다면 `begin` 이전의 위치를 가리킬 수 있어야 합니다. 이런 용도로 `forward_list`는 `before_begin`이라는 멤버 함수를 제공합니다. 원소를 삽입할 때는 이와 똑같은 동작을 `emplace_front()` 멤버 함수로 할 수 있습니다.

독특한 `forward_list`를 설명하기 위해 역지로 문제를 생각해 봤습니다. 실제 사용빈도도 높지 않고 굳이 이런 동작을 할 필요도 없겠지만 구조를 이해하기 위한 것이라고 생각해 주세요,

`forward_list`의 제일 뒤에 원소를 추가하는 동작은 쉽지 않습니다. 대부분의 컨테이너에서 제일 마지막 원소를 가리키는 위치는 `end()` 함수로 얻은 위치에서 -1한 위치입니다. 그런데 `forward_list`는 `end()` 위치를 알 수는 있지만 거기에서 -1한 위치 즉, 진짜 원소가 있는 한 칸 뒤로 갈수는 없습니다. 따라서 원소의 뒤에 뭔가를 추가하려면 밑의 코드와 같이 제일 앞에서 원소의 개수만큼 전진해 나아가야 합니다. `forward_list`에는 `push_back()`과 같은 함수가 아예 없습니다.

한편 원소의 개수도 `size` 멤버가 없기 때문에 그냥 구할 수 있는 것이 아니고 설명한 바와 같이 앞에서 부터 한 칸씩 세 나아가야 합니다. `forward_list`를 쓰고 싶은 마음이 저절로 사라지지 않습니까? 이 컨테이너가 이렇게 되어 있는 이유는 제일 앞에 밝힌 것과 같습니다. 필수 기능 구현에 불필요한 내용은 아예 제공하지 않기 때문입니다.

다행히도 제일 뒷자리를 찾아 원소를 삽입했다면 삽입하는데 사용량도 멤버함수는 삽입한 원소를 가리키는 반복자를 리턴합니다. 따라서 다음번에 원소를 추가하는 동작은 상수시간에 가능하게 됩니다. 사용해 본 적은 없지만 그래도 예를 들지 않을 수는 없어 이와 같은 프로그램을 작성해 보았습니다.

```
// forward_list의 제일 뒤에 원소를 삽입한다

#include <iostream>
#include <forward_list>
#include "String.h"
using namespace std;

int main( )
{
    forward_list<String> numbers { "1", "22", "333" };

    auto bb = numbers.before_begin( );           // bb는 "1"이전의 위치를 가리킴
    numbers.insert_after( bb, "0" );             // 제일 앞에 "0" 삽입

    numbers.emplace_front( "-1" );               // 제일 앞에 -1 생성삽입

    int size = distance( numbers.cbegin( ), numbers.cend( ) ); // 원소의 갯수
    forward_list<String>::iterator p { numbers.begin( ) };
}
```

```

advance( p, size-1 ); // String("333")을 가리키도록 p를 이동

auto last = numbers.emplace_after( p, "4444" );
// p 다음에 String("4444")를 생성삽입

// 함수의 반환값을 이용하면 계속 원소를 뒤에 추가할 수 있다
last = numbers.emplace_after( last, "55555" );
// 제일뒤에 String("55555")를 생성삽입

cout << "관찰메시지 없음" << endl;
cout << endl;

cout << "forward_list가 담을 수 있는 최대 원소 개수: "
    << numbers.max_size( ) << endl;
cout << "forward_list의 원소 수: "
    << distance( numbers.begin( ), numbers.end( ) ) << endl;

cout << endl;
cout << "forward_list의 원소 출력" << endl;
for ( auto& number : numbers )
    cout << number << ' ';
cout << endl;
cout << endl;
}

```

빨간색으로 표시한 것 중에 advance와 distance는 반복자와 관련된 전역 함수입니다.

```

Microsoft Visual Studio 디버깅 콘솔
관찰메시지 없음

forward_list가 담을 수 있는 최대 원소 개수: 268435455
forward_list의 원소 수: 7

forward_list의 원소 출력
-1 0 1 22 333 4444 55555

```

forward_list의 최대 원소 개수는 같은 타입의 원소를 담는 vector나 deque와 비교해 볼 때 더 적을 수밖에 없습니다. 왜냐하면 리스트의 노드는 원소만을 담지는 않기 때문입니다.

