

STL - 4주 2일

Wulong

지난 강의에서 array를 소개하며 STL 컨테이너를 알아보기 시작하였습니다. array는 스마트 배열이라고 생각하면 됩니다. array는 자료를 저장하기 위한 메모리를 동적할당 하지 않습니다. 또한 메모리는 물리적으로 연속(contiguous)되어 있습니다. 따라서 클래스로 만든 array는 배열이 가진 장점에 더하여 배열에서 할 수 없었던 기능들을 멤버로 제공하니 스마트한 배열인 것입니다. 생각해 보는 겁니다. 내가 다루어야 할 자료의 갯수가 컴파일 타임에 정해져 있으며 실행되는 동안 갯수가 변동될 일이 전혀 없는 경우라면 이 보다 더 좋은 자료구조는 없습니다. 속도와 메모리면 둘 다에서 말입니다.

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

int main()
{
    array<String, 5> words { "corona", "virus", "world", "crisis", "pandemic" };

    for ( int i = 0; i < words.size(); ++i )
        cout << words[i] << endl;
}
```

메모리가 연속되어 있다는 것은 각 원소에 접근하는 시간이 $O(1)$ 이라는 점에서 다른 자료구조와 구별되는 큰 장점입니다. STL 알고리즘 함수인 sort는 메모리가 붙어있는 자료구조가 아니라면 정렬해 주지도 않습니다. 알고리즘 함수와 컨테이너는 서로 독립되어 있다고 했습니다. 어떻게 알고리즘 함수 sort는 컨테이너의 메모리가 연속되어 있다는 것을 알 수 있을까요? 알고리즘 함수와 반복자를 연결해 주는 반복자에 그 해답이 있습니다. STL에서는 반복자에게 물어볼 수 있거든요. 이렇게 말이죠.

- 반복자야. 너는 어떤 종류의 반복자니?
- 혹시 네가 가리키고 있는 원소는 메모리가 붙어있는 자료구조에 들어있니?

이미 알고리즘 함수를 사용해 보기는 했지만 array<String, 5> words를 사전식으로 비교하여 오름차순으로 정렬해 보고 잠깐 알고리즘과 자료구조의 관계를 설명하겠습니다. 아래 코드는 array에 저장된 자료들을 정렬한 후 출력하는 코드입니다.

```
#include <iostream>
#include <array>
#include <algorithm>
```

```

#include "String.h"
using namespace std;

int main( )
{
    array<String, 5> words { "corona", "virus", "world", "crisis", "pandemic" };

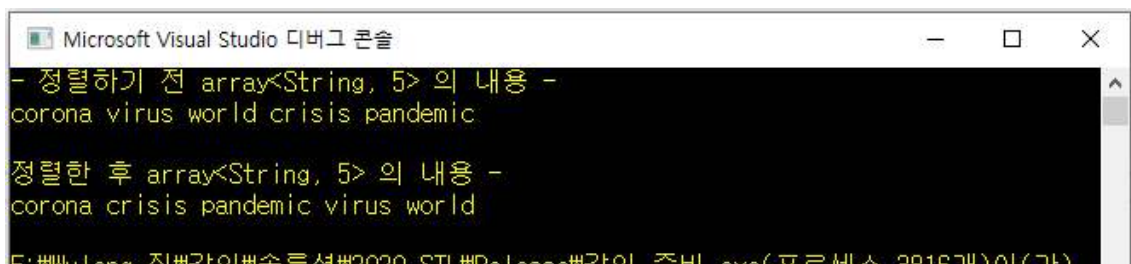
    cout << "- 정렬하기 전 array<String, 5> 의 내용 - " << endl;
    for ( int i = 0; i < words.size( ); ++i )
        cout << words[i] << " ";
    cout << endl;
    cout << endl;

    sort( words.begin( ), words.end( ), []( const String& a, const String& b ) {
        return a.getString( ) < b.getString( );
    } );

