

STL - 5주 1일

Wulong

5주 강의 2020. 4. 13 ~ 2020. 4. 17	
<p>지난 주 4.6 ~ 4.10</p>	<p>STL 컨테이너</p> <p>STL 클래스 준비 array</p> <p>vector</p>
<p>이번 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>다음 주 4.20 ~ 4.24</p>	<p>STL 컨테이너</p> <p>deque</p> <p>forward_list list</p>

지난 주

안녕하세요? STL 수강생 여러분! 4주 2일차 강의는 감동이었죠? ^^;

E-class에서는 내가 올린 강의를 누가 언제 얼마나 조회했는지 살펴볼 수 있습니다. 학생들이 같은 자료를 몇 번씩이나 조회한 걸 보면 더욱 책임감을 느낍니다. 자료만 보고 공부하는 것이 쉽지 않으리라 생각합니다. 그래도 같은 자료를 언제고 다시 볼 수 있다는 것은 좋은 점일 수도 있겠다라는 생각도 듭니다. 몇 가지 방식으로 학생들이 올린 자료를 어떻게 보고 있는지 관찰할 수 있는데 3주 2일차 강의 같은 경우 조금 어려워하는 것을 알 수 있었습니다. 조회수가 많이 떨어지더라구요. 4주 강의를 올리고 지금 5주 강의를 하는 시점인데 아직 3주 강의를 조회하지 않은 학생들도 꽤 있었습니다. 언제든지 볼 수 있다는 것이 장점이긴 하지만 계속 강의를 올리는데 별써 못 따라 오는 것은 아닌가 걱정이 됩니다. 계속하는 잔소리이긴 하지만 계속 보는 것이 공부입니다. 보고 또 보면 나중에는 안 봐도 알게 되고 마치 원래부터 알고 있었던 것처럼 생각되게 됩니다.

다시 이야기해 봅니다. 이 강의자료는 차근 차근 자세하게 모든 내용을 이야기하지 않습니다. 또한 이 자료는 읽어 보라고 만들고 있는 것이 아닙니다. 순서대로 코딩해보는 내 수업방식을 그대로 문서로 옮겨 놓은 것입니다. 3주 2일차 문제에서 코드에 문제점이 무엇이나를 묻는 문제가 있었습니다. 어떤 학생들은 무엇이 문제인지 모르겠다고 합니다. 이 문제는 코드에 있는 대로 입력하고 실행하면 화면에 그냥 문제점이 나오는 문제입니다. 내가 설명해 줄 내용이 없습니다. 모르겠다는 학생은 눈으로 읽기만 한 것입니다. 읽어서 모르면 모르는 겁니다. 읽어서 알수 있다면 이전에 해 본 것입니다. 그것도 여러 번 해 본 것입니다. 코딩은 눈으로 읽어서 배우는 것이 아닙니다.

String 클래스를 이용하여 객체의 생성과 소멸 등을 관찰하는 것이 중요합니다. '4주 1일 강의 3 - 클래스 String' 설명 영상에서 내가 실력이 모자라 마지막에 보여주고 싶은 것을 못 보여줬는데 여러분들 모두 해결하셨죠? 내가 그렇게 코딩을 잘 못합니다. 여러분이 안 도와주면 강의도 못할 정도입니다. 시간을 들여 3주 2일차 과제를 다 살펴보았습니다. 그리고 지적하고 싶은 내용도 다 적어봤습니다. 학습 효과는 아주 좋다고 생각하지만 다시 하고 싶진 않습니다. 그냥 밤 새게 되거든요.

지난 4주 강의록을 통해서 무엇이 중요한지 공부했겠지만 컨테이너에 String을 원소로 사용하며 관찰한다면 눈으로 메모리를 관찰할 수 있습니다. 강의 영상에서와 같이 String을 모두 프로젝트에 포함한 상태에서 이번 주에 vector를 관찰하겠습니다. 시작하기 전에 다시 말합니다. 책을 보고 vector가 어떤 컨테이너인지 공부해야 합니다. 여기에 적어 놓은 것은 어디에서도 문서로는 보기 힘든 나만의 강의자료입니다. 책을 보고 관련 내용을 공부한 후 따라 해 보면 더 효과가 있습니다.

먼저 메모리를 설명해 보겠습니다. 오늘은 실제 강의에서 하는 것과 같이 간단한 POD 타입에서 시작하여 객체의 메모리가 어떻게 생성되는지를 관찰해 보겠습니다. 눈으로 읽지 마시고 코드를 바꿔 쳐 가며 노트에 메모리 그림을 그려서 이해하기 바랍니다. 그럼 시작해 보겠습니다.

