

5-2. Offset

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

이번 시간에는

- 새로운 개념, **offset**에 대해 소개합니다.
- **Offset** 개념을 보다 능동적으로 사용할 수 있는 요소들을 구경해 봅니다.
 - C++로 넘어가기 위해 구경해 둘 만한 내용이에요
 - 이전에 나왔던 다양한 개념들과 연결해 가며 납득할 수 있도록 구성해 두었어요
 - 그럭저럭 복습이 될 수 있을 듯!

이번 시간에는

- **Index와 offset**

- 이 둘은 결국은(runtime에는) 동일하다 볼 수 있지만 약간 뉘앙스가 달라요

- C에서 새로운 단어를 정하는 다양한 방법

- 새 Data **이름**, Code **이름**을 정할 수 있어요
- 이름보다는 중립적인, 새 '단어'를 새로 정할 수 있어요
- 새 '**형식 이름**'도 정할 수 있어요

- **구조체**

- '한 칸의 내용'을 직접 **정의**하고, **offset** 개념을 써서 각 부분을 사용할 수 있어요

이번 시간에는

- 바로 시작해 봅시다.

Index

- Python의 `list`나 C의 **배열** 등을 다룰 때 사용해 보았을 **index** 개념은, 현실 속 순서 개념을 다루기 위한 친구입니다.
 - '몇 번째'에 해당하는 개념

Index

- Python의 `list`나 C의 **배열** 등을 다룰 때 사용해 보았을 **index** 개념은, 현실 속 순서 개념을 다루기 위한 친구입니다.
 - '몇 번째'에 해당하는 개념
- Python에서는 다양한 '시퀀스 **형식**'이 마련되어 있습니다.
 - `str`, `list`, `tuple`

Index

- C에서는 **index** 개념에 완벽히 부합하는 **형식**은 아쉽게도 존재하지 않습니다.

Index

- C에서는 **index** 개념에 완벽히 부합하는 **형식**은 아쉽게도 존재하지 않습니다.
 - 단순히 여러 칸을 **정의**하기(받기)위해 **선언**해 사용하는 **배열**
 - `int arr[2];` 처럼 **선언**해 놓고 `arr[-1]` 이나 `double *p_rate = arr;` 처럼 다루어도 돼요
 - 도대체 무슨 의도를 가지면 이런 표현이 당위성을 갖게 되는지는 잘 모르겠지만 아무튼 그래요

Index

- C에서는 **index** 개념에 완벽히 부합하는 **형식**은 아쉽게도 존재하지 않습니다.
 - 단순히 여러 칸을 **정의**하기(받기)위해 **선언**해 사용하는 **배열**
 - `int arr[2];` 처럼 **선언**해 놓고 `arr[-1]` 이나 `double *p_rate = arr;` 처럼 다루어도 돼요
 - 도대체 무슨 의도를 가지면 이런 표현이 당위성을 갖게 되는지는 잘 모르겠지만 아무튼 그래요
 - 한 칸에 대한 **포인터 값**을 담기 위해 **선언**해 사용하는 **포인터 변수**
 - 사실 위에 있는 **수식** `arr[-1]`은 **수식** `*(arr - 1)`과 동일했어요
 - **수식** `arr`을 먼저 **계산**한 **결과값**을 '기준 위치'로 두고 거기서 음수 방향으로 한 칸 옆

Index

- C에서는 **index** 개념에 완벽히 부합하는 **형식**은 아쉽게도 존재하지 않습니다.
 - 단순히 여러 칸을 **정의**하기(받기)위해 **선언**해 사용하는 **배열**
 - `int arr[2];` 처럼 **선언**해 놓고 `arr[-1]` 이나 `double *p_rate = arr;` 처럼 다루어도 돼요
 - 도대체 무슨 의도를 가지면 이런 표현이 당위성을 갖게 되는지는 잘 모르겠지만 아무튼 그래요
 - 한 칸에 대한 **포인터 값**을 담기 위해 **선언**해 사용하는 **포인터 변수**
 - 사실 위에 있는 수식 `arr[-1]`은 수식 `*(arr - 1)`과 동일했어요
 - 수식 `arr`을 먼저 **계산**한 결과값을 '기준 위치'로 두고 거기서 음수 방향으로 한 칸 옆
- 옹... 방금 등장한, '기준 위치 대비 얼마만큼 옆'이라는 표현이 새 단어 **offset**과 매우 가까운 표현이에요!