    cout << "정렬한 후 array<String, 5> 의 내용 - " << endl;
    for ( int i = 0; i < words.size( ); ++i )
        cout << words[i] << " ";
    cout << endl;
}

```

실행해 보세요. 관찰메시지 덕분에 객체가 어떻게 생성되고 이동하고 소멸하는지 볼 수 있습니다. 복잡한 출력 메시지는 한 번 잘 살펴볼 가치가 있습니다. 다음은 관찰메시지가 없는 실행화면입니다. 어떻습니까? array<String, 5>에 저장된 String 객체들이 사전 순서대로 위치를 바꿨나요?



정렬 함수 sort를 알아보겠습니다. VS2019의 sort는 4개의 오버로딩된 함수들이 있습니다만 기본은 다음과 같습니다.

- sort(어디에서, 어디까지, 어떻게 비교하지?);

어디에서 어디까지는 처리해야 할 원소들의 범위입니다. 기호로 나타내보면 [어디에서, 어디까지)와 같이 씁니다. 구간 기호 [과) 은 오타가 아닙니다. STL의 약속에 따라 시작은 원소를 포함하고 마지막은 실제 원소가 아닙니다. 위의 프로그램에서는 모든 STL 컨테이너가 제공해야만 하는 표준 멤버 함수인 begin()과 end()를 사용하여 처리해야 할 구간을 표시하였습니다.

다. 조금 더 익숙했던 표기인 주소로 다시 표현하면

```
sort( &words[0], &words[5], []( const String& a, const String& b ) {
    return a.getString( ) < b.getString( );
} );
```

이렇게 씁니다. words[0]은 포함되고 words[5]는 가리킬 수는 있지만 실제 원소는 없습니다. 이건 약속입니다. 메모리가 연속된 컨테이너에서 처리해야 할 자료의 범위가 [beg, end)라면 전체 원소의 갯수는 end - beg 입니다. beg == end라면 빈 컨테이너입니다.

sort 함수에게 처리해야 할 범위를 알려주면 sort 함수는 처리해야 할 자료가 < 연산자를 정의하고 있는 지 살펴봅니다. < 연산자가 있다면 default로 이 연산자를 사용하여 두 원소를 비교합니다. 내가 만든 String class는 < 연산자가 없습니다. 그렇다면 sort 함수에게 어떻게 두 원소를 비교해야 할 지 알려줘야 합니다. 할 일을 어떻게 알려주는 거였죠? 그렇습니다. 세 번째 인자에는 호출가능 타입이 들어가는 겁니다.

sort 함수는 범위 안에 있는 원소들 중 2개를 집어내어 호출가능 타입에 넘깁니다. 어떻게 정렬할 지 알려달라고 말이죠. 그래서 위 프로그램에서 사용한 호출가능 타입인 람다에 전달되는 인자가 String이 되는 것입니다. 위 코드에서는 전달된 String 객체를 표준 객체인 std::string으로 변환하여 사전식 오름차순으로 순서를 정하고 있습니다. 비교 결과는 항상 bool 이어야 합니다. 비교 결과에 따라 sort는 두 원소의 내용을 서로 바꿉니다. 어떻게 정렬 순서를 정할지는 프로그래머 마음입니다. 내림차순으로 정렬하고 싶다면 이렇게 하면 됩니다.

```
sort( &words[0], &words[5], []( const String& a, const String& b ) {
    return a.getString( ) > b.getString( );
} );
```

기호 모양대로 생각해 보세요. < 올라가고 > 내려갑니다.

정렬할 때 어떤 방식으로 원소 2개를 골라 호출가능 타입에 전달할 것인지는 정렬 알고리즘에 따라 다릅니다. 호출가능 타입을 더 적은 횟수로 호출할 수 있다면 더 빠른 정렬알고리즘이 되겠습니다. STL은 sort 함수가 어떤 알고리즘이어야 한다고 정의하지 않습니다. 단지 표준에는 STL 정렬함수 “sort는 $O(n \log n)$ 복잡도를 가져야 한다”라고 적혀 있을 뿐입니다. 메모리를 생각하며 천천히 관찰하세요.

[실습] 출력결과와 같이 나오도록 array를 정렬하라. (10분)

```
int main( )
```

```

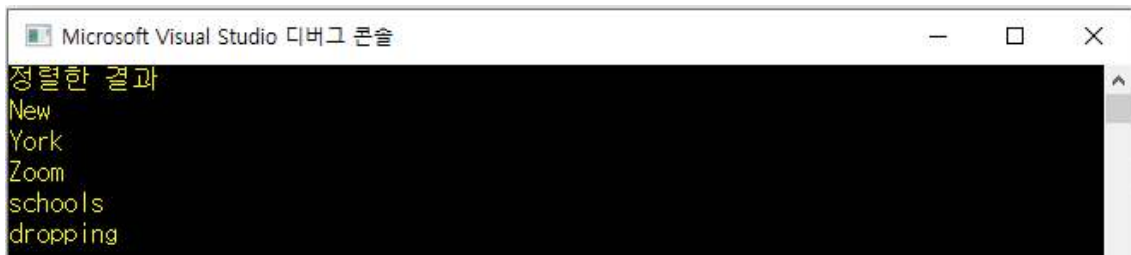
{
    array<String, 5> words { "New", "York", "schools", "dropping", "Zoom" };

