STL - 7주 1일

Wulong

7주 강의 2020. 4. 27 ~ 2020. 5. 1	
지난 주 4.20 ~ 4.24	**STL 컨테이너** forward_list list list만의 멤버함수 원소 제거/ merge와 splice / sort와 unique 순차 컨테이너를 사용하는 실습 문제
7주 4.27 ~ 5.1	**STL 반복자** 반복자 range-based for 반복자 요구사항 구현 iterator_traits 제네릭 함수 컨테이너 어댑터
다음 주 5.4 ~ 5.8	**STL 컨테이너** map multimap set multiset

지난 주

안녕하세요? STL 수강생 여러분!
순차컨테이너를 다 공부하고 복습해 볼 수 있는 실습 문제들 재미있게 하고 있나요? 보기로 올려놓은 파일 말고도 어떤 자료라도 마음대로 다룰 수 있다는 것을 알아주세요.

과제는 천천히 살펴보면서 채점을 하겠습니다. 올려놓은 사진들을 거의 보긴 했습니다만 정답이 많지는 않았습니다. 강의자료를 천천히 살펴보고 순서대로 코딩하면서 이해해야 할 수 있는 문제입니다. 내가 만든 String 클래스가 이동문법을 제공함으로써 얻을 수 있는 이점을 잘 느낄 수 있었을 것입니다.

이번 주

컨테이너를 계속 공부하는 것보다 이 시점에서 **반복자의 정체**를 알아보는 것이 공부하기에 더 좋습니다. 우리 책에서는 1장에서 바로 반복자를 소개합니다. 또 한 번 강조하겠습니다. 먼저 책을 보세요. 구글링해서 반복자를 찾아보세요. 그렇지 않으면 아마 이번 강의를 보고 좌절할지도 모릅니다. 물론 어디서도 볼 수 없는 방식으로 설명하는 내 강의를 따라 온다면 그 정도로 어려워하지는 않으리라 나는 확신합니다만 그래도 쉬운 내용이 절대 아닙니다.

반복자 소개는 책 1.4와 1.5를 보세요. 간결하게 잘 정리된 내용을 볼 수 있습니다. 오늘 하려는 강의는 당연히 여기에 나와 있는 내용이 아닙니다. 책에 있는 뻔한 내용을 내가 강의할 이유는 없습니다. 그건 시간 아까운 일입니다. 잘 정리된 책 보다 내가 반복자를 더 잘 소개할 자신도 없습니다.

오늘 강의는 **반복자의 정체**를 밝혀보는 내용입니다. 책 2.7에 있는 내용이며 우리 책에서는 이 절을 "건너뛰어도 좋다"라고 적어 놨네요. 그렇지만 내 강의의 목적은 STL을 떠받치는 핵심 컴포넌트들인 컨테이너, 반복자, 알고리즘을 모두 직접 만들어 보는 것입니다. 따라서 이 강의를 꼭 이해하길 바라면서 이 문서를 작성합니다. 앞의 String부터 모든 내용이 이어져 나아가니 놓치지 말고 지난 강의를 참조하면서 따라 오기 바랍니다.

반복자는 컨테이너가 자신의 원소들을 순회할 수 있도록 외부에 제공하는 인터페이스입니다. 그러므로 String 클래스가 반복자를 제공하도록 코딩하겠습니다. String 객체에 .을 찍으면 vector처럼 begin(), end()와 같은 멤버가 나오도록 말이죠.

반복자의 종류를 확인하는 코드에서 시작하여 range-based for의 작동 방식을 밝혀보겠습니다. 이렇게 하는 과정에서 반복자가 무엇인지 단계별로 알아가게 될 것입니다.

String이 제공하는 반복자가 진짜 표준 반복자가 되도록 코딩하겠습니다. 그렇게 할 수 있다면 String은 모든 STL 알고리즘 함수를 이용할 수 있을 것입니다. 반복자가 String과 알고리즘 함수를 이어줄 것이니까요! 그럼 반복자 시작합니다. Go! Go!

iterator

책 31~41, 123~136

반복자는 C++ 프로그램이 서로 다른 자료구조(컨테이너와 범위(range))를 일관된 방식으로 다룰 수 있도록 **포인터를 일반화**한 것입니다. 템플릿으로 작성된 알고리즘 함수가 서로 다른 자료형에 대하여 의도한 대로 정확하면서도 효율적으로 작동할 수 있도록, 표준 라이브러리는 반복자가 가져야 할 인터페이스뿐만 아니라 반복자의 의미론 그리고 반복자가 지켜야할 복잡도도 규정하고 있습니다.