메모리

프로그램에서 사용되는 메모리는 전역변수가 있는 DATA, 지역변수가 있는 STACK 그리고 동적으로 할당된 메모리가 사는 Free Store(HEAP)가 있습니다. 동적변수라는 용어를 사용하지 않은 것을 눈치 채셨나요? **동적변수라는 것은 없다고 설명하고 싶어 그렇습니다.** 시스템으로 부터 동적으로 할당받은 메모리가 있을 뿐입니다. **그 메모리를 가리키는 포인터 변수는 동적변수가 아닙니다.** 이러한 것들을 제일 간단한 것부터 시작하여 실제 사용되는 제일 복잡한 경우까지 **코드와 설명 그리고 그림을 사용하여 가능한 자세하게 메모리를 설명**하겠습니다. 다만 DATA 영역은 프로그래머의 관심사가 아닙니다. 이와 관련된 내용은 이전 강의에서 잘 설명했습니다. 여기에서는 모두 STACK과 HEAP간의 관계를 설명합니다.

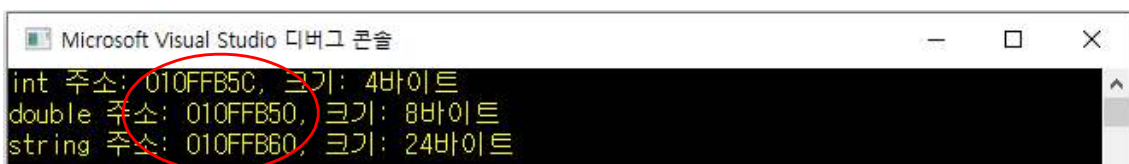
지역변수는 STACK에 생성됩니다. 물론 지역변수 외에 함수의 인자도 STACK을 사용하여 전달됩니다. 아래 프로그램을 입력하고 실행시켜 주세요.

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

int main( )
{
    int i { 1 };
    double d { 2 };
    string s { "3" };

    cout << "int 주소: " << &i << ", 크기: " << sizeof( i ) << "바이트" << endl;
    cout << "double 주소: " << &d << ", 크기: " << sizeof( d ) << "바이트" << endl;
    cout << "string 주소: " << &s << ", 크기: " << sizeof( s ) << "바이트" << endl;
}
```

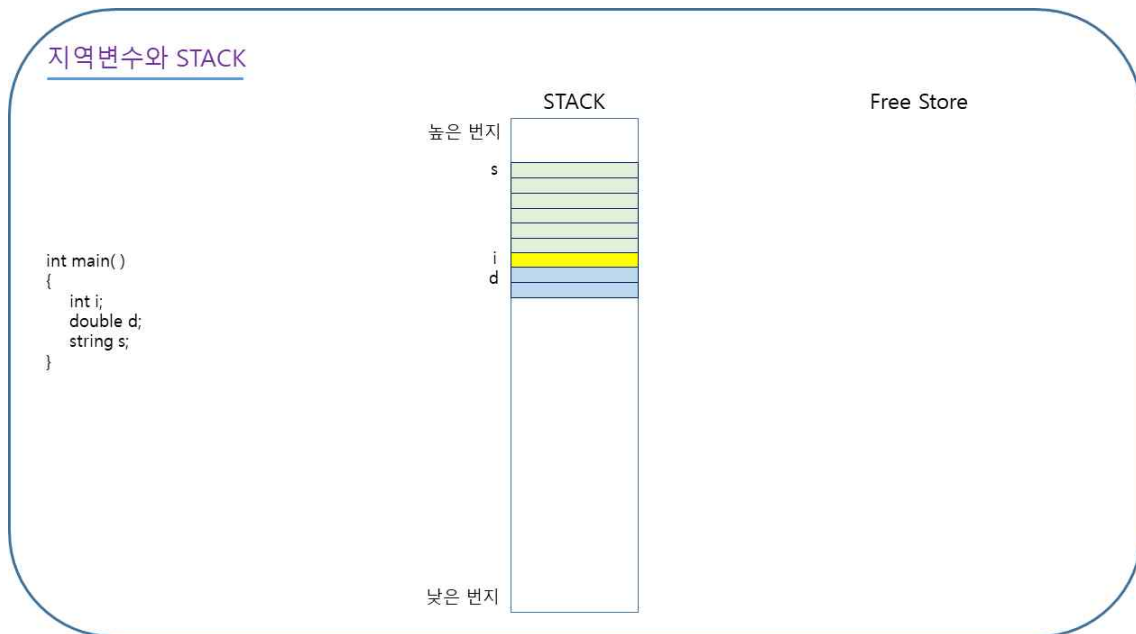
아래는 실행화면입니다. 주소는 그때 그때 다릅니다. 그때 그때는 프로그램이 실행될 때입니다. 메모리의 크기는 실행화면과 같을 겁니다. 메모리 크기가 다른 학생있나요? 손 들어 보세요. 뭐가 다르죠? 아! string의 크기가 28바이트라고요. 계속 강조했잖아요. 비주얼 스튜디오의 빌드 모드를 Release로 해 주세요.