    // 여기에 들어갈 정렬 코드를 적어라.

    cout << "정렬한 결과" << endl;
    for ( int i = 0; i < words.size( ); ++i )
        cout << words[i] << endl;
}

```

이렇게 출력되도록 정렬해 보세요.



어떤 기준으로 정렬한 것인지 알겠지요? String의 길이 즉, 글자 수 기준 오름차순 정렬입니다. 하나 더 해 보죠.

[실습] 출력결과와 같이 나오도록 array를 정렬하라. (20분)

```

int main( )
{
    array<String, 16> words { "Anger", "as", "Japanese", "Prime", "Minister offers", "two",
                             "cloth", "masks", "per", "family", "while", "refusing", "to", "declare",
                             "coronavirus", "emergency" };

    cout << "정렬하기 전" << endl;
    for ( int i = 0; i < words.size( ); ++i )
        cout << words[i] << " ";
    cout << endl << endl;

    // 여기에 들어갈 정렬 코드를 적어라.

    cout << "정렬한 결과" << endl;
    for ( int i = 0; i < words.size( ); ++i )
        cout << words[i] << " ";
    cout << endl;
}

```

이렇게 출력되도록 정렬해 보세요. String 클래스를 변경할 필요는 전혀 없습니다. 메모리를

정확하게 이해할 수 있다면 간단한 코드로 문제를 해결할 수 있습니다. 코드는 간단할테지만 메모리를 모른다면 답을 할 수 없을 것입니다.

