STL - 5주 2일

Wulong

계속해서 메모리를 살펴봅니다. 지난 시간에 vector<int>까지 살펴보았습니다. 누구나 좋아하고 편하게 사용하는 동적배열인 vector가, 어떻게 더 큰 메모리를 동적으로 만들어 주는지 알아봤습니다. int와 같은 POD를 다루려고 컨테이너를 사용하지는 않습니다.

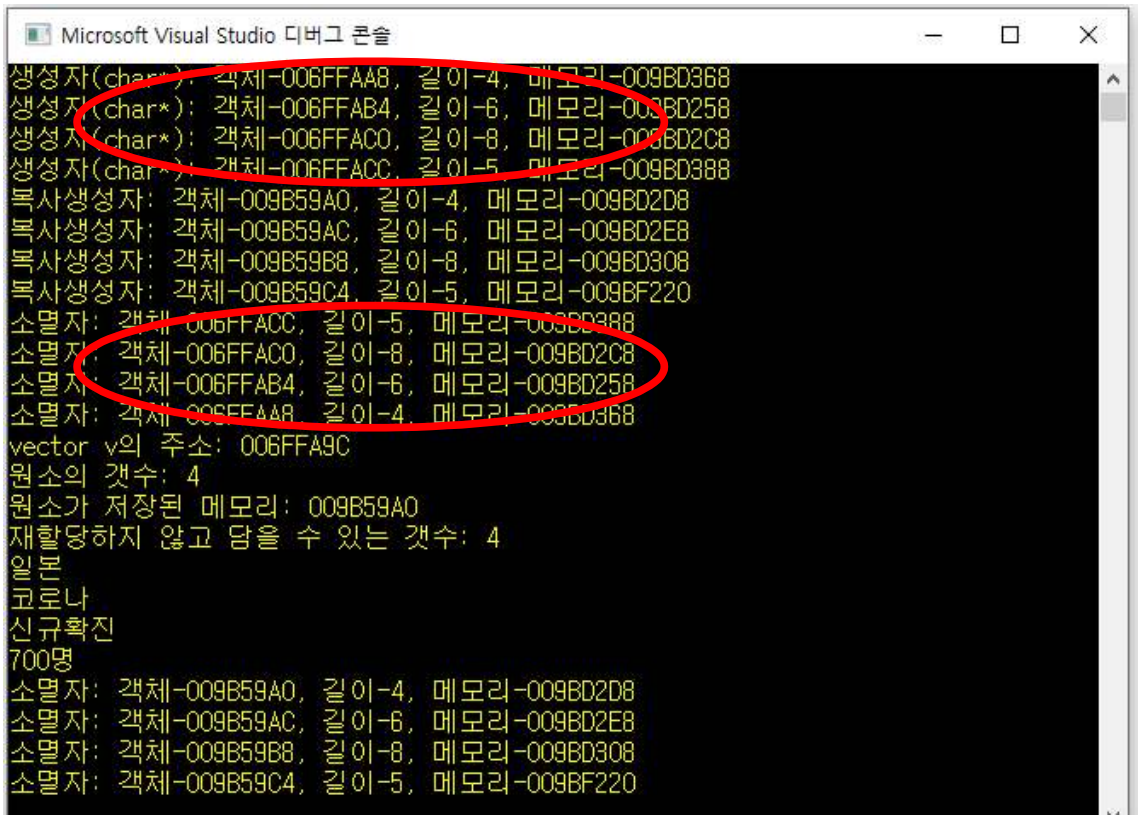
vector를 제대로 이용하려면 사용자가 만든 객체를 저장한다는 것과, 객체는 멤버를 갖고 있으므로 객체 마다 각자의 상태를 가지며 그에 따른 동작을 하리라는 것도 잘 이해하고 코딩해야 합니다. **vector가 하지 않아도 되는 동작** 즉, 프로그래머가 어디에서 무슨 일이 일어나는지 이해하지 못하여 **쓸 데 없는 함수가 호출되게 만드는 동작을 줄이는 것이 vector를 제대로 사용하는 것**입니다. 지금까지와 같이 강의 이외에 필요한 내용은 책이나 검색을 통해 공부하고, 코드는 눈으로 읽지 말고 꼭 단계별로 실행시키며 따라해 보기 바랍니다. 그럼 진짜 vector 코딩하러 가겠습니다.

벡터와 클래스 객체

String 객체를 vector에 저장하며 관찰메시지를 모두 이해할 때 까지 조금씩 실행시키며 다음 코드를 연습합니다.

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

int main( )
{
    vector<String> v { "일본", "코로나", "신규확진", "700명" };

    cout << "vector v의 주소: " << &v << endl;

    cout << "원소의 갯수: " << v.size( ) << endl;
    cout << "원소가 저장된 메모리: " << v.data( ) << endl;
    cout << "재할당하지 않고 담을 수 있는 갯수: " << v.capacity( ) << endl;

    // 컨테이너의 모든 원소를 순회하려면 range-based for를 사용하자
    for ( const String& str : v )
            cout << str << endl;