반복자 중에서 가장 기본적인 일만 할 수 있는 입력반복자 i에는 *i라는 표현식을 쓸 수 있습니다. 그러면 반복자가 가리키는 자료형의 값을 얻을 수 있습니다. 이때 *i를 반복자의 value type이라고 합니다. 모든 반복자는 difference type이라는 부호가 있는, 정수와 유사한 자료형을 제공합니다. 이와 같은 기본적인 내용은 책 1.4를 참고하세요.

반복자는 포인터를 추상화한 것이기 때문에 반복자는 C++ 포인터의 의미론을 일반화한 것이라 생각할 수 있습니다. 그래서 반복자를 인자로 받는 모든 함수 템플릿은 포인터를 인자로 받아도 잘 동작하게 되는 것입니다.

반복자는 모두 **6 종류(category)**가 있습니다. 네! 손 든 학생. 말씀하세요.

"지난 시간에 반복자는 5종류가 있다고 했잖습니까? 책에도 5종류라고 나와 있던데요?"

그렇습니다. 우리 책 33쪽 1.4.2 반복자 카테고리에 분명 5종류라고 나와 있습니다. 그런데 우리 책은 C++14 표준 기준입니다. C++17 이후엔 반복자를 다음과 같이 6 종류로 나눕니다.

input -> forward -> bidirectional -> random access -> **contiguous iterators,**
output

이 종류는 또한 순서를 나타내는데요. 화살표 방향으로 갈수록 더 많은 기능을 제공합니다. 그러니 contiguous 반복자가 가장 많은 기능을 하는 반복자가 되겠습니다. 실제 반복자의 종류는 다음과 같은 클래스 상속 구조로 표현합니다. 순전히 상속관계만을 나타내기 위하여 사용된 클래스이므로 클래스 정의에는 아무것도 없는 텅 빈 클래스입니다.

```
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };
struct contiguous_iterator_tag: public random_access_iterator_tag { };
```

반복자가 어떤 종류의 반복자인지 화면에 출력하는 프로그램입니다. 내가 사용하는 비주얼 스튜디오 2019에서는 다음 실행화면과 같은 결과가 출력됩니다. C++에서 contiguous 메모리를 사용하는 컨테이너에는 array, vector 그리고 string이 있습니다. 이들 컨테이너가 제공하는 반복자는 contiguous 반복자입니다만 아직 표준 내용대로 종류가 나뉘어 있지는 않습니다. 사실 contiguous 반복자가 random 반복자에 비해 더 할 수 있는 일은 거의 없습니다. 아직 최신 표준이 구현된 것은 없지만 contiguous 메모리의 특성을 살린다면 random 반복자로는 할 수 없는 memcpy를 알고리즘 함수에서 사용하게 될 것이라 예상할 수 있습니다. deque은 랜덤반복자이지만 memcpy 함수를 사용할 수 없습니다.

```
// 코드 7-1 : 반복자의 종류 출력

#include <iostream>
#include <forward_list>
#include <list>
#include <deque>
#include <array>
#include <vector>
#include <iterator>
#include "String.h"
using namespace std;

template <class T>
void showCategory(  const T& );

int main( )
{
    istream_iterator<char> in_iter;
    ostream_iterator<short> out_iter { cout };
    forward_list<int>::iterator forward_list_iter;
    list<float>::iterator list_iter;
    deque<double>::iterator deque_iter;
    array<String, 0>::iterator array_iter;
    vector< vector<char> >::iterator vector_iter;

    showCategory( in_iter );
    showCategory( out_iter );
    showCategory( forward_list_iter );
    showCategory( list_iter );
    showCategory( deque_iter );
    showCategory( array_iter );
    showCategory( vector_iter );
}

template <class T>
void showCategory( const T& t )
{
    cout << typeid(iterator_traits<T>::iterator_category).name( ) << endl;
}
```

[실습] 포인터는 어떤 타입의 반복자인가 출력하라. (5분)

range-based for

모든 STL 컨테이너는 모든 원소를 처음부터 마지막 원소까지 순회하며 access하는 경우에는 range-based for를 사용할 수 있습니다. STL 컨테이너는 아니지만 배열(array가 아님)에도 range-base for를 사용할 수 있는데 이것은 컴파일 타임에 배열의 자료형과 크기가 확정되기 때문에 가능한 것입니다.