```
Microsoft Visual Studio 디버그 콘솔
int 주소: 010FFB50, 크기: 4바이트
double 주소: 010FFB50, 크기: 8바이트
string 주소: 010FFB60, 크기: 24바이트
```

여기에서 관찰할 것은 모든 주소가 비슷한 메모리 영역에 있다는 것입니다. 실제 변수 i, d, s는 모두 지역에서 생성된 지역변수이고, 지역변수는 STACK에 생성되므로 이렇게 나오는 것입니다.

출력된 주소를 메모리 그림으로 나타내 보겠습니다. 메모리의 한 칸을 4바이트라고 생각하면 string은 6칸이고 그 아래 주소에 int가 1칸 그리고 double이 2칸을 차지하게 됩니다.



STACK은 쓰레드당 한 개씩 생성되며 비주얼 스튜디오에서의 기본 크기는 1MB입니다. 지역 변수가 순서대로 정의되었으므로 STACK에 순서대로 생성될 것이라고 예상해 볼 수도 있겠지만 관찰결과와는 그렇지 않네요. STACK에 변수를 push하면 높은 번지에서 낮은 번지로 push해 갑니다. 컴파일러가 지역변수를 push해서 만들 필요는 없었을 것이라고 예상해 볼 수 있습니다. 실제로 컴파일러는 속도나 메모리의 최적화에 적합하도록 메모리를 구성하기 때문에 우리가 그것이 어떻게 된 것인지 관찰하려면 컴파일러가 생성한 어셈블리어를 봐야합니다. 본다고 해서 항상 동일한 어셈블리어를 만들어내지 않기 때문에 의미는 없습니다. 그렇지만 STACK에 변수를 사용하는 순서를 지키는 경우가 하나 있는데 그것은 함수의 인자전달입니다. 함수의 인자전달 방식을 **calling convention**이라고 합니다. 다음 코드를 입력하고 실행해 보세요.

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

void f( int, double, string );

void reverse_f( string, double, int );

int main( )
{
    int i { 1 };
}
```

```

double d { 2 };
string s { "3" };

f( i, d, s );           // i, d, s로 호출

reverse_f( s, d, i );   // 순서를 거꾸로 호출

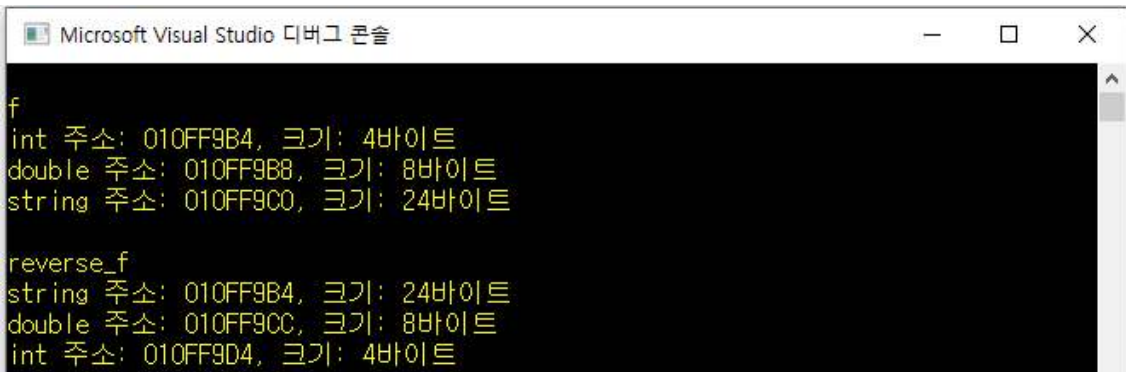
}

void f( int i, double d, string s )
{
    cout << endl << "f" << endl;
    cout << "int 주소: " << &i << ", 크기: " << sizeof( i ) << "바이트" << endl;
    cout << "double 주소: " << &d << ", 크기: " << sizeof( d ) << "바이트" << endl;
    cout << "string 주소: " << &s << ", 크기: " << sizeof( s ) << "바이트" << endl;
}

void reverse_f( string s, double d, int i )
{
    cout << endl << "reverse_f" << endl;
    cout << "string 주소: " << &s << ", 크기: " << sizeof( s ) << "바이트" << endl;
    cout << "double 주소: " << &d << ", 크기: " << sizeof( d ) << "바이트" << endl;
    cout << "int 주소: " << &i << ", 크기: " << sizeof( i ) << "바이트" << endl;
}