}
```

실행화면입니다. 실행 후 여러분 화면을 보세요. 관찰메시지가 나오게 해 놓은 상태입니다.



vector<String> v를 생성할 때 초기화 리스트(initializer list)로 { "일본", "코로나", "신규확진", "700명" }; 으로 생성하였기 때문에 v의 원소 개수(size)는 4, v의 용량(capacity)도 4가 됩니다. 용어를 섞어 썼다고 혼란스러워 할 필요는 없습니다. 다 같이 많이 사용하는 용어라 그냥 쓰겠습니다.

출력 화면을 잘 읽어 보세요. 마음에 들지 않는 곳이 있습니까. 어떤 곳이 거슬립니까. 천천히 살펴보세요. 메모리를 보고 다른 사람에게 이 프로그램에서 일어나는 일을 설명할 수 있겠습니까?

제일 먼저 봐야 하는 것은 객체의 생성과 소멸의 개수가 맞는 것입니다. 생성 4번 복사생성 4번 소멸 8번이니 갯수는 정확하게 맞습니다. 프로그램 코드를 다시 보세요. 프로그램에서 객체를 8번 생성했나요? 아닙니다. 눈에 보이는 객체 생성은 초기화 리스트를 이용하여 4개의 객체를 만든 것 밖에 없습니다. 그런데 메모리를 잘 관찰해보니 그냥 생성자로 만든 객체 4개는 복사생성 이후 모두 소멸된 것을 알 수 있습니다. 이들 쌍은 출력창에 동그라미 표시를 해 놓겠습니다. 그러면 복사생성된 객체 4개는 왜 복사생성된 것일까요? 내가 만든 String 객체는 깊은 복사를 하는 객체라서 복사비용이 만만치 않을텐데 말이죠!

어떤 객체가 STL 컨테이너의 원소가 되려면 몇 가지 조건을 만족시켜야 합니다. 그 중에 "컨테이너의 원소는 복사생성가능(copy-constructable)하여야한다"라는 항목이 있습니다. 즉, 객

체를 복사생성할 수 없다면 vector의 원소가 될 수 없다는 것입니다. 이제 프로그램의 동작을 보고 어떤 일이 일어난 것인지 순서대로 설명해 보겠습니다.

- 초기화 리스트를 사용하여 STACK에 String 객체 4개를 생성하였다.
- v를 STACK에 생성하였고 String 객체 4개가 저장될 메모리를 HEAP에 생성하였다.
- STACK에 생성된 String 객체들을 HEAP에 복사생성하였다.
- STACK에 있는 String 객체는 더 이상 사용하지 않으므로 파괴하였다.
- 지역객체인 v가 지역을 벗어나며 자동으로 소멸자가 호출된다.
- v는 자신이 관리하고 있는 4개의 String 객체의 소멸자를 호출한다.

간단하게 정리해 봤습니다. 얼마든지 더 자세하게 설명할 수도 있습니다만 이 정도도 충분합니다. 컴파일러는 어떻게 STACK에 생성한 객체를 모두 파괴할 수 있는 코드를 생성할 수 있었을까요? 그것은 이 객체들이 이름 없는 객체들이기 때문입니다. 이름 없는 객체를 다시 사용할 수는 절대로 없으니까 그렇습니다.

String의 관찰메시지 덕분에 복사생성된 객체는 HEAP에 생성됨을 확인할 수 있습니다. 그럼 지금 관찰한 것에서 이런 의문을 가져야 합니다. 왜 더 사용할 수도 없는 객체를 만든 후 그것을 복사생성해서 v에 옮겨야 할까? 어떻게 줄일 수 없을까?

그렇게 할 수 있습니다. 자. 지금 우리의 목표는 쓸데없는 생성과 소멸 각 4번을 줄이는 것입니다. 여러분도 이미 알고 있는 것이 있죠? 4주 2일 강의의 마지막에 있는 내용입니다.

```
int main( )
{
    vector<String> v;

    v.emplace_back( "일본" );
    v.emplace_back( "코로나" );
    v.emplace_back( "신규확진" );
    v.emplace_back( "700명" );

    cout << "vector v의 주소: " << &v << endl;

   이하 생략
```

vector의 emplace_back 멤버는 확보한 메모리에 vector가 직접 String을 생성하는 함수입니다. 따라서 함수의 인자로 String 생성 시 필요한 인자만 건네줍니다. 이것을 모르면 이렇게 코딩할 수 있습니다.

```
    v.emplace_back( String("일본") );
```

제 정신이 아닌 일본만큼이나 바보 같은 코드입니다. 물론 돌아는 갑니다. 우리는 관찰메시지

를 보며 공부하고 있으니까 바로 잘못된 것을 알 수 있습니다만 실전 코드에 관찰메시지가 있을 리 없습니다. **항상 메모리를 생각하며 코딩하는 겁니다.** 잊지 마세요. 자 그럼 다 같이 실행시켜 보고 출력화면 관찰해 주세요.

다 잘 되죠? 네? 뭐라구요! 아까 보다 더 복잡해졌다구요. 그럴~리가요... 잠깐만요. 나도 실행시켜 보겠습니다. Ctrl+F5. 얍!

이런. 이런. 한 번에 성공하는 적은 거의 없다니까요. 한 번에 성공하면 프로그래머가 아니죠! 틀려도 아무 상관없어요. 얼른 반성하고 고치면 되니까요. 시간도 안 걸리는데요. 뭐. 내가 학생일 때는 천공카드에 연필로 코딩해서 전산실 갖다 주고 실행결과 보려면 일주일이 걸렸고 한 글자라도 잘못적거나 오퍼레이터가 잘못 타이핑하면 처음부터 다시 해야 했습니다. 그런데 컴퓨터가 점점 빨라지더니 대학원 때는 FORTRAN으로 짠 FFT 실행시키고 결과 보려면 한 시간 정도 기다리면 되서 담배피고 놀다왔던 기억이 있네요. 그러니 프로그램을 얼마나 배웠겠어요. 지금은 누르자마자 실행화면 나오니까 계속 이것저것 생각나는 대로 코딩하면 되죠. 아이구! 또 놀고 있었네요. 여러분 출력화면 보고 어떤 일이 일어나는지 아시겠죠. vector가 자꾸 메모리를 늘리면서 이사하고 있네요. 이렇게 해야죠.