```

Microsoft Visual Studio 디버그 콘솔
정렬하기 전
Anger as Japanese Prime Minister offers two cloth masks per family while refusing
g to declare coronavirus emergency
정렬한 결과
Aegnr as Jaaeenps Peimr Meeffiinorrst otw chlot akms epr afilmy ehilw efginrs
u ot acdeelr acinoorrsuv ceeegmry
  
```

vector

책 79 ~ 99쪽

중요한 컨테이너라 글자 크기를 키우고 색도 바꿔 봤습니다. 책 읽어 주세요. STL 컨테이너 중에서 가장 많이 사용되는 컨테이너입니다. 가장 많이 사용된다고 해서 제일 좋다거나 매우 효율적인 컨테이너라고는 말할 수는 없습니다. STL 표준에 컨테이너가 여러 개 있는 것은 다 이유가 있는 것입니다.

vector는 동적배열입니다. 동적이란 무슨 뜻이라고요? 네. **프로그램이 실행될 때** 무엇인가가 변화한다는 의미입니다. 배열에서 변할 수 있는 것은 배열의 원소 갯수입니다. vector는 실행 시 원소의 갯수가 변할 수 있습니다. 모든 것이 다 그렇듯이 다 편리한 만큼 댓가가 있습니다. 메모리를 이해하지 못하고 vector를 사용하면 생각하는 것보다 느리다고 불평하며 사용하게 될 것입니다.



vector는 반드시 물리적으로 연속된 메모리를 사용하여 원소를 저장합니다. 오른쪽에 화살표를 그려 놓은 것은 확보한 메모리를 다 사용하였을 때 원소의 갯수를 늘릴 수 있다는 것을 뜻합니다. 그림을 이렇게 그려 놓으면 오해하기 쉬운 점이 메모리가 그냥 오른쪽으로 조금씩 늘어난다고 생각한다는 것입니다. 그렇게 될 수는 없습니다. 메모리는 new로 요청해서 확보하는 것입니다. 그림에서 벡터에 원소 6개를 담을 수 있습니다. B 다음에 하나를 더 담고 싶으면 방법은 이렇게 하는 것 말고 없습니다.

- 현재 담을 수 있는 원소의 갯수(capacity)보다 더 큰 새 메모리를 요청한다.
- 새 메모리에 현재 원소를 모두 복사한다.
- 복사한 원소들의 다음 위치에 새 원소를 추가(push_back)한다.
- 이전 메모리를 반환한다.

메모리를 새로 할당받고 거기로 옮겨 가는 것입니다. 얼마나 비용이 큰 동작인지 알고 사용해야 합니다. 이전 강의에서도 이야기 했지만 **new, delete 명령을 사용하지 않을 수 있다면 그렇게 하는 것이 최상**이라는 점을 꼭 기억하기 바랍니다. 특히 게임 루프에서 이런 코드를 사용하는 것은 최악입니다.

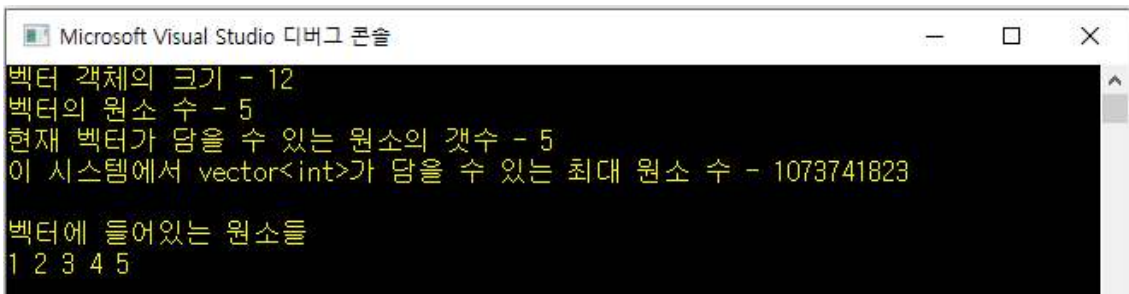
벡터의 이름이 왜 벡터가 되었는지 찾아보는 것도 재미있겠습니다. (Why a C++ Vector called a vector?). 물리와 수학에서의 벡터와 같은 개념입니다. 벡터 자체가 어떻게 동작하는지 살펴보는 것이 내 강의의 핵심입니다만 이번 시간에는 먼저 간단한 프로그램으로 벡터의 동작을 관찰하는데 까지만 살펴보겠습니다.

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

int main( )
{
    vector<int> v { 1, 2, 3, 4, 5 };

    cout << "벡터 객체의 크기 - " << sizeof( v ) << endl;
    cout << "벡터의 원소 수 - " << v.size( ) << endl;
    cout << "현재 벡터가 담을 수 있는 원소의 갯수 - " << v.capacity( ) << endl;
    cout << "이 시스템에서 vector<int>가 담을 수 있는 최대 원소 수 - "
        << v.max_size( ) << endl;

    cout << endl;
    cout << "벡터에 들어있는 원소들" << endl;
    for ( int i = 0; i < v.size( ); ++i )
        cout << v[i] << " ";
    cout << endl;
}
```



```
Microsoft Visual Studio 디버그 콘솔
벡터 객체의 크기 - 12
벡터의 원소 수 - 5
현재 벡터가 담을 수 있는 원소의 갯수 - 5
이 시스템에서 vector<int>가 담을 수 있는 최대 원소 수 - 1073741823

벡터에 들어있는 원소들
1 2 3 4 5
```

내가 강조하고 싶은 내용은 먼저 객체 `v`의 크기가 12바이트라는 것입니다. `v`는 STACK에 생성된 지역객체입니다. 12바이트의 내용은 다음 강의에 알아보겠습니다. 담을 수 있는 최대 갯수는 큰 의미가 없습니다. 시스템에서 확보할 수 있는 최대 메모리를 원소의 크기로 나눈 값에 불과합니다. 내 시스템은 최대 4GB 메모리를 사용할 수 있는 모양입니다. 왜 의미가 없다고 이야기 했나하면 `vector`의 메모리는 반드시 연속되어야 한다고 했기 때문입니다. 사용가능한 메모리가 4GB라도 중간에 1바이트라도 다른 곳에서 사용중이라면 `vector`의 메모리로 사용할 수 없습니다.