한편, 내가 만든 클래스 String도 char라는 동일 타입의 원소를 저장하므로 개념상으로는 STL 컨테이너와 같다고 생각할 수 있습니다. 거꾸로 접근해 보겠습니다. 일단 range-based for를 시도해보고 잘못된 것을 고치면서 String에 필요한 것이 무엇인지 알아보겠습니다.

```
// 코드 7-2 : range-based for를 사용하여 String을 순회

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

int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    for ( char c : str )
            cout << c << ' ';
    cout << endl;
}
```

비주얼 스튜디오에서는 str에 빨강 줄이 쳐지고 마우스를 가져가면 begin 멤버가 없다고 나옵니다. 사실 범위기반의 for 루프는 컴파일 과정에서 다음과 유사한 코드로 바뀝니다.

```
auto beg = str.begin( );
auto end = str.end( );
char c;
for ( ; beg != end; ++beg ) {
        c = *beg;
        cout << c << ' '
}
cout << endl;
```

for 루프에서 값을 꺼낼 때 char c; 대신 auto c;라고 썼다면 auto c = *beg;과 같이 초기화할 수 있을 것입니다. 원래는 위와 같이 작성해야할 코드를 range-based for를 사용하여 간단하게 쓸 수 있는 것입니다. 복잡한 사용법을 간단하게 만든 것들이 프로그래밍 언어들에 많이 있는데 이들을 통칭 syntactic sugar(꿀 문법, 아직 번역어는 없으니 원어대로 알아두세요)라고 합니다.

그럼 class String이 무엇을 제공하면 range-based for가 실행되도록 할 수 있겠습니까? 바로 begin(), end() 멤버를 제공하면 되겠습니다. String은 p에 len개의 char를 저장합니다. 그러니 String.h와 String.cpp에 begin()과 end()를 다음과 같이 추가합니다.

```
// String.h에 추가

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

    // 반복자를 위한 멤버
    // 2020. 4. 25 추가
    char* begin( );
    char* end( );
```

```
// String.cpp에 추가

// 반복자를 위한 멤버
// 2020. 4. 25 추가
char* String::begin( )
{
    return p;
}

char* String::end( )
{
    return p + len;
}
```

이제 range-based for를 사용하여 String str의 원소를 하나씩 출력해 봅니다. 문제없이 실행됩니다. 그런데 좋아하기만 할 일은 아닙니다. 왜냐하면 String의 반복자는 그냥 포인터이기 때문입니다. 사실 포인터는 랜덤반복자입니다. 또한 포인터 앞에 *연산자를 붙이면 값을 가져올 수 있다는 것도 이미 정의되어 있습니다. 할 일이 별로 없었을 뿐입니다.

begin(), end() 멤버를 만들었으니 String의 정렬도 폼나게 이렇게 할 수 있습니다.

```
// 코드 7-3 : String의 멤버를 sort로 전달

int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    sort( str.begin( ), str.end( ) );

    cout << str << endl;
}
```

결과를 살펴보니 이 문장은 **pangram**임을 알 수 있습니다. 그것도 아주 유명한 문장이네요.

반복자 요구사항 구현

그런데 지금까지는 너무 쉽지 않았습니까? String::begin()이 알려주는 것은 저장한 문자들의 시작번지를 가리키는 포인터일 뿐인데 아무런 문제가 없었으니까요, String과 비슷한 컨테이너들도 이렇게 할까요? 예를 들어 vector와 같은 진짜 컨테이너를 살펴보면 분명히 vector는 contiguous 컨테이너이니까 반복자를 리턴하는 멤버함수 begin()이 T*를 리턴하면 될 텐데 실제로는 iterator라는 클래스 타입을 리턴한단 말입니다.