```
int main( )
{
    vector<String> v;

    v.reserve( 4 );

    v.emplace_back( "일본" );
    v.emplace_back( "코로나" );
    v.emplace_back( "신규확진" );
    v.emplace_back( "700명" );

    이하 생략
```

문제점 해결되었습니다. 벡터가 쓸데없는 동작을 하지 않도록 메모리 공간을 미리 잡아 놓았습니다. 또 객체도 vector가 직접 만들도록 해서 생성과 소멸을 최소로 하였습니다. 이 보다 더 줄일 수는 없습니다.

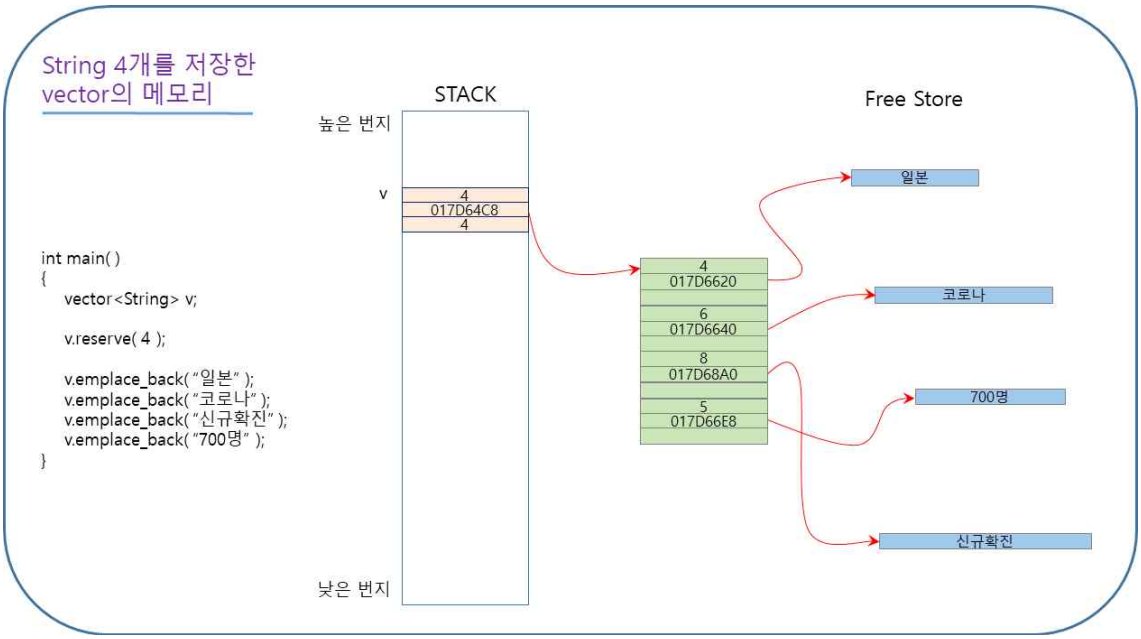
컨테이너에 원소를 넣은 것은 뭔가 필요한 동작을 하고 싶기 때문이겠죠. 마음껏 사용하면 됩니다. vector의 원소들을 모두 순회하며 화면 출력하는 코드 바로 위에 오름차순으로 정렬하는 코드를 다음과 같이 추가해 봤습니다.

```
    생략

    v.emplace_back( "신규확진" );
    v.emplace_back( "700명" );
```

```
cout << "오름차순으로 정렬합니다" << endl;
sort( v.begin( ), v.end( ), [](  String a, String b ) {
        return a.getString( ) < b.getString( );
        } );

cout << "오름차순으로 정렬한 결과입니다" << endl;

// 컨테이너의 모든 원소를 순회하려면 range-based for를 사용하자
for ( const String& str : v )
        cout << str << endl;

생략
```

실행시켜 보세요. 그리고 꼭 아세요. 이 코드는 쓰레기라는 것을!

vector<String> v에 4개의 String 객체가 삽입된 모습을 그림으로 남깁니다. 잘 살펴보세요.



지금까지 잘 따라온 여러분 칭찬합니다. 안에서 어떤 일이 일어나서 어떻게 이런 그림이 나오게 된 것인지 이제 나도 알고 남도 알게 해 줄 수도 있을 겁니다. 자! 그럼 여러분 믿고 한 걸음 더 나아갑니다.

컨테이너에서 객체의 복사와 이동

가능하다면 그런 일이 일어나지 않으면 좋겠지만 v.capacity()가 4, v.size()도 4인 현재 vector v에 String 객체를 하나 추가하려고 합니다. 일단 코드 전체를 적어 보겠습니다.

#include는 생략합니다.

```
int main( )
{
    vector<String> v;

    v.reserve( 4 );

    v.emplace_back( "일본" );
    v.emplace_back( "코로나" );
    v.emplace_back( "신규확진" );
    v.emplace_back( "700명" );

    cout << "------------------" << endl;
    cout << "원소를 추가하기 전" << endl;
    // v를 사용하여 원하는 일을 할 만큼했다. 그런데 원소가 한 개 더 필요하다.
    // 아래와 같이 원소를 추가하면 어떤 일이 일어날까?

    v.emplace_back( "PANDEMIC" );

    cout << "원소를 추가한 후" << endl;
    cout << "------------------" << endl;
}
```

실행화면의 일부입니다.