출력화면에서 알 수 있듯이 `v`의 모든 원소가 다 찼습니다. 원소 한 개를 `v`에 추가한다면 어떤 일이 일어날 것인지 위에서 설명한 것을 생각해 보면 알 수 있습니다. `vector`에 원소를 추가하는 것은 `vector`의 제일 뒷자리에 새 원소를 추가하는 것(`vector`의 `push_back` 멤버함수)을 말합니다. 중간 위치에 원소를 넣는 동작은 삽입(`insert`)이라고 합니다. `vector`의 뒤쪽에 아직 여유 공간이 있다면 원소를 추가하는 동작의 복잡도는 $O(1)$ 입니다. 예로 든 프로그램은 여유 공간이 없습니다. 원소 6을 추가한다면 `vector`의 동작은 $O(1)$ 에 그칠 리 없습니다. 이때 `vector`는 다음과 같이 동작합니다.

- 5개보다 더 큰 공간을 새로 확보한다.
- 5개의 원소를 새 공간에 복사한다.
- 원소 6을 뒤에 추가한다. `v.push_back(6)`
- 5개가 있었던 메모리를 반환한다.

복잡합니다. 얼마나 더 큰 공간을 확보하는 것이 좋을까요? 내가 결정할 일은 아니지만 궁금합니다. 프로그램에 원소를 하나 추가하는 코드를 넣고 다시 결과를 출력해 보겠습니다.

```
int main( )
{
    // 이전 메인 모두 생략

    v.push_back( 6 );           // 원소 1개 추가

    cout << endl;
    cout << "벡터 객체의 크기 - " << sizeof( v ) << endl;
    cout << "벡터의 원소 수 - " << v.size( ) << endl;
    cout << "현재 벡터가 담을 수 있는 원소의 갯수 - " << v.capacity( ) << endl;
    cout << "이 시스템에서 vector<int>가 담을 수 있는 최대 원소 수 - "
        << v.max_size( ) << endl;

    cout << endl;
    cout << "벡터에 들어있는 원소들" << endl;
    for ( int i = 0; i < v.size( ); ++i )
        cout << v[i] << " ";
    cout << endl;

    save( "소스.cpp" );
}
```

```

Microsoft Visual Studio 디버그 콘솔
벡터 객체의 크기 - 12
벡터의 원소 수 - 5
현재 벡터가 담을 수 있는 원소의 갯수 - 5
이 시스템에서 vector<int>가 담을 수 있는 최대 원소 수 - 1073741823

벡터에 들어있는 원소들
1 2 3 4 5

벡터 객체의 크기 - 12
벡터의 원소 수 - 6
현재 벡터가 담을 수 있는 원소의 갯수 - 7
이 시스템에서 vector<int>가 담을 수 있는 최대 원소 수 - 1073741823

벡터에 들어있는 원소들
1 2 3 4 5 6

```

벡터가 확보한 메모리가 원소 7개를 담을 수 있는 새 공간으로 바뀌었습니다. 찾아보면 메모리 공간의 주소도 출력해 볼 수 있을 겁니다. 이 프로그램을 관찰이 쉽도록 바꿔 보겠습니다. 1초에 10개씩 원소를 추가하며 공간이 바뀔 때마다 vector의 정보를 출력하겠습니다. vector가 어떤 원칙으로 공간의 크기를 늘려가는지 관찰해 보세요.

```

#include <iostream>
#include <vector>
#include <thread>
#include "save.h"
using namespace std;

int main( )
{
    vector<int> v; // 비어있는 벡터

    int capacity { -1 };
    int i { 0 };

    while ( true ) {
        v.push_back( ++i ); // 원소 추가
        cout << i << " ";
        this_thread::sleep_for( 100ms );

        if ( capacity != v.capacity( ) ) {
            cout << endl;
            cout << "원소 수 - " << v.size( ) << endl;
            cout << "메모리 재할당 후 담을 수 있는 원소의 갯수 - "
                << v.capacity( ) << endl;
            cout << endl << endl;

            capacity = v.capacity( );
            cout << "원소 추가 - ";
        }
    }
}

```



```

    }
}

```

이것이 이해해야 할 vector의 핵심동작입니다. 우리 책은 좋은 책입니다. 그래서 이와 유사한 프로그램이 96쪽에 있습니다. 가능한 한 vector가 메모리를 재할당 하지 않도록 하는 것이 vector 사용의 핵심입니다. “아니! 벡터는 동적배열이고 메모리가 다 차면 알아서 새 메모리를 확보하는 것이 매력인데 그렇게 하지 않도록 사용하라니 이게 무슨 이야기야.”라고 생각할 수도 있습니다. 그런데 그렇지 않습니다. 진짜 어쩔 수 없는 경우 메모리가 재할당되는 것은 피할 수 없는 일이지만 그렇게 되지 않도록 예상하고 신경써서 초기 메모리를 확보하는 것이 벡터의 올바른 사용법 되겠습니다.