다음에 알아볼 list와 같이 forward_list도 sort, unique, splice, merge와 같이 리스트 구조에 따른 특수한 멤버함수가 있습니다. 이들은 list에서 설명하도록 하고 forward_list는 이만 넘어가겠습니다.

list



STL의 list는 이중 연결 리스트입니다. 원소를 한 번에 하나씩 앞뒤로 순회할 수 있는 양방향 반복자(bidirectional iterator)를 제공하며 어느 위치에서든 상수 시간에 원소를 삽입·삭제할 수 있습니다. vector나 deque과 달리 항상 상수 시간에 원소에 접근할 수 있는 랜덤반복자를 제공하지는 않지만 많은 알고리즘 함수는 순차적으로 액세스할 수만 있으면 충분하기 때문에 문제될 것은 없습니다.

원소를 위의 그림과 같이 노드로 관리하는 list는 노드의 장점을 활용한 멤버 함수를 제공합니다. 이 함수들은 다음 다섯 개이며 설명은 책을 보고 여기서는 프로그램을 작성하여 동작을 알아보도록 하겠습니다. 관련 정보를 찾아보며 코딩 결과를 보면 빨리 이해할 수 있습니다.

- remove - 원소를 제거하기
- splice와 merge - 원소들을 잘라 붙이고 병합하기
- sort와 unique - 정렬하고 중복 원소들을 제거하기

제일 먼저 list에서 원소를 제거하는 remove를 알아보니다. 코드를 그대로 입력해 주세요.

```
// list에서 remove로 원소를 제거하기

include <iostream>
#include <list>
#include "String.h"
using namespace std;

int main( )
{
    list<String> words { "공무원", "고강도", "거리두기", "한주", "더..",
                        "교실", "수업은", "시기상조" };

    // 여기에서 "고강도"를 제거하자

    for ( auto i = words.cbegin( ); i != words.cend( ); ++i )
        cout << *i << endl;
}
```

여기서는 list의 전체 원소를 화면 출력하는데 지금까지와는 달리 반복자를 사용하였습니다. 물론 전체 원소를 출력하려면 범위 기반의 for를 사용하면 됩니다. 그렇지만 일부분만을 출력하고 싶거나 반복자 구간을 인자로 받는 함수를 작성하려면 반복자 사용법에 익숙해야 합니다. 역방향 반복자를 사용하면 원소를 거꾸로 순회할 수도 있습니다.

```
for ( auto i = words.cbegin( ); i != words.cend( ); ++i )
    cout << *i << endl;
```

이 코드에서 i의 자료형은 POD 타입이 아닙니다. i의 자료형을 다음과 같이 화면 출력해 볼 수 있습니다. 아마 알아보기 어려울 겁니다. 포인터의 동작과 유사하게 * 연산자(indirection operator)로 원소의 값을 access할 수도 있습니다.

```
auto i = words.cbegin( );
cout << typeid(i).name( ) << endl;
```

실제 프로그램에서 직접 자료형을 쓴다면 이렇게 쓰면 됩니다.

```
list<String>::const_iterator i = words.cbegin( );
```

이 i의 자료형이 바로 auto가 추론한 자료형입니다. auto가 있어 이렇게 직접 쓰지 않아도 되니 다행입니다만 진짜 자료형이 이렇다는 것을 알고 있어야 합니다. const_iterator가 list<String>의 멤버임을 명심하세요. cbegin()은 begin()과 다릅니다. 원소를 가리키는 반복자입니다만 원소의 값을 읽기만 할 수 있는 반복자입니다.

```
i != words.cend( );
```

그 다음은 for 루프의 종료조건입니다. 랜덤반복자와는 달리 list의 반복자로는 비교연산을 할 수 없습니다. 따라서 i가 cend()와 같은 지 같지 않은 지를 비교할 수 있을 뿐입니다.

```
++i;
```

i가 POD 타입이 아니기 때문에 i++를 사용하는 것보다는 ++i를 사용합니다. 항상 ++i를 사용하면 되겠습니다. 그럼, 실습 문제를 잠깐 해 보겠습니다. 여러분이 직접할 수 있는 문제이니 먼저 직접 해결해 보세요.