Index와 offset

- **Index**는 여러분의 의도를 반영하는 단어입니다.
 - 문자열 표현 "HELP"의 경우 명백히 이 나열 자체가 (현실의) 영어 단어를 상징하므로 수식 "HELP"[0]의 0은 '단어의 첫 글자'를 얻겠다는 의도를 반영한다 볼 수 있습니다

Index와 offset

- **Index**는 여러분의 의도를 반영하는 단어입니다.
 - 문자열 표현 "HELP"의 경우 명백히 이 나열 자체가 (현실의) 영어 단어를 상징하므로 수식 "HELP"[0]의 0은 '단어의 첫 글자'를 얻겠다는 의도를 반영한다 볼 수 있습니다
- 이에 비하면 **offset**은 의도 측면에서는 중립적인 편이고, 그 대신 '기준 위치'와의 연계에 큰 무게를 두는 단어입니다.
 - 조금 납득하기 어려울 수 있으니 간단한 코드를 직접 적어 봅시다

Index와 offset

- 적당히 알아서 켜고, .c 파일에 아래 내용을 적어 봅시다:

```
#include <stdio.h>

int main()
{
    int arr[3];

    // 복붙 쓰세요
    int offset_0 = &arr[0] - arr;
    int offset_1 = &arr[1] - arr;
    int offset_2 = &arr[2] - arr;

    printf("&arr[0] - arr: %d\n"
           "&arr[1] - arr: %d\n"
           "&arr[2] - arr: %d\n", offset_0, offset_1, offset_2);
}
```

Index와 offset

- 적당히 알아서 켜고, .c 파일에 아래 내용을 적어 봅시다:

```
#include <stdio.h>
```

```
int main()  
{
```

```
    int arr[3];
```

```
    // 복붙 쓰세요
```

```
    int offset_0 = &arr[0] - arr;
```

```
    int offset_1 = &arr[1] - arr;
```

```
    int offset_2 = &arr[2] - arr;
```

```
    printf("&arr[0] - arr: %d\n"  
           "&arr[1] - arr: %d\n"  
           "&arr[2] - arr: %d\n", offset_0, offset_1, offset_2);
```

```
}
```

다 적었으면 실행해 봐도 좋아요.

실행하기 전에,
arr이 int 배열 이름을 감안해서
출력 결과가 어떻게 나올 지 잠시 예상해 봐요!

Index와 offset

- 다 적었다면 Ctrl + F5를 눌러 출력 결과를 확인해 봅시다.

Index와 offset

- 다 적었다면 Ctrl + F5를 눌러 출력 결과를 확인해 봅시다.
 - 오용... 혹시 0, 4, 8 뜨는 건 아닌가 생각했었는데 0, 1, 2가 뜨고 있습니다






Index와 offset

- 약간만 더 살펴 볼까요?
이번에는 F10을 눌러 디버깅을 시작하고,
조사식 탭에 아래 수식들을 적어 봅시다:

조사식 1	
이름	
▶	<code>&arr[1]</code>
▶	<code>arr + 1</code>
	<code>&arr[1] - arr</code>
	<code>(arr + 1) - arr</code>
▶	<code>&arr[(arr + 1) - arr]</code>

Index와 offset

- 뭔가 규칙적이면서도 그렇지 않은 결과들이 나오고 있습니다...