반복자는 포인터를 추상화한 것이지 포인터는 아닙니다. 컨테이너 내부에서 그냥 포인터를 사용하여 반복자를 구현하였더라도 포인터로는 구현할 수 없는, 진짜 반복자들과 동일한 인터페이스를 제공하기 위한 작업이 더 필요합니다. 표준 반복자가 되려면 **STL 반복자 요구사항**을 지켜야 합니다. (책 2.7을 참조)

String의 begin()에서 char*를 리턴했는데 sort 알고리즘 함수에서 아무 문제가 없는 것은 포인터를 반복자로 사용할 수 있도록, 포인터에 대하여 반복자 클래스를 특수화하였기 때문입니다. 우리도 **String 클래스가 자기만의 반복자를 제공하도록 프로그램**한다면 반복자 내부에서 벌어지는 일들을 더 깊게 이해할 수 있으며 STL을 더 자유롭게 사용하는데 많은 도움이 될 것입니다.

String의 begin()을 호출하면 String::iterator를 리턴하도록 코딩해 보겠습니다. 순서대로 잘 따라 오기 바랍니다.

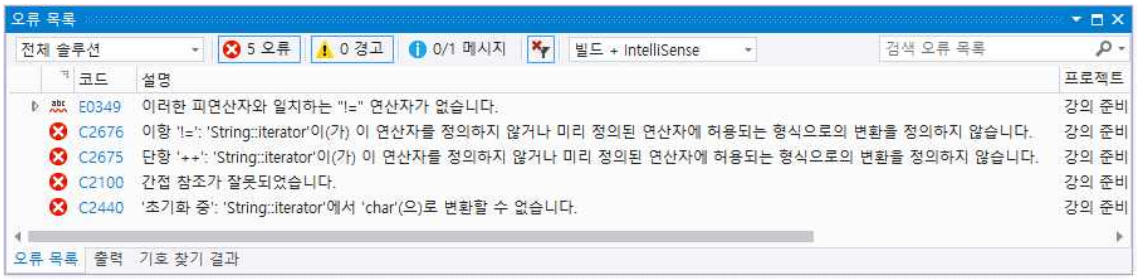
```
// String.h의 class String 앞에 class String_Iterator 추가

#pragma once

class String_Iterator {
    char* p { nullptr };

public:
    String_Iterator( char* p ) : p { p }  { };
};

class String {

// 생략
```

```
// String.h에 함수 선언 변경

    // 반복자를 위한 멤버
    // 2020. 4. 25 추가
    using iterator = String_Iterator;

    iterator begin( );
    iterator end( );
```

```
// String.cpp에 함수 정의 변경

// 반복자를 위한 멤버
// 2020. 4. 25 추가

String::iterator String::begin( )
{
    return iterator( p );
}
```

```
String::iterator String::end( )
{
    return iterator(p + len);
}
```

클래스 String이 iterator를 리턴할 수 있게 바꿨으니 String을 정렬하는 코드 7-3을 다시 돌려봅니다. 이런! 오류가 나서 컴파일 실패합니다. sort 함수는 뭔가 복잡한 거 같습니다. 더 앞으로 돌아가서 range-base for를 구현했던 코드 7-2를 실행해 봅니다.



또 한 번 이런! 역시 오류가 나며 실패합니다. 여기서 "안 되는구나!"하고 멈추면 공부라 할 것도 없겠죠. 천천히 오류메시지를 살펴봅니다.

!= 연산자가 없다
++ 연산자가 없다
간접 참조가 잘못되었다. (* 연산자가 없다는 말입니다)

마지막의 "초기화 중~" 오류는 간접 참조가 해결되면 해결됩니다. 지금 하려는 건 내가 만든 반복자 String_Iterator가 내가 만든 컨테이너인 String의 원소들을 순회하게 하는 것입니다. 그렇게 하려면 위와 같은 연산이 반드시 제공되어야 하겠습니다. 이 연산들은 **forward iterator의 요구사항**입니다. 천천히 생각해 보는 겁니다. 시작위치 beg에서 ++로 한 걸음씩 전진합니다. 언제까지? 시작위치가 마지막 위치인 end가 아닌 동안입니다. 즉, beg != end인 동안 *beg으로 원소의 값을 억세스하면서 말이죠.

String_Iterator가 C++ 표준 forward iterator가 되도록 기능을 추가하겠습니다.

```
// String.h에 추가

class String_Iterator {
    char* p { nullptr };

public:
    String_Iterator( char* p ) : p { p }  { };

    bool operator!=( const String_Iterator& rhs ) const {
            return p != rhs.p;
```

```
    }

    String_Iterator& operator++( ) {
            ++p;
            return *this;
    }

    char operator*( ) {
            return *p;
    }
};
```

다시 코드 7-2를 실행시켜 보세요. 문제없이 실행되죠?