[실습] words에서 String("고강도")를 제거하라. (5분)

```
#include <iostream>
#include <list>
#include <algorithm>
#include "String.h"
using namespace std;
```

```

int main( )
{
    list<String> words { "공무원", "고강도", "거리두기", "한주",
                        "더..", "교실", "수업은", "시기상조" };

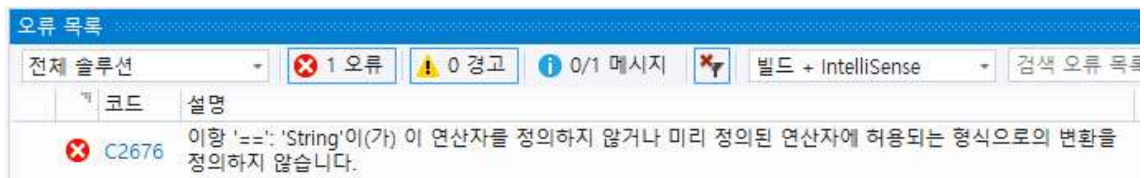
    // 여기에서 "고강도"를 제거하자

    words.erase( remove( words.begin( ), words.end( ), String("고강도") ), words.end() );

    for ( auto i = words.cbegin( ); i != words.cend( ); ++i )
        cout << *i << endl;
}

```

밑줄과 같이 코딩했으면 지금까지의 강의를 잘 이해하고 있는 것입니다. 칭찬합니다. 그런데 이 프로그램을 컴파일하니 이런 메시지가 나오고 실패합니다.



무슨 뜻인지 아시겠습니까? 알고리즘 함수 remove가 words의 [begin(), end()) 구간에서 String("고강도")와 **같은** 원소를 지우려고 합니다. 그런데 컴파일러는 같다는 판단을 못하겠다고 컴파일을 거부합니다.

프로그램에서 같다는 것은 어떤 의미입니까? String은 클래스입니다. 두 String 객체가 같은 가는 == 연산자의 결과로 판단하게 됩니다. 그럼 어떻게 하면 될까요? 그렇습니다. “이것이 같은 것이다”라고 알려주면 되는 것입니다.

```

// String.h 파일

std::string getString( ) const;

// 비교연산자
// 2020. 4. 18 추가
bool operator==( const String& rhs ) const noexcept;

friend std::ostream& operator<<( std::ostream& os, const String& s );

```



```
// String.cpp 파일

// 2020. 4. 18 추가
bool String::operator==( const String& rhs ) const noexcept {
    if ( len == rhs.len ) {
        for ( int i = 0; i < len; ++i ) {
            if ( p[i] != rhs.p[i] )
                return false;
        }
        return true;
    }
    return false;
}
```

프로그램을 다시 실행해 보세요. 오류 메시지 없이 컴파일 성공하고 원하는 원소가 제거되었음을 관찰할 수 있습니다. 그런데 list는 이렇게 프로그램할 필요가 없습니다. 이런 방식으로 프로그램하면 안됩니다. 지금 알아보고 있는 것이 “list는 자신만의 제거 함수 remove가 있다”라는 것을 잊지 않고 있죠?

list에서 원소 제거는 다음과 같이 합니다. 쉽게 알아볼 수 있도록 list<int>를 사용하여 예를 들어 보겠습니다.