```

인자의 전달 순서를 달리하여 함수 f와 reverse_f를 호출하였습니다. C++ 언어에서 인자는 제일 마지막 인자가 가장 처음으로 STACK에 push 됩니다. 함수 f를 호출하면 s, d, i 순서로 push 됩니다. 주소를 관찰해보면 s의 주소가 가장 큼을 알 수 있습니다.



```

Microsoft Visual Studio 디버그 콘솔

f
int 주소: 010FF9B4, 크기: 4바이트
double 주소: 010FF9B8, 크기: 8바이트
string 주소: 010FF9C0, 크기: 24바이트

reverse_f
string 주소: 010FF9B4, 크기: 24바이트
double 주소: 010FF9CC, 크기: 8바이트
int 주소: 010FF9D4, 크기: 4바이트

```

함수 reverse_f는 인자의 순서가 거꾸로 입니다. 따라서 int가 가장 먼저 push되며 주소값이 제일 큼니다. 모두 순서대로 전달됨을 알 수 있습니다. 재미있게도 이 코드를 실행했는데 순서가 이와 같이 나오지 않을 수도 있습니다. 그런 학생은 손들어 주세요.

자유 메모리 공간에 할당한 메모리를 관찰해 보겠습니다. 다음 코드를 입력하고 실행시켜 주세요.

```

#include <iostream>
using namespace std;

int main( )
{
    int* p;

    p = new int[3] { 1, 2, 3 };

    cout << "HEAP에 할당한 int의 값" << endl;
    for ( int i = 0; i < 3; ++i )
        cout << p[i] << endl;

    cout << endl;
    cout << "p의 주소: " << &p << endl;
    cout << "p의 크기: " << sizeof( p ) << "바이트" << endl;

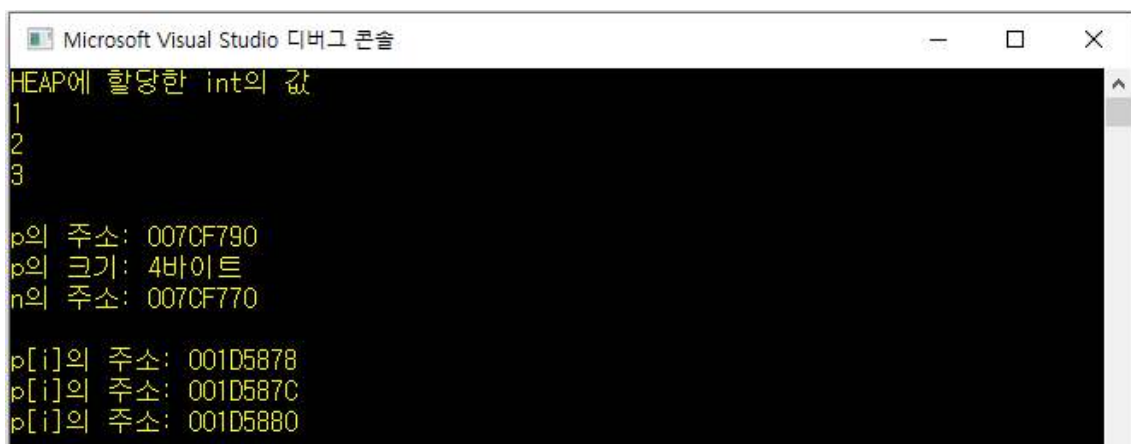
    // 비교를 위한 int
    int n { 2020 };
    cout << "n의 주소: " << &n << endl;

    cout << endl;
    for ( int i = 0; i < 3; ++i )
        cout << "p[i]의 주소: " << &p[i] << endl;

    delete[] p;
}

```

메모리를 비교 관찰할 수 있게 int 변수를 추가하였습니다. 결과를 천천히 관찰해 주세요. 프로그램 코드와 실행화면을 번갈아 살펴보며 메모리를 그려보세요. 나와 같이 그려볼 수 있었나요?