```
------------------
원소를 추가하기 전
생성자(char*): 객체-00105140, 길이-8, 메모리-0010D020
이동생성자: 객체-00105110, 길이-4, 메모리-0010D1B0
이동생성자: 객체-0010511C, 길이-6, 메모리-0010D0B0
이동생성자: 객체-00105128, 길이-8, 메모리-0010D0C0
이동생성자: 객체-00105134, 길이-5, 메모리-0010D100
소멸자: 객체-001058F0, 길이-0, 메모리-00000000
소멸자: 객체-001058FC, 길이-0, 메모리-00000000
소멸자: 객체-00105908, 길이-0, 메모리-00000000
소멸자: 객체-00105914, 길이-0, 메모리-00000000
원소를 추가한 후
------------------
```

vector v가 6개의 String 객체를 저장할 공간을 확보한 후 새 원소를 생성하고 이전 4개 객체를 새 메모리에 이동생성 하였음을 알 수 있습니다. 물론 이전 공간에 있던 원소들은 삭제하였습니다. 우리가 사용 중인 **String 객체는 이동문법을 지원**합니다. 그럼 vector는 이동을 지원하지 않는 객체는 원소로 사용할 수 없나요? 그렇지 않죠. 오늘 강의 처음에서 이야기 했잖아요. 컨테이너의 원소는 복사가 기본입니다. 이동을 지원하는 더 좋은 객체가 있다면 복사 대신에 이동을 사용하여 성능을 더 높일 수 있을 뿐입니다. 이전에 만들어진 class들은 이동을 지원하지 않을 수 있습니다. 그래도 vector의 원소로 사용하는 데 문제는 없습니다. 여기서 한 가지 더!

"만일 어떤 객체가 복사는 지원하지 않는데 이동은 가능하다면 그 객체를 컨테이너의 원소로 사용할 수 있나요?" 이런 훌륭한 질문을 하는 학생도 있습니다. 그렇습니다. 그런 객체가 있다면 컨테이너의 원소로 사용할 수 있습니다. 여러분도 그런 객체를 알고 있습니다. 분명히!

이동을 지원하지 않는 경우 컨테이너의 동작을 확인해보기 위해 String의 이동생성과 이동할당연산자를 주석처리하고 실행해 보았습니다. 여러분은 구경만 하세요. 결과화면입니다.

```
------------------
원소를 추가하기 전
생성자(char*): 객체-013F5178, 길이-8, 메모리-013F9E98
복사생성자: 객체-013F5148, 길이-4, 메모리-013F9EC8
복사생성자: 객체-013F5154, 길이-6, 메모리-013F9E38
복사생성자: 객체-013F5160, 길이-8, 메모리-013FF070
복사생성자: 객체-013F516C, 길이-5, 메모리-013FEF80
소멸자: 객체-013F5110, 길이-4, 메모리-013F9F48
소멸자: 객체-013F511C, 길이-6, 메모리-013F9F38
소멸자: 객체-013F5128, 길이-8, 메모리-013F9E08
소멸자: 객체-013F5134, 길이-5, 메모리-013F9E28
원소를 추가한 후
------------------
```

이동을 지원하지 않는 클래스라면 복사하면 됩니다. 이것을 **이동의미론**(move semantic)이라고 합니다. 어떤 클래스가 이동을 지원한다면 왜 그런 겁니까? 왜 클래스에 이동생성자를 코딩하죠? 대답해 보세요. 복사생성보다 이동생성이 좋기 때문이라구요? 그건 당연한 대답이고 진짜 이유는 무엇입니까? 그렇죠. 그것은 "**클래스가 자원을 할당하고 있기 때문**"이라고 답을 해 줘야죠. 클래스와 관련된 복잡한 이야기가 바로 거기서 시작되는 거였죠.

여러분 진짜 잘 따라 왔습니다. 이것까지 이해하면 더 복잡한 것은 없습니다. 혹시 누군가 메모리를 더 꼬아 사용한 코드가 있더라도 여기까지 이해한 여러분에게 어려운 메모리는 없습니다. 이것이 진짜 vector입니다. 다른 활용 예제를 조금 후에 보이기로 하고 여기서 이번 주 과제를 내겠습니다.

[설명] 위의 프로그램은 vector<String> v에 4개의 String을 저장하였다. 메모리 구성은 그림과 같다. 이때 강의록에서 설명한 것과 같이 capacity가 4인 상태에서 v.emplace_back("PANDEMIC");을 실행하였다고 하자. v는 새 메모리를 할당받아 원소를 이동할 것이다.

[과제] 원소를 추가하기 전과 원소를 추가한 후의 메모리를 종이에 그림 그려라.

[제출] 그린 그림을 사진촬영한 후 이미지 파일을 과제로 제출한다.
(사진 한 장으로 충분할 것이다)
[기한] 6주 2일 강의 시작 전

[평가] 제출한 사진을 보고 핵심 내용이 맞는지 10점 만점으로 점수 평가함.
(언제나 그렇듯 다른 사람과 적극적으로 상의하는 것을 권장한다)

벡터의 원소 제거

vector에 정수를 저장하고 조건에 맞는 원소를 제거해 보겠습니다. 코드를 입력해 주세요.

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

int main( )
{
    vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // [문제]를 해결하는 코드를 작성하라

    for ( int n : v )                    // int의 경우 복사나 const &나 차이없음
            cout << n << ' ';
    cout << endl;
}
```

v에 1부터 10까지 정수 10개가 저장되어 있습니다. **v에 있는 원소를 제거**하려고 합니다. 어떤 것을 제거해야 하는지는 원하는 사람마음입니다만 어떤 것을 표현하는 것은 다음과 같이 세 가지 방식이 있습니다.