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

int main( )
{
    default_random_engine engine;
    uniform_int_distribution<> uid { 0, 10 };

    list<int> numbers;

    for ( int i = 0; i < 100; ++i )
        numbers.emplace_back( uid( engine ) );

    for ( auto i = numbers.cbegin( ); i != numbers.cend( ); ++i )
        cout << *i << ' ';
    cout << endl;

    // numbers에서 0을 제거하라
    numbers.remove( 0 );
```

```

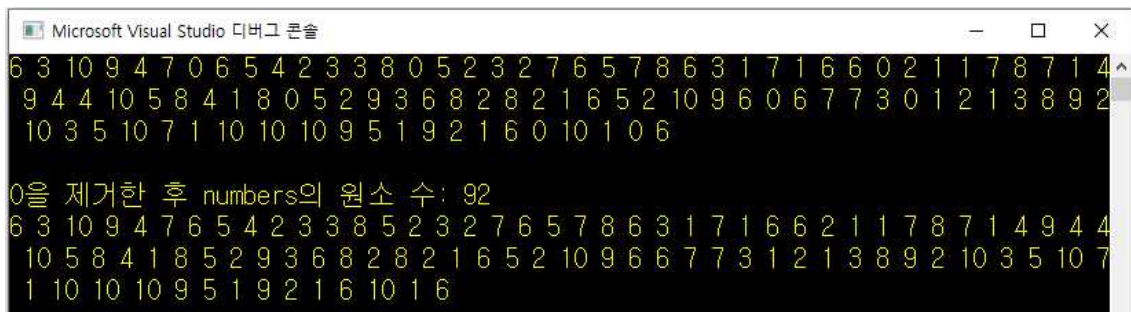
    cout << endl;
    cout << "0을 제거한 후 numbers의 원소 수: " << numbers.size( ) << endl;
    for ( auto i = numbers.cbegin( ); i != numbers.cend( ); ++i )
        cout << *i << ' ';
    cout << endl;
}

```

아래 문장은 균일한 분포(uniform distribution)를 갖도록 [0, 10]까지의 정수를 100개 생성하면 각 int가 생성될 확률은 0.11(11/100)입니다. 개수로는 각각 9개가 생성될 것이라고 기대할 수 있습니다.

```
uniform_int_distribution<> uid { 0, 10 };
```

아래의 출력화면에서 8개의 0이 제거되었음을 확인할 수 있습니다.



```

Microsoft Visual Studio 디버그 콘솔
6 3 10 9 4 7 0 6 5 4 2 3 3 8 0 5 2 3 2 7 6 5 7 8 6 3 1 7 1 6 6 0 2 1 1 7 8 7 1 4
9 4 4 10 5 8 4 1 8 0 5 2 9 3 6 8 2 8 2 1 6 5 2 10 9 6 0 6 7 7 3 0 1 2 1 3 8 9 2
10 3 5 10 7 1 10 10 10 9 5 1 9 2 1 6 0 10 1 0 6

0을 제거한 후 numbers의 원소 수: 92
6 3 10 9 4 7 6 5 4 2 3 3 8 5 2 3 2 7 6 5 7 8 6 3 1 7 1 6 6 2 1 1 7 8 7 1 4 9 4 4
10 5 8 4 1 8 5 2 9 3 6 8 2 8 2 1 6 5 2 10 9 6 6 7 7 3 1 2 1 3 8 9 2 10 3 5 10 7
1 10 10 10 9 5 1 9 2 1 6 10 1 6

```

다시 정리해보면 [문제]를 전역함수로 해결할 수도 있고 멤버함수로 해결할 수도 있습니다. 여기서 당연히 이런 질문을 할 수 있습니다. 내가 질문하지 않더라도 여러분은 당연히 이런 의문이 생길 것입니다.

[문제] numbers에서 0을 제거하라

// 1. 전역 함수 erase-remove idiom으로 제거

```
numbers.erase( remove( numbers.begin( ), numbers.end( ), 0 ), numbers.end( ) );
```

// 2. 멤버로 제거