Microsoft Visual Studio 디버그 콘솔

```

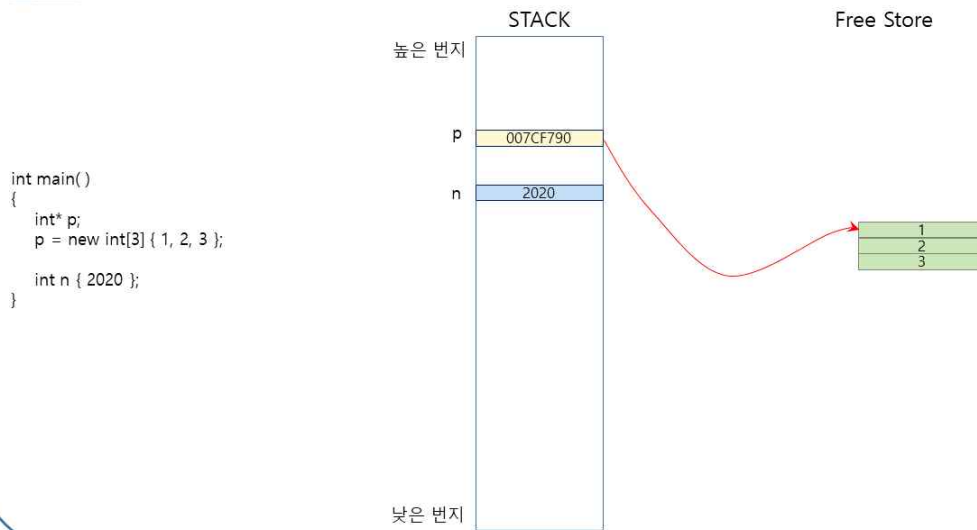
HEAP에 할당한 int의 값
1
2
3

p의 주소: 007CF790
p의 크기: 4바이트
n의 주소: 007CF770

p[i]의 주소: 001D5878
p[i]의 주소: 001D587C
p[i]의 주소: 001D5880

```

HEAP에 할당한 메모리



이 그림을 바로 이해하지 못하는 학생도 꽤 있으리라 생각합니다. 지금까지 경험상 그렇습니다. 이거 어려운거 아닙니다. 코드 보고 출력 보고 그림 보고 같이 생각해 보는 겁니다. 그리고 나서 내가 이해한 것을 다른 사람에게 설명해 보세요. 그런 후 다른 사람 설명을 들어 보세요. 훨씬 나을 겁니다. 잘 모르겠으면 꼭 그렇게 하세요. 다음에 하련다 하지 말고 지금 더 시간 들여 공부해보세요. 그러면 다음에는 지금보다 조금 더 알 수 있을 겁니다. 공부가 뭐 그런겁니다. 반복이죠 뭐.

이제 지난 시간 소개한 String을 살펴보겠습니다. 다음 코드를 입력하고 실행시켜 주세요.

```

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

int main( )
{
    String s { "힘내라 대한민국!" };

    cout << "s의 크기: " << sizeof( s ) << "바이트" << endl;
    cout << "s의 번지: " << &s << endl;

    int n { 2020 };
    cout << "n의 번지: " << &n << endl;

    cout << endl;
    cout << "s가 할당한 메모리의 번지: " << static_cast<void*>(&s[0]) << endl;
    int* p = new int { 777 }; // HEAP 비교를 위한 번지
}

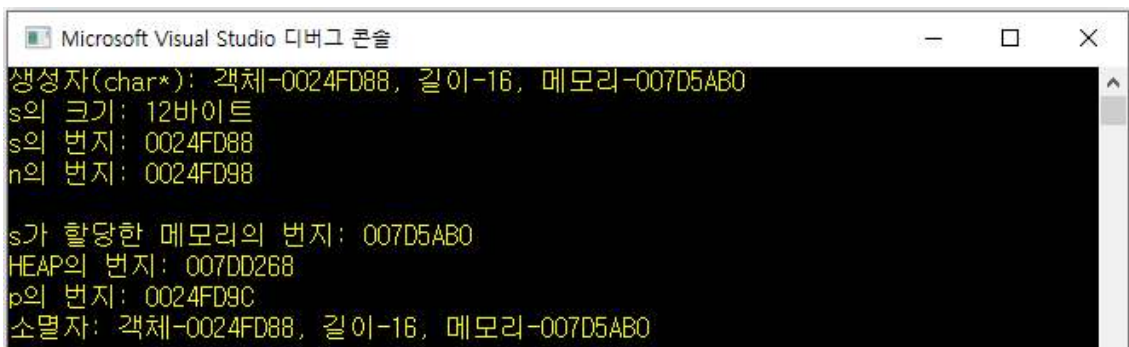
```

```

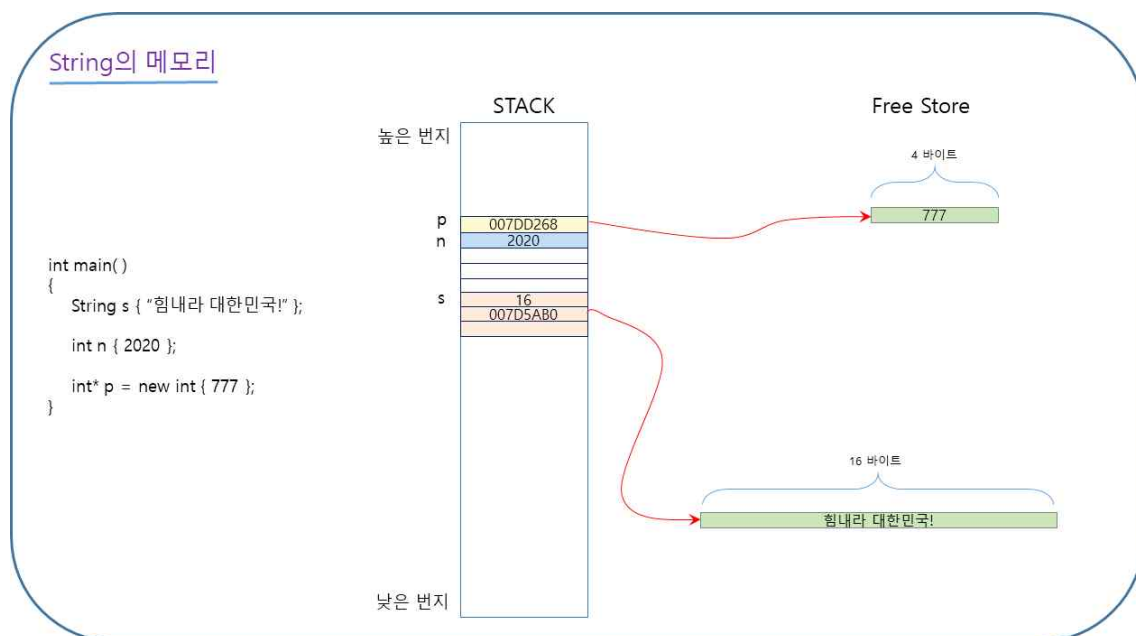
    cout << "HEAP의 번지: " << p << endl;
    cout << "p의 번지: " << &p << endl;
}