지금 우리는 String을 STL 컨테이너라 생각하고 표준 STL 컨테이너가 그렇게 하는 것처럼 String이 iterator를 제공하도록 하였습니다. 그런데 진짜 STL 컨테이너라면 iterator 이외에 다음과 같은 반복자도 제공하여야 합니다.

const_iterator
reverse_itertor
const_reverse_iterator

읽기 전용 반복자인 const_iterator는 필요하면 금방 만들 수 있을 것입니다. 그런데 컨테이너의 원소를 반대방향으로 순회할 수 있는 **역방향 반복자(reverse iterator)**는 **반복자 어댑터**입니다(책 37쪽 참고). 컨테이너의 원소를 거꾸로는 순회할 수 없는 forward_list는 역방향 반복자를 제공할 수 없습니다.

클래스 String은 원소들을 contiguous 메모리에 저장합니다. 따라서 역방향으로 원소를 순회할 수 있는 것은 물론 랜덤 억세스도 가능합니다. 클래스 String이 reverse_iterator를 제공하도록 필요한 클래스를 만들고 String도 수정하겠습니다. 이 작업이 성공한다면 다음 프로그램이 문제없이 컴파일되고 실행되어야 합니다.

```
// 코드 7-4 : 역방향 반복자로 String을 순회

int main( )
{
    String str { "The quick brown fox jumps over the lazy dog" };

    for ( auto i = str.rbegin( ); i != str.rend( ); ++i )
            cout << *i;
    cout << endl;
}
```

코드 7-4를 입력해보면 무엇을 해야 하는지 명확합니다. String.h에 역방향 반복자 class를 새로 만들고 필요한 기능을 추가합니다. 클래스 이름을 String_Reverse_Iterator로 해서 만들어 보겠습니다.

```
// String.h의 class String_Iterator 밑에 코딩하라

class String_Reverse_Iterator {
    char* p { nullptr };

public:
    String_Reverse_Iterator( char* p ) : p { p } { };

    bool operator!=( const String_Reverse_Iterator& rhs ) const {
            return p != rhs.p;
    }

    String_Reverse_Iterator& operator++( ) {
            --p;
            return *this;
    }

    char operator*( ) {
            return *(p-1);
    }
};
```

역방향 반복자는 독특하게 동작합니다. 빨간색으로 표시한 곳을 보세요. 역방향 반복자에게 앞으로 가라고 ++연산을 하면 포인터를 뒤로 이동합니다. 또 역방향 반복자가 가리키고 있는 원소의 값을 달라고 *연산을 하면 실제 가리키고 있는 원소의 이전 원소의 값을 리턴합니다. 이건 내가 내 마음대로 만든 것이 아니고 이렇게 하기로 약속한 것입니다. 실제 역방향 반복자에는 몇 가지 관련함수가 더 있습니다만 우리 강의에서는 이걸로 충분합니다.

클래스 String이 역방향 반복자를 지원할 수 있도록 멤버를 추가합니다.

```
// String.h에 추가

    // 반복자를 위한 멤버
    // 2020. 4. 25 추가
    using iterator = String_Iterator;
    using reverse_iterator = String_Reverse_Iterator;

    iterator begin( );
    iterator end( );
```

```
    reverse_iterator rbegin( );
    reverse_iterator rend( );
```

```
// String.cpp에 추가

// 반복자를 위한 멤버
// 2020. 4. 25 추가

String::iterator String::begin( )
{
    return iterator( p );
}

String::iterator String::end( )
{
    return iterator(p + len);
}

String::reverse_iterator String::rbegin( )
{
    return reverse_iterator( p + len );
}

String::reverse_iterator String::rend( )
{
    return reverse_iterator( p );
}
```

클래스 String의 begin()과 rend(), end()와 rbegin()이 같은 위치를 가리키고 있도록 코딩한 것을 살펴보세요. 자. 그럼 준비가 다 되었으니 역방향 반복자로 String을 순회하는 코드 7-4를 실행시켜 봅니다.

짝짝짝!　　멋지게 성공했습니다.

어떻습니까? 따라 올 만 했나요? 그렇게 어렵지는 않았죠?
반복자와 컨테이너가 어떻게 엮여 돌아가는지 알 것 같은 기분이죠?