- **숫자 3을 제거**
- **앞에서부터 3번째 원소를 제거**
- **짝수를 모두 제거**

그런데 이 방식은 모두 다른 함수로 구현되어 있어 처음 배울 때는 제대로 사용하는 것이 쉽지 않습니다. 알아보겠습니다.

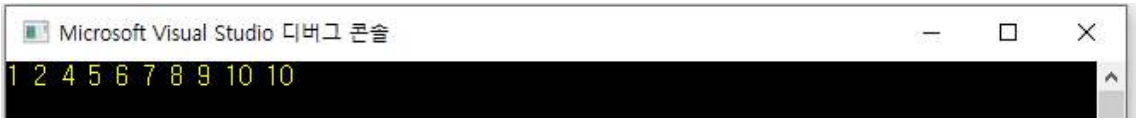
[문제] v에서 3을 제거하라.

컨테이너에 있는 원소를 대상으로 원하는 동작을 하려면 알고리즘 함수를 사용하는 것이 기본입니다. 알고리즘 함수는 모두 전역함수입니다. 제거하는 알고리즘 함수는 **remove**입니다. 문제를 해결하는 코드는 다음과 같습니다.

```
    remove( v.begin( ), v.end( ), 3 );
```

remove 함수는 처리할 구간을 인자로 받습니다. 지금은 전체입니다. 세 번째 인자로 v에 있는 원소와 같은 타입인 3을 받았습니다. 다른 타입을 인자로 전달하면 컴파일되지 않습니다. 알고리즘 함수 remove는 vector<int> v와 관계없습니다. 그저 v.begin(), v.end()라는 반복

자(반복자는 클래스이며 v와는 전혀 다른 타입입니다)라는 반복자만 전달되었을 뿐입니다. 하지만 제거해야 할 타입이 맞는지를 귀신같이 알아냅니다. 어떻게 알 수 있는 걸까요? 그것은 전달된 반복자로 부터 필요한 정보를 알아낼 수 있는 기법이 있기 때문입니다. 어쨌든 프로그램의 문제 부분에 해결 코드를 넣고 실행해 보겠습니다.



어떻습니까? 3이 확실하게 제거되었죠? 그렇습니다. 3은 제거되었습니다. 그런데, 그런데 말입니다. 뭔가 이상하지 않습니까? vector의 size()를 출력하지는 않았지만 v의 원소는 9개가 되어야 하는데 원소는 그대로 10개입니다. 줄지 않았습니다. 이것을 이해하는 것이 아주 중요합니다. STL의 **핵심**이라고 할 수 있습니다.

이유는 이렇습니다. 알고리즘 함수 remove는 반복자를 전달받아 반복자를 이용하여 v의 원소에 접근합니다. 접근은 읽기/쓰기를 의미합니다. remove는 반복자로 원소들을 살펴보다가 반복자가 가리키는 원소의 값이 3이면 그 원소를 다음 원소의 값으로 덮어쓸 수 있습니다(책에 상세한 그림이 있습니다). 말 그대로 값을 쓸 수만 있습니다. 이번 주 강의에서 계속 메모리 그림을 그리며 공부했습니다. 위의 프로그램에서 v는 STACK에 있는 변수이고 개수, 용량, 메모리위치를 저장하고 있습니다. 알고리즘 함수 remove가 현재 원소의 개수 10을 9로 바꿀 재주가 있습니까? 전혀 없습니다. 없어요.

메모리를 그려보세요. remove에게 전달한 것은 HEAP에 있는 1부터 10까지 메모리의 위치입니다. 그러니 remove가 STACK에 저장된 숫자 10을 9로 바꿀 수는 없는 것이죠. 게다가 v의 모든 변수는 private입니다. 귀신 할애비가 와도 이 숫자를 바꿀 수는 없어야 한다는 것이 class의 기본 설계 개념 아닙니까!

그럼, 알고리즘 함수는 객체를 지우긴 지웠는데 vector에게 지웠다고 어떻게 알려줘야 할까요? vector에게 뭔가 달라졌다고 알려줘야 vector가 원소 개수를 9로 바꿀 것 아닙니까. 알고리즘과 컨테이너 사이에 있는 것은 단 하나 반복자밖에 없습니다. 그러니 반복자를 사용해서 알려줘야 합니다. remove 함수는 반복자를 return 합니다. 이 반복자는 v에서 의미있는 데이터 구간의 끝을 가리킵니다. 우리 코드에서는 실행화면의 마지막 10을 가리키는 반복자를 리턴하는 것입니다. 그러면 실제 개수를 조절하는 것은 vector의 몫이니 다음과 같이 코딩합니다. 원하는 위치를 제거하는 함수는 vector의 멤버함수가 될 수밖에 없겠죠. 함수이름은 erase입니다.

```
// [문제] v에서 3을 제거하라
auto i = remove( v.begin( ), v.end( ), 3 );
v.erase( i );
```

실행하면 이제 우리가 원하는 결과와 같습니다.

이 해결방법은 보통 다음과 같이 한 줄로 코딩하며 **erase-remove idiom**이라고 합니다.

```
// [문제] v에서 3을 제거하라
v.erase( remove( v.begin( ), v.end( ), 3 )  );
```

다음으로 조건에 맞는 원소를 제거해 보겠습니다.