```

관찰메시지가 있어 생성자와 소멸자에서 자기 메모리를 알 수 있네요. 객체 s 자신의 메모리는 STACK에, 객체 s가 확보한 메모리는 자유 메모리 공간에 있음을 알 수 있습니다. 변수 n과 p는 순전히 비교를 위해 덧붙인 것입니다.



이제 이 실행화면을 살펴보고 아래와 같이 메모리를 그림 그려볼 수 있음을 알겠나요? Free Store는 말 그대로 자유롭게 때문에 할당받은 메모리 공간을 따로 그리면 되겠습니다. 그런데 한 번 할당받은 공간은 모두 연속된 메모리 공간이라는 것을 잊지 마세요. 또한 자유 메모리 공간은 꼭 시작번지를 저장해 놓아야 한다는 것도 잊으면 안 됩니다. 기억하고 있지 않으면 되돌려 줄 방법이 없기 때문이죠.



[실습] 다음 프로그램에서 STACK과 HEAP에 생성되는 객체를 그림으로 그려라. (10분)

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

int main( )
{
    String* p = new String{ "힘내라 대한민국!" };
}
```

자신 있게 그림 그려 볼 수 있었으리라 기대합니다. 이제 vector의 메모리를 설명해도 좋겠습니다. 벡터는 클래스 템플릿입니다. 먼저 원소가 하나도 없어 비어있는 벡터를 하나 만들어 보겠습니다.

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

int main( )
{
    vector<int> v;
    int n;

    cout << "vector v의 주소: " << &v << endl;
    cout << "비교를 위한 int의 주소: " << &n << endl;

    cout << "vector v의 크기: " << sizeof( vector<int> ) << "바이트" << endl;
}
```

비어있는 vector라도 무엇을 담을 것인지를 지정해 줘야 합니다. 템플릿이니까요. 실행 결과입니다.



```
Microsoft Visual Studio 디버그 콘솔
vector v의 주소: 0021F724
비교를 위한 int의 주소: 0021F734
vector v의 크기: 12바이트
```

이제 메모리를 출력하는 화면 보는 것이 조금 익숙해졌죠? 그림 그릴 수 있겠죠? 지금 v는 어디에 있나요? 크기는 12바이트를 차지하고 있네요. 자! 여기서 질문입니다. vector<int> v가 int를 3개 저장하도록 하면 v의 크기는 얼마일까요? 프로그램을 이렇게 바꿔 봅시다.

```

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

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

    cout << "vector v의 주소: " << &v << endl;
    cout << "비교를 위한 int의 주소: " << &n << endl;

    cout << "vector v의 크기: " << sizeof( vector<int> ) << "바이트" << endl;
}

```

뭘 묻고 있는 건지 알겠죠? v의 크기는 얼마라구요? 아무도 대답이 없네요. 저기 뒤에서 웹툰 보고 있는 학생! 네. 맞아요. 대답해 보세요. 얼른 실행해 보면 나오잖아요. 그렇죠? v의 크기는 변하지 않습니다. 그대로죠. 놀고 있는지 알았더니 잘 따라 오고 있었네요. 오해해서 미안합니다.