원소를 딱 100개를 사용한다면 array를 사용하고 100개면 될 것 같은데 만에 하나 120개가 되더라도 프로그램이 실행되도록 하고 싶은 경우 vector를 사용하는 것입니다. vector는 정말 그렇게 하라고 생성 시 메모리를 확보할 수 있는 멤버를 제공합니다. 생성자의 인수로 확보할 메모리를 요청할 수 있지만 이것보다는 메모리를 예약해 달라고 부탁하는 함수 `reserve`를 사용합니다. 위의 프로그램에 한 줄 추가하겠습니다. 메모리 재할당이 일어나지 않은 것을 관찰하세요.

```

int main( )
{
    vector<int> v;                                // 비어있는 벡터

    v.reserve( 200 );                             // int 200개 공간 예약

    int capacity { -1 };
    int i { };
}

```

그럼 vector에 진짜 원소를 넣어 보겠습니다.

```

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

int main( )
{
    vector<String> v;

    v.push_back( String( "코로나" ) );

    // 원소를 관리하는 컨테이너는 범위기반의 for를 사용하자
    for ( String s : v )

```

```

        cout << s << endl;

        save( "소스.cpp" );
    }

```

```

Microsoft Visual Studio 디버그 콘솔
생성자(char*): 객체-00D8FBD0, 길이-6, 메모리-01259E20
이동생성자: 객체-0125A288, 길이-6, 메모리-01259E20
소멸자: 객체-00D8FBD0, 길이-0, 메모리-00000000
복사생성자: 객체-00D8FBD0, 길이-6, 메모리-01259E70
코로나
소멸자: 객체-00D8FBD0, 길이-6, 메모리-01259E70
소멸자: 객체-0125A288, 길이-6, 메모리-01259E20

```

vector v에 원소를 딱 하나 넣었습니다. String의 관찰메시지를 보고 프로그램의 동작을 추적할 수 있겠습니까? 원소를 하나 더 추가하고 실행해 보겠습니다.

```

#include "save.h"
using namespace std;

int main( )
{
    vector<String> v;

    v.push_back( String( "코로나" ) );
    cout << "----- 원소추가 -----" << endl;
    v.push_back( String( "바이러스" ) );

    // 원소를 관리하는 컨테이너는 범위기반의 for를 사용하자
}

```

```

Microsoft Visual Studio 디버그 콘솔
생성자(char*): 객체-004FF924, 길이-6, 메모리-0063A3E8
이동생성자: 객체-00638FB8, 길이-6, 메모리-0063A3E8
소멸자: 객체-004FF924, 길이-0, 메모리-00000000
----- 원소추가 -----
생성자(char*): 객체-004FF924, 길이-8, 메모리-0063A378
이동생성자: 객체-00635184, 길이-8, 메모리-0063A378
이동생성자: 객체-00635178, 길이-6, 메모리-0063A3E8
소멸자: 객체-00638FB8, 길이-0, 메모리-00000000
소멸자: 객체-004FF924, 길이-0, 메모리-00000000
복사생성자: 객체-004FF924, 길이-6, 메모리-0063A298
코로나
소멸자: 객체-004FF924, 길이-6, 메모리-0063A298
복사생성자: 객체-004FF924, 길이-8, 메모리-0063A388
바이러스
소멸자: 객체-004FF924, 길이-8, 메모리-0063A388
소멸자: 객체-00635178, 길이-6, 메모리-0063A3E8
소멸자: 객체-00635184, 길이-8, 메모리-0063A378

```

객체 2개를 추가하는데 이렇게 많은 special 함수가 불리다니요. **괜찮은가요?** 그럴 리 없습니
다. 메모리를 미리 확보하면 이 메시지를 줄일 수 있겠네요. 예약하고 다시 실행합니다.

```

Microsoft Visual Studio 디버그 콘솔
생성자(char*): 객체-0097FB80, 길이-6, 메모리-00DCD000
이동생성자: 객체-00DC9468, 길이-6, 메모리-00DCD000
소멸자: 객체-0097FB80, 길이-0, 메모리-00000000
----- 원소추가 -----
생성자(char*): 객체-0097FB80, 길이-8, 메모리-00DCCFF0
이동생성자: 객체-00DC9474, 길이-8, 메모리-00DCCFF0
소멸자: 객체-0097FB80, 길이-0, 메모리-00000000
복사생성자: 객체-0097FB80, 길이-6, 메모리-00DCD150
코로나
소멸자: 객체-0097FB80, 길이-6, 메모리-00DCD150
복사생성자: 객체-0097FB80, 길이-8, 메모리-00DCD050
바이러스
소멸자: 객체-0097FB80, 길이-8, 메모리-00DCD050
소멸자: 객체-00DC9468, 길이-6, 메모리-00DCD000
소멸자: 객체-00DC9474, 길이-8, 메모리-00DCCFF0
  