[문제] v에서 홀수를 제거하라.

어떻게 할까 잠깐 생각해 보세요. 데이터는 v에 모두 있고 제거하는 것은 알고리즘 함수가 해줄 일입니다. 그런데 알고리즘함수에게 "홀수인 원소들만 찾아 지워주세요"라고 부탁하면 되는 상황이잖아요. 지금까지 강의만으로도 어떻게 해결할 수 있는지 이해하실 겁니다. 이렇게 하면 되겠습니다.

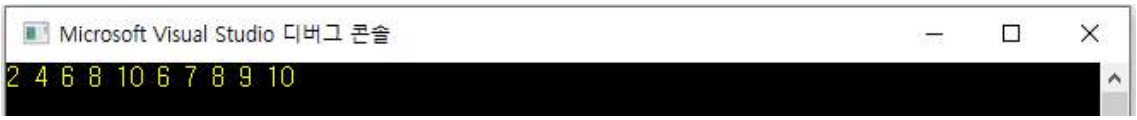
```
// [문제] v에서 홀수를 제거하라
remove( v.begin( ), v.end( ), [](  int i ) {
        if ( i % 2 )
                return true;
        return false;
} );
```

그런데 어떻습니까? 이 코드는 빨간색 줄이 쳐지지는 않지만 컴파일하면 실패합니다. 이유는 이렇습니다. C++은 함수오버로딩이 가능합니다. 그래서 인자의 타입이 다르다면 위와 같은 함수가 실행될 수도 있어야 합니다. 그런데 C++ 표준에서는 조건에 따라 달라지는 함수를 더 명확하게 표현하기로 결정하였습니다. 그래서 다음과 같은 이름을 정했습니다. 어떻습니까? 조건에 따라 처리한다는 의미가 더 명확해 졌죠?

```
remove_if( v.begin( ), v.end( ), [ ]( int i ) {
        if ( i % 2 )
                return true;
        return false;
} );
```

알고리즘 함수와 람다의 관계를 이해하는 것이 중요합니다. **알고리즘 함수는 반복자로 전달된 범위 내의 원소를 람다에게 전달하여 조건에 맞는 원소인지 물어봅니다**. 그래서 람다의 인자는 int가 되는 것이고 람다의 리턴타입은 bool이 되는 것입니다.

실행화면입니다.



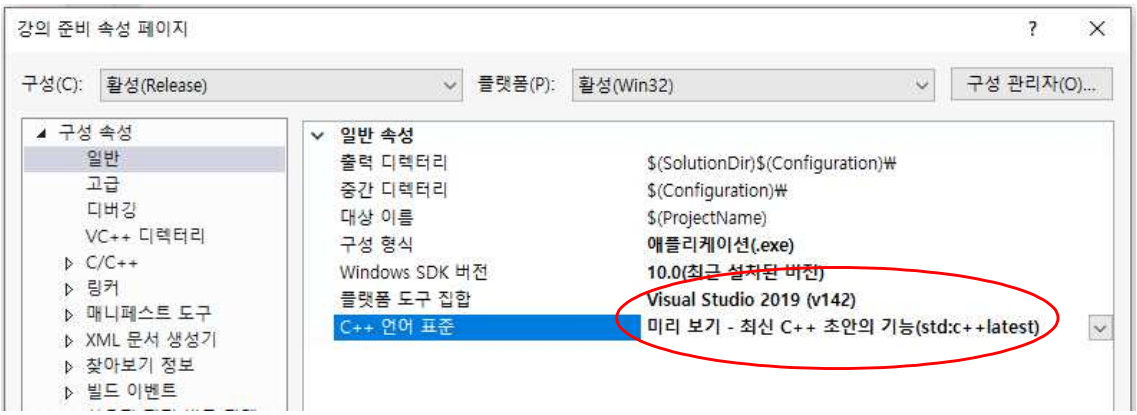
제대로 고치고 홀수 판정 부분도 진짜 프로그래머들이 쓰는 코드로 바꾸겠습니다.

```
// [문제] v에서 홀수를 제거하라
v.erase( remove_if( v.begin( ), v.end( ), [](int i ) {
        return i & 1;
} ), v.end() );
```

코드가 복잡합니다. STL 프로그래머들에겐 익숙한 코드겠지만 읽기에 쉬운 코드는 아닙니다. 개선할 수는 없을까요? 있습니다. C++20에서는 이것을 지금까지 설명한 것과 같이 복잡하게 하지 않아도 됩니다. 컨테이너가 **vector일 때는 전역함수 erase를 사용**할 수 있습니다. 그럼 지금까지 열심히 내가 한 것과 같이 erase-remove idiom을 설명한 책과 블로그는 다 없어져야 할까요? 그렇지 않습니다. 표준에서 새로운 내용이 있다고 이전 것이 한 번에 없어지지는 않습니다. 과거에 만든 프로그램과 시스템의 유지보수를 위해 이전 코드들도 진짜 없앨 거라고 하지 않는 이상 사용할 수 있습니다. **그럼 C++20으로 바꿔 볼까요?**

```
// [문제] v에서 3을 제거하라
erase( v, 3 );                                  // C++20
```

어떻습니까. 정말 간단하죠? 쉽게 찾아볼 수 없는 코드입니다. 아직 널리 전파되지 않았을 겁니다. 블로그에도 소개되지 않았을 거예요. 왜냐하면 아직 C++20이 공식 발표되지 않았기 때문이죠. 그래도 이 사양은 이미 확정된 것이어서 컴파일러들은 미리 이런 코드를 쓸 수 있도록 업데이트 할 수 있습니다. 여러분 이 코드가 그냥 빨강 줄이 쳐지나요? 그럼 이렇게 바꿔보세요. 최신 비주얼 스튜디오 설치하고 프로젝트 속성으로 들어갑니다.