조사식 1			
이름	값	형식	
▶  &arr[1]	0x00fdfa00 {924531}	int *	
▶  arr + 1	0x00fdfa00 {924531}	int *	
 &arr[1] - arr	1	int	
 (arr + 1) - arr	1	int	
▶  &arr[(arr + 1) - arr]	0x00fdfa00 {924531}	int *	

Index와 offset

- 방금 나온 수식들을 옮겨 적어 보았어요!

$$\&arr[1] == arr + 1$$
$$\&arr[1] - arr == 1$$

Index와 offset

- 방금 나온 수식들을 옮겨 적어 보았어요!

$$\&arr[1] == arr + 1$$

$$\&arr[1] - arr == 1$$

수학적인 관점에서 보면,
위 등식의 양변에서 arr을 빼면
아래 등식이 나와요!

Index와 offset

- 방금 나온 수식들을 옮겨 적어 보았어요!

$$\begin{array}{l} \&arr[1] \\ (\text{int } *) \end{array} \quad \begin{array}{l} == \text{arr} + 1 \\ == (\text{int } *) + (\text{int}) \end{array}$$
$$\begin{array}{l} \&arr[1] - \text{arr} \\ (\text{int } *) - (\text{int } *) \end{array} \quad \begin{array}{l} == 1 \\ == (\text{int}) \end{array}$$

각 자리의 **형식**을 생각해 보면
이런 느낌으로 **계산**이 진행되고 있음!

Index와 offset

- 요약하면...
 - **포인터 값 + offset 값 → offset 적용된 포인터 값**
 - **포인터 값 - 포인터 값 → offset 값**(음수거나 0일 수도 있음)

...정도로 보면 정확해요!

Index와 offset

- 요약하면...
 - 포인터 값 + offset 값 \rightarrow offset 적용된 포인터 값
 - 포인터 값 - 포인터 값 \rightarrow offset 값(음수거나 0일 수도 있음)
- Q. 포인터 값 + 포인터 값 계산을 하면 뭐가 나올까요?

Index와 offset

- 요약하면...
 - **포인터 값 + offset 값 → offset 적용된 포인터 값**
 - **포인터 값 - 포인터 값 → offset 값**(음수거나 0일 수도 있음)
- Q. **포인터 값 + 포인터 값** 계산을 하면 뭐가 나올까요?
 - 뭔가 느낌이 찝찝하긴 하지만, 직접 확인해 봅시다

Index와 offset

- 대충 이런 느낌으로 테스트하면 될 것 같아요!

```
#include <stdio.h>

int main()
{
    int *p1, *p2;

    p1 - p2;
    p1 + p2;
}
```

Index와 offset

- 의외로, 즉시 빨간 줄이 그어지는 것을 볼 수 있습니다.
 - p2의 **형식** 때문에 문제가 생긴 것이고, 따라서 **선언만** 봐도 VS가 오류를 짚을 수 있어요!

```
int main()
{
    int *p1, *p2;

    p1 - p2;
    p1 + p2;

    return
}
```

int *p2

식에 정수 계열 형식이 있어야 합니다.

Index와 offset

- 흠... 왜 빨셈은 잘만 인정되면서 덧셈은 거부하고 있는 걸까요?

Index와 offset

- 흠... 왜 빨셈은 잘만 인정되면서 덧셈은 거부하고 있는 걸까요?
 - 어떤 수업을 친구랑 듣는데 중간고사 등수가 나는 7등, 개는 12등이라면,
개 등수 - 내 등수 는 어느 정도 의미를 갖는 반면(내가 더 잘 봤다는 사실을 알 수 있음)
개 등수 + 내 등수 인 19는 별 의미 없는 것 같아요