```
numbers.remove( 0 );
```

[질문] 전역 알고리즘 함수로 원하는 원소를 제거할 수 있는데 왜 멤버함수가 있는 것일까?

잠깐 동안이라도 생각해 보세요. 그럴듯한 설명을 할 수 있습니까? 전역함수와 멤버함수는 어떤 차이가 있는지 설명할 수 있으면 이 문제에 답을 할 수 있습니다.

이미 지난 강의에서 강조해서 설명한 것과 같이 **알고리즘과 컨테이너는 서로 모르는 사이**라는 것이 이 문제의 답입니다. 전역함수인 `remove`는 반복자로 지정된 범위를 인자로 받아 책 95 쪽의 그림 2-4와 같은 방식으로 원소를 제거합니다. `remove`는 제거하려는 원소가 어떤 컨테이너의 원소인지 알 방법이 없습니다. 그런데 `list` 컨테이너의 **멤버함수인 `remove`**는 전혀 다릅니다. **멤버함수이기 때문에 `list`의 특성을 그대로 다 이용할 수 있습니다.** 원소를 지울 때도 그림 2-4와 같이 뒤쪽의 원소들을 연속해서 처리할 이유가 없습니다. 그냥 앞·뒤 노드의 포인터를 재지정하고 메모리를 삭제하면 그만입니다. 그렇게 하려고 `list`를 설계한 것이니 노드 기반 컨테이너의 특성을 이용하면 되는 것입니다. 멤버함수니까요.

STL에서 **같은 기능의 전역함수와 멤버함수가 있다면 두 말할 것도 없이 멤버함수를 선택해야 합니다.** “전역함수보다 더 좋으니 멤버함수를 만들어 놓은 것이다”라고 생각하면 되겠습니다.

다음은 `splice`와 `merge`를 사용해 보겠습니다. 문제를 간단하게 하기 위해 결과를 쉽게 알아볼 수 있는 `int`를 원소로 사용합니다. 뻔한 예제라 그냥 책을 보면 됩니다. 특별한 함수들이라 그냥 넘어가긴 그래서 사용만 해 봅니다. 오버로드된 함수의 일부만 사용하였습니다. 코드와 출력화면이 그냥 설명이니 잠깐 살펴보세요. 그리고 원소를 바꿔 실행해 보세요.

```
// list의 splice와 merge

#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

void print( string name, list<int>& cont );    // list 내용 화면출력

int main( )
{
    list<int> odd { 5, 1, 3, 9, 7 };
    list<int> even { 2, 6, 10, 4, 8 };
    list<int> num;

    cout << "--- 컨테이너의 원소 출력 ---" << endl;
    print( "odd: ", odd );
    print( "even: ", even );
    print( "num: ", num );

    // odd와 even을 num에 merge한다
    num.merge( odd );
    num.merge( even );
}
```

```

    cout << endl << "--- merge 결과 ---" << endl;
    print( "odd: ", odd );
    print( "even: ", even );
    print( "num: ", num );

    // num의 일부 원소를 part에 splice
    list<int> part;
    part.splice( part.begin( ), num, ++++num.begin( ), -----num.end( ) );

    cout << endl << "--- splice 결과 ---" << endl;
    print( "num: ", num );
    print( "part: ", part );
}

void print( string name, list<int>& cont )
{
    cout << name ;
    for ( int n : cont )
        cout << n << ' ';
    cout << endl;
}

```

컨테이너의 원소를 계속 출력할 필요가 있어 따로 함수를 만들었습니다. 그대로 코딩하고 실행하면 다음과 같은 출력화면을 볼 수 있습니다.

```

Microsoft Visual Studio 디버그 콘솔
--- 컨테이너의 원소 출력 ---
odd: 5 1 3 9 7
even: 2 6 10 4 8
num:

--- merge 결과 ---
odd:
even:
num: 2 5 1 3 6 9 7 10 4 8

--- splice 결과 ---
num: 2 5 10 4 8
part: 1 3 6 9 7

```

이 색이 더 잘 보이네요. 진즉에 바꿀걸 그랬습니다. odd와 even을 num에 merge한 출력결과를 관찰해 보세요. 어떻게 이런 순서로 원소들을 합친 것인지 이해하겠습니까? 퍼즐이라고 생각하고 어떻게 이런 결과가 나올까 생각해보세요. 물론 어떻게 한 것인지 책에 나와있습니다. num을 part로 splice한 결과도 함수에서 지시한 대로 정확합니다. 이 함수에서 평소에는 사용할 일이 없는 방법으로 인자를 사용해 봤습니다. 거꾸로는 이런 연산을 할 수 없습니다.

마지막으로 `sort`와 `unique`를 사용해 보겠습니다. 이 두 함수는 따로 사용하는 것보다는 하나로 묶어서 사용되기 때문에 같이 사용해봅니다.