조건에 맞는 원소를 지울 때는 이렇게 합니다.

```
// [문제] v에서 홀수를 제거하라
erase_if( v, [ ]( int n ) { return n & 1; } );          // C++20
```

진짜 간단하죠? 그런데 이 코드는 생각해 봐야 할 점이 있습니다. 여러분은 이 코드를 보고 어떤 감상을 가질지 모르겠습니다만 나는 이 새로운 코드가 편하지 만은 않습니다. 왜냐하면 이 코드는 **편리함을 위해 원칙을 훼손한 코드**이기 때문입니다. STL에서 컨테이너와 알고리즘은 반복자로만 소통하는 것이 대원칙인데 이 함수는 그것을 완전 무시한 것입니다. 그만큼 STL에서 vector의 사용이 많다는 이야기입니다. 나도 vector를 꽤나 상세하게 설명했네요.

여러분들 지금까지 공부한 것 복습도 할 겸 실습 문제를 하나 내고 vector를 마치겠습니다.

[실습] 다음 프로그램으로 데이터 파일 "정수 만개"를 생성하였다.

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

int main( )
{
    ofstream out( "정수 만개" );

    default_random_engine dre;
    uniform_int_distribution<> uid( 0, 10'0000 );

    for ( int i = 0; i < 1'0000; ++i ) {
            out << uid( dre ) << "  ";
            if ( i % 10 == 9 )
                    out << endl;
    }
}
```

[문제] 파일 "정수 만개"의 내용을 vector 컨테이너로 읽어 와라.
컨테이너의 원소 중에서 값이 [0, 10000)인 원소의 갯수를 세서 화면 출력하라.
값이 0이상 10000이하인 원소의 갯수는 몇 개가 되어야 하는가 적어라.
(참고) 조건에 맞는 원소의 갯수는 알고리즘 함수 count_if를 사용한다.
(주의) 저장하는 것과 읽어 오는 것은 별개의 프로그램으로 작성하라.

[고찰] 화면에 출력된 원소의 갯수는 예상한 것과 같은가?

[검증] 작성한 프로그램이 메모리와 속도면에서 최적의 코드인가?

deque

책 99 ~ 116

덱이라고 발음합니다. 원소들을 deque(double-ended queue)로 저장하는 컨테이너입니다. 컨테이너의 앞이나 뒤에 원소를 삽입하거나 삭제하는 동작의 복잡도는 $O(1)$입니다. 원소의 중간에서 삽입·삭제하는 경우의 복잡도는 $O(n)$입니다.

그림에서와 같이 deque은 원소의 앞쪽으로 그리고 원소의 뒤쪽으로 메모리가 확장되는 동적 배열과 같은 컨테이너입니다. 왜 같다는 표현을 썼냐하면 진짜 모든 메모리가 물리적으로 연속되어 있지는 않기 때문입니다. 마치 물리적으로 연속된 것처럼 내부에서 메모리를 관리하고 있기 때문에 사용자는 deque의 메모리가 붙어있다고 생각할 필요도 없이 그냥 array나 vector를 사용하는 것과 같은 방법으로 deque을 사용할 수 있습니다. 메모리가 붙은 컨테이너의 특권인 [ ] 연산자를 제공하는 것도 물론입니다. 메모리가 붙어있기 때문에 sort 함수로 원소들을 정렬할 수도 있습니다.

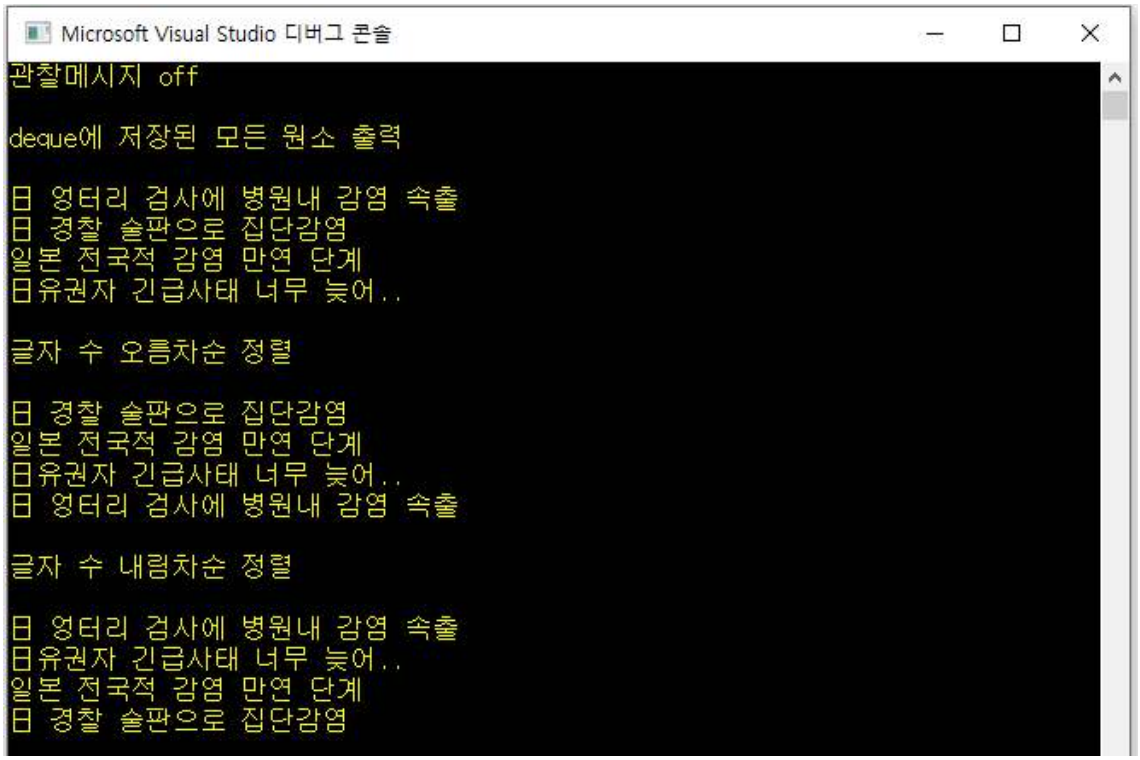
vector에 비하여 앞쪽에서 원소를 추가·삭제할 수 있다는 특징이 있어 책에 나온 것과 같이 선입선출(first-in first-out) 트랜잭션에 적합하다고 합니다만 나는 deque을 써서 프로그램해 본 적은 없습니다.

나는 deque을 이렇게 설명합니다. deque은 vector와 list의 중간에 있는 컨테이너라고요. 다음에 공부할 list는 잘 알고 있을 것입니다. list는 각 원소를 포인터로 연결하고 있죠. vector는 진짜로 모든 원소를 연속된 메모리에 저장합니다. deque은 이 둘의 중간에 있는 컨테이너입니다. 정해진 크기(이 크기는 가변입니다. 모순된 것처럼 들리겠지만!)의 메모리 블럭을 포인터로 연결하여 vector와 같이 원소에 대한 빠른 접근을 제공함과 동시에 각 블럭을 포인터로 연결하여 list와 같은 유연함을 제공하려고 하는 것입니다. deque은 정해진 블럭에 원소가 다 차면 새 블럭을 만들어 이전 블럭과 연결합니다. 그래서 컨테이너의 앞과 뒤에 $O(1)$ 복잡도로 원소를 삽입·삭제할 수 있는 것입니다. 물론 vector에서 이야기 한 것과 같이 메모리 블럭이 추가되는 비용은 따로 계산해야 합니다.