Index와 offset

- 흠... 왜 빨셈은 잘만 인정되면서 덧셈은 거부하고 있는 걸까요?
 - 어떤 수업을 친구랑 듣는데 중간고사 등수가 나는 7등, 개는 12등이라면,
개 등수 - 내 등수 는 어느 정도 의미를 갖는 반면(내가 더 잘 봤다는 사실을 알 수 있음)
개 등수 + 내 등수 인 19는 별 의미 없는 것 같아요
- 맞아요. 이 부분이 '단순 위치 값'과 '**offset** 값'의 기본적 차이라 볼 수 있어요

Index와 offset

- 흠... 왜 뺄셈은 잘만 인정되면서 덧셈은 거부하고 있는 걸까요?
 - 어떤 수업을 친구랑 듣는데 중간고사 등수가 나는 7등, 개는 12등이라면,
개 등수 - 내 등수 는 어느 정도 의미를 갖는 반면(내가 더 잘 봤다는 사실을 알 수 있음)
개 등수 + 내 등수 인 19는 별 의미 없는 것 같아요
- 맞아요. 이 부분이 '단순 위치 값'과 '**offset** 값'의 기본적 차이라 볼 수 있어요
 - 우리가 쓰는 모든 **위치 값(포인터 값)**들은 전부 동일한 **위치를 '원점'으로 가져요**
 - 메모리의 0x00000000 또는 0x0000000000000000 자리를 원점으로 가져요
 - 근데 그 원점 자체, 또는, '원점과의 거리'는 그리 큰 의미를 가지고 있지 않아요
 - 수학에서 벡터를 다루고 있다면 원점이 '힘의 원점'과 같은 느낌으로 작용해서 덧셈이 의미를 갖지요. 그렇지 않을 때는 '두 점의 x좌표의 합'은 구해 본 기억이 거의 없을 거예요!
 - C에서는 메모리의 0x0 자리를 **위치 값**으로 갖는 이름은 일반적으로 존재하지 않긴 해요. 그런데 존재한다 해도, 이로 인해 '원점과의 거리' 자체에 의미가 부여되지는 않아요!

Offset

- 단어 **offset**을 사용하는 것에는,
'기준 위치'나 '기준 위치와의 거리'에 의미를 두겠다는 의도가 들어 있어요.
 - 시험 점수가 순서대로 나열되어 있을 때,
'0등이 누군지 알기'는 index 개념이 강한 목표가 되고,
'내가 이겼는지 친구가 이겼는지 알기'는 **offset** 개념이 강한 목표가 될 거예요
 - 반면 '내가 몇 등인지 알기'는 나름 독립적인 목표라고 볼 수 있을 듯?
 - 내가 0등인지 알고 싶었다면 앞 목표와,
친구와의 등수 비교를 위한 기준 위치로 내 등수를 쓰려는 거라면 뒷 목표와 연계돼요

Offset

- 단어 **offset**을 사용하는 것에는,
'기준 위치'나 '기준 위치와의 거리'에 의미를 두겠다는 의도가 들어 있어요.
- int 배열 arr을 사용할 때,
수식 &arr[1]과 같은 표현에서 arr은 기준 위치, 1은 **offset**이 돼요.
 - 기준 위치는 컴파일러가 정했고, **offset**은 그렇지 않았어요

Offset

- 여기서 중요한 점!
 - (중요 1) 아무튼, **수식** `&arr[1]`은
명백히 '메모리 위의' int 한 칸에 대한 **포인터 값**을 얻기 위해 적을 수 있어요

Offset

- 여기서 중요한 점!
 - (중요 1) 아무튼, **수식** `&arr[1]`은 명백히 '메모리 위의' int 한 칸에 대한 **포인터 값**을 얻기 위해 적을 수 있어요
 - (중요 2) 이렇다 보니, **형식**은 같지만 서로 다른 두 **배열** `arr1`, `arr2`가 있을 때, 동일한 **offset 값**(나오는 **수식**)을 사용한 **수식** `&arr1[1]`과 **수식** `&arr2[1]`가 (**offset**이 같음에도) **계산** 결과가 서로 다르게 나올 것임이 보장돼요