그렇습니다. vector는 동적배열이라고 합니다. 실행 시에 메모리를 마음대로 바꿀 수 있는 건 메모리를 HEAP에 확보한다는 말입니다. 그럼 그렇게 하기위한 vector<int> v의 책임은 무엇입니까? 그렇습니다. 제일 먼저 v가 기억하고 있어야 할 것은 확보한 메모리의 주소입니다. 위의 프로그램에서는 int 3개를 저장하려고 HEAP(길게 쓰려니 그냥 HEAP이라고 하겠습니다. 사실 섞어 써도 문제없어요)에 할당한 공간의 주소를 저장해야 합니다. 그리고 무엇을 더 저장하면 좋을까요? 네. 맞아요. 원소의 갯수입니다. 현재 3개이니 이걸 저장해야 합니다.

하나가 더 있는 데 뭘지 아시겠습니까? vector는 필요하면 공간을 더 늘릴 수 있잖아요. 지난 시간에 원소를 추가하며 관찰했던 프로그램 기억나죠? 그래요. vector는 원소가 다 있는지 또는 그렇지 않은지 비교해 볼 수 있는 숫자 하나를 더 저장하고 있어야 합니다. 그래서 원소를 새로 추가할 때마다 공간이 있는지 없는지 검사해서 공간이 부족하면 새 공간을 확보할 수 있는 것이거든요. 정리합니다. 벡터가 기억하고 있어야 할 내용!

- 현재 원소의 갯수 (size)
- 원소가 저장되어 있는 주소 (data)
- 재할당하지 않고 저장할 수 있는 갯수 (capacity)

각각의 값을 저장하는데 int면 충분하겠죠? 그래서 vector<int> v의 크기는 12바이트가 되는 것입니다. 사실 vector가 무엇을 저장하든지 12바이트면 됩니다. 프로그램 따라 해 주세요.

```

#include <iostream>
#include <vector>

```

```
using namespace std;

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

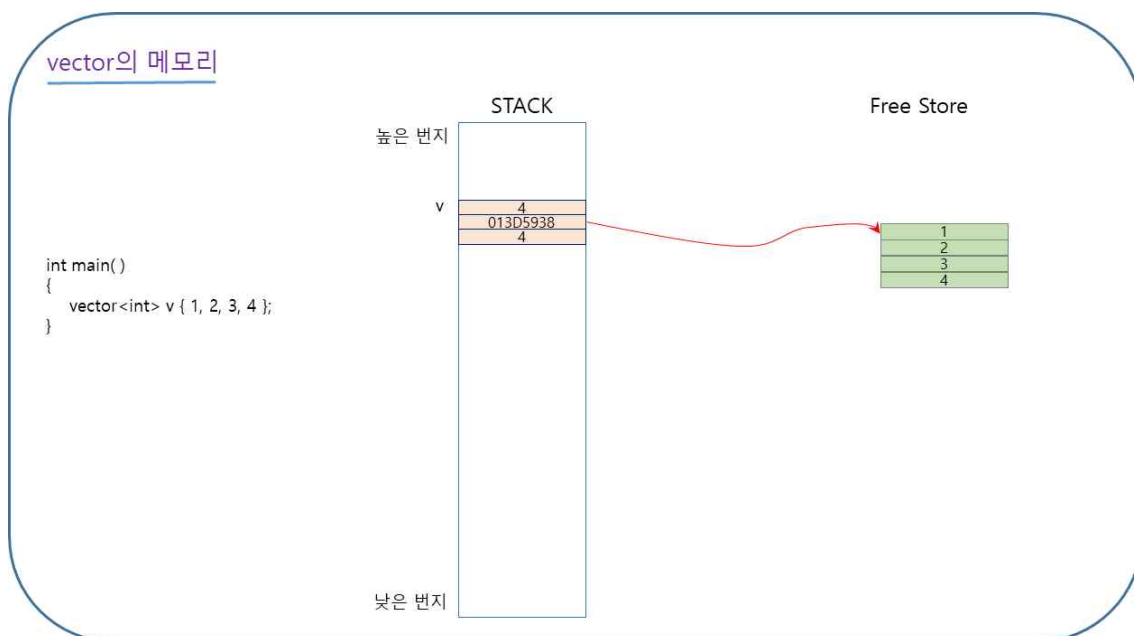
    cout << "vector v의 주소: " << &v << endl;

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

실행 결과입니다. 네. 손 든 학생. 뭐죠? 주소가 다르게 나온다고요? 뭔 그런 농담을.^^.