메모리가 진짜로 붙어있지 않아 원소에 접근하는 시간이 vector보다 조금 느릴 수밖에 없습니다. 그렇지만 이 느리다는 것이 블럭이 아주 많이 있지 않다면 의미 있는 것은 아닙니다. 그럼 vector보다 좋은 점은 무엇일까요? 그것은 vector에 비하여 더 많은 갯수의 원소를 담을 가능성이 크다는 것입니다. 왜냐하면 free store에서 메모리를 할당 받을때 벡터보다는 더 작은 메모리를 여러 개 할당할 수 있기 때문이죠. vector가 요구하는 메모리는 전체가 통으로 비어있어야 하므로 아주 큰 메모리를 달라고 하면 실패할 수도 있습니다. 자! 설명은 이 정도하고 간단한 연습으로 deque과 친해져 보겠습니다.

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

int main( )
{
    deque<String> news;

    news.push_back( String { "日 경찰 술판으로 집단감염" } );
    news.push_front( String { "日 엉터리 검사에 병원내 감염 속출" } );

    news.emplace_back( "일본 전국적 감염 만연 단계" );
    news.emplace_back( "日유권자 긴급사태 너무 늦어.." );

    cout << "관찰메시지 off" << endl;
    cout << endl;

    cout << "deque에 저장된 모든 원소 출력" << endl;
    cout << endl;
    for ( const String& news : news )
            cout << news << endl;
    cout << endl;


    // 글자 수 오름차순으로 정렬
    sort( news.begin( ), news.end( ), [](  const String& a, const String& b ) {
            return a.size( ) < b.size( );

            } );

    // 결과 출력
    cout << "글자 수 오름차순 정렬" << endl;
    cout << endl;
    for ( int i = 0; i < news.size( ); ++i )
            cout << news[i] << endl;
    cout << endl;

    // 결과를 반대로 출력
    cout << "글자 수 내림차순 정렬" << endl;
    cout << endl;
    for ( auto i = news.crbegin(); i < news.crend(); ++i )
            cout << *i << endl;
    cout << endl;
}
```

실행결과 화면입니다.



이건 순전히 **호기심**에 추가한 코드입니다. deque 메모리가 실제로는 연속되어 있지 않음을 알아봅니다. 오늘도 재미있게 공부하셨길 바랍니다.

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

class Test {
    int arr;

public:
    void show( ) const {
            cout << "실험 중: " << this << endl;
    }
};


int main( )
{
    deque<Test> d;
```

```
        // deque의 메모리가 연속되어 있지 않음을 확인

        for ( int i = 0; i < 10; ++i )
                d.emplace_back( );


        cout << "원소의 주소를 출력한다" << endl;
        for ( auto&& test : d )
                test.show( );
        cout << endl;

        cout << endl;
        cout << "앞쪽에 원소를 추가한 후 주소를 출력한다" << endl;
        d.emplace_front( );

        for ( int i = 0; i < d.size(); ++i  )
                d[i].show( );


        save( "소스.cpp" );
    }
```