Offset

- 여기서 중요한 점!
 - (중요 1) 아무튼, **수식** `&arr[1]`은 명백히 '메모리 위의' `int` 한 칸에 대한 **포인터 값**을 얻기 위해 적을 수 있어요
 - (중요 2) 이렇다 보니, **형식**은 같지만 서로 다른 두 **배열** `arr1`, `arr2`가 있을 때, 동일한 **offset 값**(나오는 **수식**)을 사용한 **수식** `&arr1[1]`과 **수식** `&arr2[1]`가 (**offset**이 같음에도) **계산** 결과가 서로 다르게 나올 것임이 보장돼요
 - (중요 중요) 그렇기 때문에, 기준 **위치**를 의도에 따라 그 때 그 때 다르게 부여함으로써 동일한 **offset 값**을 사용하더라도 서로 다른 칸을 그 때 그 때 특정해 쓸 수 있게 돼요!

넘어가는 슬라이드

- 음. 뭔가 어려운 듯 하면서 '뭐 어차피 숫자네' 할 수 있는 내용이었습니다.
 - 사실 보통 C/C++에선 [] **연산자** 대괄호 안에 내가 적는 것들은 그냥 **index**라 불러요
 - '거리 개념'을 다루지 않을 땐 그냥 **index**라 부르면 돼요
 - 굳이 둘을 구분하고 싶을 때만 방금 것처럼 선을 그어 생각하면 돼요
- 이제 이 **offset** 개념을 조금 더 적극적으로 사용해 볼게요.

예시 구경하기

- CSP_5_2_base.c를 VS에 탑재해 봅시다.
 - 탑재한 다음, 코드의 내용을 살짝 구경해 보세요
 - 아까 5-1에서 적어 본 것과 꽤 비슷할 거예요. 이번엔 능력치를 세 개만 써요

예시 구경하기

- 네 캐릭터의 능력치 정보가 들어 있고,
Duel()을 재차 호출함으로써 네 캐릭터의 등수를 매겨 담은 프로그램입니다.
 - 지금은 '무조건 left번 캐릭터가 이김' 규칙이 들어 있기는 해요
- 약간 시간을 들여서, 두 캐릭터의 세 능력치를 각각 비교하여 대결을 진행하도록
Duel() 내용물 **문장**들을 구성해 봅시다

예시 구경하기

- 음... 이 코드는 뭔가 이상합니다.
 - '캐릭터 하나에 대한 정보'가 서로 다른 **배열**들에 각각 흩어져 있어요
 - 그냥 아까처럼 캐릭터별 **배열 선언(정의)**을 해 두면
한 캐릭터에 대한 능력치 **값**들이 한 데 모여 있었을 듯?

예시 구경하기

- 음... 이 코드는 뭔가 이상합니다.
 - '캐릭터 하나에 대한 정보'가 서로 다른 **배열**들에 각각 흩어져 있어요
 - 그냥 아까처럼 캐릭터별 **배열 선언(정의)**을 해 두면
한 캐릭터에 대한 능력치 **값**들이 한 데 모여 있었을 듯?
 - 지금은 캐릭터가 네 명이지만, 나중에 캐릭터가 100명 넘게 있다거나 할 때는
만약 각 캐릭터 정보를 서로 다른 **배열 선언**을 적어 가며 각각 담았다면
서로 다른 **배열 이름** 100여 개가 생겨버리게 될 거예요
 - 그러느니 그냥 능력치 **이름** 네 개를 써서 **배열** 네 개를 만드는 게 좀 더 편할 듯 해요

1차 개선: 캐릭터 정보를 배열 하나에

- 생각해 보니,
'캐릭터 하나에 대한 정보'를 **배열**로 표현한다면,
캐릭터 여럿에 대한 정보 또한 '그들의 **배열**'로 다룰 수 있을 듯 합니다.