```

어딘가 변화가 있을텐데 어딘지 찾을 수 있습니까? 잘 찾아보세요. 뭔가 확 줄었을 것 같은데 별 효과 없다고요? 그렇지 않습니다. 원소 갯수가 하나에서 두 개로 바뀐것이라 그렇지, 안 그러면 관찰 메시지가 쏟아집니다. 그래도 그렇지 뭔가 생각보다 너무 많은 메시지가 나오는 것 같지 않으세요? **이 프로그램은 순 영터리 프로그램입니다. C++ 잘 못 배우면 이런 프로그램 짜 놓고 C++이 C보다 속도가 느리다는 헛소리를 하게 되는 겁니다.**

이 프로그램 어디가 잘못되었는지 바로 지적할 수 있어야 합니다. 바꿔 보겠습니다.

```

int main( )
{
    vector<String> v;
    v.reserve( 2 );

    v.push_back( String( "코로나" ) );
    cout << "----- 원소추가 -----" << endl;
    v.push_back( String( "바이러스" ) );

    // 원소를 관리하는 컨테이너는 범위기반의 for를 사용하자
    for ( String& s : v )
        cout << s << endl;

    save( "소스.cpp" );
}
  
```

쓸데 없이 객체가 복사되지 않도록 언제나 신경을 곤두세워 코딩해야 합니다. 실전 코딩에서 관찰메시지를 보고 코딩하면 안됩니다. 그냥 손가락이 자동으로 &(레퍼런스) 붙여야 합니다. & 뭐라구요? **&는 “복사하지 않겠어”라는 말입니다.** 원본을 그대로 사용하겠다는 것입니다.

```

Microsoft Visual Studio 디버그 콘솔
생성자(char*): 객체-00EFFE7C, 길이-6, 메모리-00F8D378
이동생성자: 객체-00F85908, 길이-6, 메모리-00F8D378
소멸자: 객체-00EFFE7C, 길이-0, 메모리-00000000
----- 원소추가 -----
생성자(char*): 객체-00EFFE7C, 길이-8, 메모리-00F8D318
이동생성자: 객체-00F85914, 길이-8, 메모리-00F8D318
소멸자: 객체-00EFFE7C, 길이-0, 메모리-00000000
코로나
바이러스
소멸자: 객체-00F85908, 길이-6, 메모리-00F8D378
소멸자: 객체-00F85914, 길이-8, 메모리-00F8D318

```

이 정도만 이해하고 코딩해도 어디가서 욕먹지 않을겁니다. 그런데 최신 기술을 부지런히 연마한 프로그래머가 보면 혹시 욕할지도 모르겠네요. **공부하라고 말이죠!**

```

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

int main( )
{
    vector<String> v;
    v.reserve( 2 );

    v.emplace_back( "코로나" );
    cout << "----- 원소추가 -----" << endl;
    v.emplace_back( "바이러스" );

    // 원소를 관리하는 컨테이너는 범위기반의 for를 사용하자
    for ( String& s : v )
        cout << s << endl;

    save( "소스.cpp" );
}

```

이것이 최상입니다. 더 멋지게 코딩할 수는 없습니다. 있으면 알려주세요. 반성하겠습니다. C++ 공부를 게을리 한것을.

수업 스타일을 살려보려 뒷부분은 평소 강의하는 대로 적어보았습니다. 출력화면을 관찰하는 것이 중요합니다. 지금은 어렵겠지만 메모리를 그려보며 이해해야 합니다. C++ 코딩은 메모리를 이해하는 것이 핵심입니다. 다음 시간 벡터와 벡터의 메모리 관리를 공부하겠습니다.

[4주 2일 - 과제] 지금까지 강의자료를 다시 읽어보기.