```
Microsoft Visual Studio 디버그 콘솔
vector v의 주소: 010FFEA4
원소의 갯수: 4
원소가 저장된 메모리: 013D5938
재할당하지 않고 담을 수 있는 갯수: 4
```

메모리 그림입니다. 마음이 푸근할 겁니다. 어렵지 않으니깐요.



사실 vector에서 관리하는 3개 값의 순서는 내 마음대로 그린 겁니다. 중요 사항이 아니니까요. 중요한 것은 바로 지금입니다. 원소 1개를 추가하는 거죠. int 777을 추가해 보겠습니다.

```

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

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

    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;

    // 777을 추가 합니다.

    v.push_back( 777 );

    cout << endl << "----- 원소 추가 후 -----" << endl;

    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;
}

```

실행결과 화면입니다.

```

Microsoft Visual Studio 디버그 콘솔
vector v의 주소: 010FFEA4
원소의 갯수: 4
원소가 저장된 메모리: 013D5938
재할당하지 않고 담을 수 있는 갯수: 4

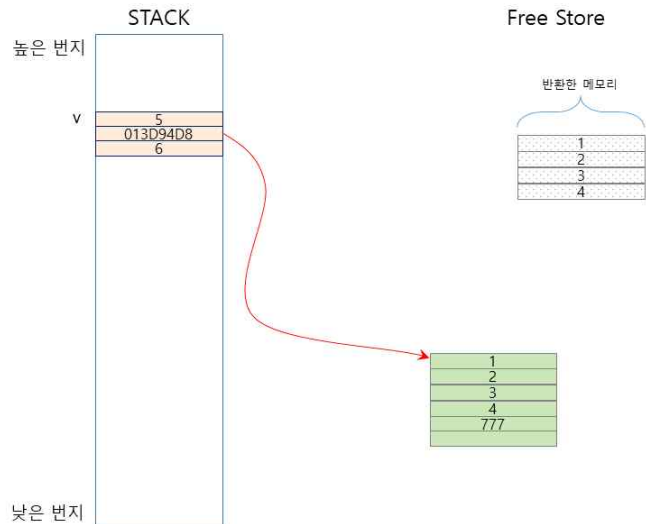
----- 원소 추가 후 -----
vector v의 주소: 010FFEA4
원소의 갯수: 5
원소가 저장된 메모리: 013D94D8
재할당하지 않고 담을 수 있는 갯수: 6

```

메모리가 늘어났습니다. 그런데 같은 번지가 아닙니다. 그림으로 그릴 수 있겠죠. 문제 없겠죠? 나도 그려 보겠습니다.

int 추가 후 vector의 메모리

```
int main()
{
    vector<int> v { 1, 2, 3, 4 };
    v.push_back( 777 );
}
```



어떻습니까? vector의 메모리는 늘어나는 것이 아니죠? 다른 곳으로 이사 가는 것입니다. vector의 뒤쪽에 원소를 추가하는 push_back이나 emplace_back 동작은 $O(1)$ 복잡도를 갖는 동작입니다만 재할당이 일어날 때는 그렇지 않습니다. new와 delete를 호출하고 모든 원소를 복사해야 합니다. 실제로 재할당 시 push_back은 아주 많은 비용이 드는 동작이 됩니다. 가능하다면 그럴 일이 없도록 신경 써서 코딩하면 되겠죠?

재할당이 안되도록 메모리를 많이 예약해 놓으면 어떨까요? 얼마나 많이 예약하면 좋을까요? 그에 따른 비용은 어떻게 되는 건가요? **메모리를 잘 이해한다면 vector를 제대로 이용할 수 있습니다.** 지금까지 많이 공부했는데 진짜는 따로 있습니다. 눈치채셨죠? 그럼요. int를 저장하려고 vector를 사용하지는 않는단니까요? vector<String> v를 갖고 놀아야죠. 어때요. 정말 기대되죠? 아니 힘 하나도 안 든단니까요. 지난 밤 10시쯤 시작했는데 그림 그리고 코딩하고 타자 열심히 하니까 벌써 해 뜨네요. 오늘은 여기까지 하죠 뭐! 다음 강의 시간에 계속해서 하겠습니다. 여러분도 재미있다고 밤새 하면 안되요. 건강 챙기며 하세요. 과제는 다음 시간에 생각해 보겠습니다. 안녕!