아쉬운 점은 반복자를 String.h에 그냥 코딩한 것입니다. 반복자를 다른 파일로 분리하는 것이 맞습니다만 그렇게 하면 문서로 설명하기가 너무 복잡해서 그냥 이렇게 한 것입니다. 반복자가 무엇인지 이해하는 것이 목적이니 넘어갑니다.

iterator_traits

제일 앞의 반복자의 종류를 출력하는 코드 7-1로 돌아갑니다. 지금까지 우리가 만든 String의 iterator와 reverse_iterator는 어떤 종류의 반복자입니까? 아니 어떤 종류의 반복자가 되어야 합니까? String의 메모리는 contiguous 메모리이니 String의 반복자는 contiguous_itertor가 되어야 합니다. 아니죠. 되어야 하는 것이 아닙니다. String을 만든 것은 바로 나니까 내가 그렇게 지정하면 되는 것입니다. 그런데 지금 비주얼 C++에서는 random_access_iterator가 최고 성능의 반복자입니다. 그러니 String의 반복자가 이렇게 출력되도록 하면 되겠습니다. 일단 코드를 작성해 봅니다.

```
// 코드 7-5 : String의 반복자 종류 출력

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

int main( )
{
    cout << typeid(iterator_traits<String::iterator>::iterator_category).name( ) << endl;
}
```

컴파일되지 않는 이 코드를 고쳐야 하겠습니다. 빨간색 코드 조각에 비밀이 있습니다. 지금 우리가 하려고 하는 일은 반복자로부터 여분의 정보를 얻어내는 일입니다. 포인터로 돌아가서 물어봅니다. 다음과 같은 코드로요.

```
int main( )
{
    cout << typeid(iterator_traits<String*>::iterator_category).name( ) << endl;}
}
```

이 코드는 String*가 어떤 종류의 반복자인지 묻고 있습니다. 포인터(*)는 대표적인 POD 자료형인데 어디에서 여분의 정보가 있기에 "나는 랜덤반복자요"라고 대답할 수 있는 것일까요? 코드를 조금 더 살펴보면 iterator_traits(반복자의 속성)라는 템플릿 클래스가 보일 것 입니다. iterator_traits에 String*라는 타입을 인자로 넘겨 템플릿 class를 만들면 이 클래스의 멤버인 iterator_category라는 타입을 알 수 있다는 것입니다. 이 설명은 조금 어려웠을 텐데 이것이 정확한 표현이기 때문에 적었습니다.

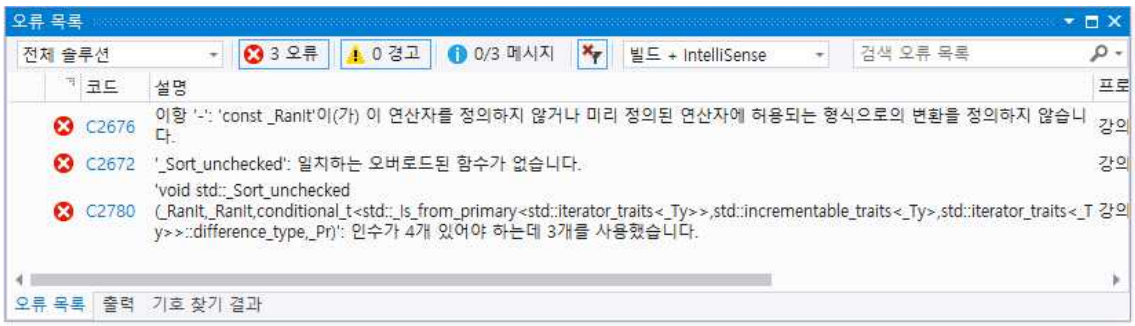
조금 쉽게 설명해 볼까요? 클래스 String도 iterator_traits 클래스에 "나도 반복자이고 나는 랜덤반복자입니다"라고 자신의 정보를 등록하면 됩니다. 표준 반복자는 등록해야할 항목이 다섯 가지가 있습니다. 이 다섯 가지 항목을 등록하는 것은 내가 만든 반복자 클래스를 만들 때 iterator_traits로 부터 상속하여 만드는 방법이 있고 또 다른 방법은 곧 바로 템플릿을 특수화하는 방법이 있습니다. 코드를 보는 것이 쉽습니다.

```
// String.h에 추가한다

class String_Iterator {
    // 생략
};

// 2020. 4. 26
template <>
struct std::iterator_traits<String_Iterator> {
    using iterator_category = random_access_iterator_tag;
    using value_type = char;
    using difference_type = ptrdiff_t;
    using pointer = char*;
    using reference = char&;
};
```

직접 등록하는 방법입니다. 설명이 필요 없는 코드이니 한 번 보세요. 그럼 코드 7-5를 다시 실행시켜 봅니다.