```
// list의 sort와 unique

#include <iostream>
#include <list>
#include <string>
#include "String.h"
using namespace std;

template <class T>
void print( string name, T& cont );           // container 내용 화면출력

int main( )
{
    list<String> words { "1", "3", "5", "7", "9", "2", "4", "6", "8", "10", "3", "3" };

    cout << "--- 컨테이너의 원소 출력 ---" << endl;
    print( "words: ", words );

    // words를 정렬한다
    words.sort( [ ] ( const String& a, const String& b ) {
        return a.getString( ) < b.getString( );
    } );

    cout << endl << "--- sort 결과 ---" << endl;
    print( "words: ", words );

    // 인접한 중복원소를 제거한다
    words.unique( );

    cout << endl << "--- unique 결과 ---" << endl;
    print( "words: ", words );
}

template <class T>
void print( string name, T& cont )
{
    cout << name ;
    for ( auto& n : cont )
        cout << n << ' ';
    cout << endl;
}
```

이전 예에서 사용했던 `print` 함수를 `template`으로 변경하였습니다. `sort` 함수는 언제나 정렬을 위한 비교방법이 필요합니다. 비교하기 때문에 원소 2개를 인자로 받아야 하는 것입니다. `sort` 함수를 `words.sort()`와 같이 인자 없이 사용하려면 약속한 방식으로 `words`의 원소를 비

교할 수 있다는 말입니다. `sort`는 기본으로 < 연산자를 사용하여 순서를 비교합니다. 따라서 지금 `words`안에 들어있는 객체는 `String`이므로 `String`에 < 연산자를 정의하면 `words.sort()` 호출로 비교할 수 있는 것입니다. 그런데 < 연산자를 미리 만들어 놓으면 associative container에서 중요한 이 내용을 잊고 지나갈 수 있기 때문에 그 때 다시 설명하겠습니다. 물론 다른 기준으로 정렬하려면 비교함수를 전달해야 하는 것은 같습니다. 출력화면은 다음과 같습니다.

```

Microsoft Visual Studio 디버그 콘솔
--- 컨테이너의 원소 출력 ---
words: 1 3 5 7 9 2 4 6 8 10 3 3
--- sort 결과 ---
words: 1 10 2 3 3 3 4 5 6 7 8 9
--- unique 결과 ---
words: 1 10 2 3 4 5 6 7 8 9
  
```

`sort` 함수의 결과가 바로 눈에 들어오지 않습니다. 익숙한 순서와 다르기 때문입니다. 이 순서는 사전식 순서(lexicographical order)입니다. 물론 숫자 순서로 정렬할 수도 있습니다. 이건 실습으로 남깁니다.

[실습] `list<String>`에 있는 원소를 숫자 오름순서로 정렬하여 출력하라.

`list`의 멤버함수 `unique`는 인접한 동일 원소를 삭제합니다. `sort`한 결과에 `unique`를 호출하였기 때문에 `words`에는 동일한(== 연산의 결과가 true) 원소가 하나도 없으며 모든 원소가 `unique`합니다. 정렬하지 않은 원소들에 `unique` 함수를 호출하면 그렇게 되지 않습니다.

이제 모든 sequence container를 알아보았습니다. 다음은 associative container를 알아볼 순서인데 먼저 iterator를 공부하고 가는 것이 더 좋습니다. iterator를 공부하는데 더 좋은 방법은 지금 우리가 만든 String을 STL container가 되도록 표준에서 원하는 내용들을 덧붙여 다듬으며 String이 iterator를 제공하도록 하는 것입니다. 이렇게 하는 데 1주 이상이 걸릴 수 있습니다만 지금까지와 같이 공부해도 금방 알기 어려운 핵심 뼈대를 따라 프로그램을 만들어 가며 진행하도록 하겠습니다. 강의내용이 계속 이어지기 때문에 한 번에 몰아서 공부하려면 어려울 겁니다. 계속 시간을 내서 공부하기 바랍니다.

다음 시간은 반복자로 가기 전에 지금까지 배운 sequence container로 만들어 볼 수 있는 프로그램 문제들을 실습하며 학습한 지식을 다지는 시간으로 쓰겠습니다.