이제 클래스 String의 반복자는 랜덤반복자가 되었습니다. 위의 다섯 가지 속성은 알고리즘 함수에서 필요하다면 사용합니다. 알고리즘 함수 sort는 랜덤반복자에서만 동작하는 함수입니다. 랜덤반복자가 아니라면 컴파일시에 많은 오류가 날 것입니다. C++20에서는 함수의 조건을 제약하고 더 정교하게 따져볼 수 있는 concept과 require가 도입됩니다.

그럼. String의 반복자도 랜덤반복자로 만들었으니 String을 정렬하는 코드 7-3을 다시 실행해 봐도 좋겠습니다.



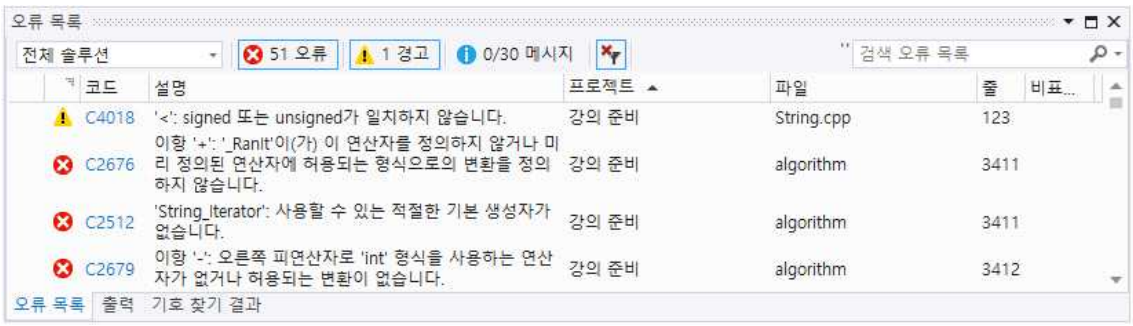
그럼. 그렇지. 안 됩니다. 한 번에 딱 되는 건 프로그램이 아닙니다. 그럴 리가 없습니다. 지금 String의 반복자는 랜덤반복자라고 이름만 붙였지 실제 랜덤반복자가 요구하는 기능을 구현한 것이 없습니다. 전진반복자의 기능만 구현했을 뿐입니다. 양방향 반복자의 기능도 못 만

들었죠. 그런데 랜덤반복자가 요구하는 기능이 있을 리 없습니다.

랜덤 반복자가 되려면 반복자에 사칙연산이 가능해야 합니다. 랜덤이 그런 의미니까요. 어느 메모리 위치로든 상수시간에 도달할 수 있는 것이 랜덤반복자입니다. 이걸 다 구현할 것까진 없습니다. 지금 오류 목록을 보면 이항 - 연산자가 없다고 하네요. 이거 하나 구현해 볼까요? 그리고 다시 컴파일하면서 필요한 코드를 추가하며 모든 오류를 없애는 작업을 반복하면 sort를 실행할 수 있습니다. 반복자니까 반복해야죠.

```
// String.h에서 String_Iterator에 연산자 추가

class String_Iterator {

    생략

    char operator*( ) {
            return *p;
    }

    ptrdiff_t operator-( const String_Iterator rhs ) const {
            return p - rhs.p;
    }
};
```

다시 컴파일합니다.



오류가 51개밖에 안됩니다. 다음에 해결할 것은 <, + 연산자 순입니다. 실제로는 10개 밑으로 만들면 될 겁니다. 이 것은 실제 강의할 때 학생들에게 해결해 보라고 하는 문제입니다. 연산자 오버로딩도 복습할 겸 끝까지 해 보세요. 그렇게 어렵지 않습니다.

[실습] 코드 7-3이 문제없이 실행되도록 String_Iterator를 코딩하라. (30분+)

오늘 강의는 여기까지 입니다. **[실습]** 문제 마칠 때까지 강의실에서 못 나갑니다. ^^.
다음 강의내용은 많지 않을 겁니다. 오늘 평소보다 너무 나갔네요.