1차 개선: 캐릭터 정보를 배열 하나에

- 생각해 보니,
'캐릭터 하나에 대한 정보'를 **배열**로 표현한다면,
캐릭터 여럿에 대한 정보 또한 '그들의 **배열**'로 다룰 수 있을 듯 합니다.
 - 지금 코드에서는 `int stats[4][3];`과 같이 **선언**하면 될 듯 해요
 - `int stats[3][4];` 아님!
 - **선언**을 추가하고, 약간 고민해 보면서 이 **선언**에 대한 `initializer`를 적어 봅시다
 - 큰 중괄호 안의 각 부분에 뭐가 들어가야 하는지 생각해 보면 될 듯 해요
 - 다 적었다면 이제 `stats`를 사용하도록 `Duel()`의 내용을 살짝 변경해 봅시다

1차 개선: 캐릭터 정보를 배열 하나에

- 이제 Duel()에 적는 실제 수식이 [] 연산자 두 개를 쓴 형태가 되었습니다.
 - 선언을 그리 적어 봤으니 당연해요

1차 개선: 캐릭터 정보를 배열 하나에

- 이제 Duel()에 적는 실제 **수식**이 [] **연산자** 두 개를 쓴 형태가 되었습니다.
 - 선언을 그리 적어 봤으니 당연해요
- **수식** stats[idx_left][0]을 조금 더 자세히 본다면...
 - **수식** stats[idx_left]까지는 '이번 캐릭터 정보의 기본 **위치**'를 의미하고,
그 뒤에 붙는 0은 '한 캐릭터 정보 안에서의 **offset**'이라 볼 수 있을 것 같아요
 - 일단 두 []가 생긴 건 같아도 실질적 의미가 다르다는 점만 볼 수 있으면 충분해요!

2차 개선: 숫자 대신 단어로

- 달라진 것들 중 하나는,
원래 버전에서는 각 능력치 이름이 **배열** 이름으로서 표현되었지만
이제는 **offset** 숫자(0은 힘 등등)로 표현하기 시작했다는 점입니다.

2차 개선: 숫자 대신 단어로

- 달라진 것들 중 하나는,
원래 버전에서는 각 능력치 이름이 **배열** 이름으로서 표현되었지만
이제는 **offset** 숫자(0은 힘 등등)로 표현하기 시작했다는 점입니다.
 - 선언해 두지 않은 **이름**(예: ints)을 적었을 때는 컴파일러가 오류를 내 주지만
유효하지 않은 **offset** 숫자(예: 3)를 적었을 때는 여간해서는 오류를 내 주지 않아요.
그래서 이러한 표현의 차이는 종종 버그의 원인으로 작용할 수 있어요

2차 개선: 숫자 대신 단어로

- 달라진 것들 중 하나는,
원래 버전에서는 각 능력치 이름이 **배열** 이름으로서 표현되었지만
이제는 **offset** 숫자(0은 힘 등등)로 표현하기 시작했다는 점입니다.
 - 선언해 두지 않은 이름(예: ints)을 적었을 때는 컴파일러가 오류를 내 주지만
유효하지 않은 **offset** 숫자(예: 3)를 적었을 때는 여간해서는 오류를 내 주지 않아요.
그래서 이러한 표현의 차이는 종종 버그의 원인으로 작용할 수 있어요
- 이런 상황에서 쓸 수 있는 C의 기능들 중 하나를 소개해 볼게요!

2차 개선: 숫자 대신 단어로

- 지금 코드의 맨 윗 부분에 아래 내용을 추가해 봅시다:

```
#define STR 0  
#define DEX 1  
#define CON 2
```


2차 개선: 숫자 대신 단어로

- 지금 코드의 맨 윗 부분에 아래 내용을 추가해 봅시다:

```
#define STR 0  
#define DEX 1  
#define CON 2
```

이렇게 적어 두면,
이제 이 파일에 나오는 단어 STR들은
모두 컴파일 도중 0으로 바뀌어 적혀요!

이 성질을 활용해서 Duel()을 좀 더 보기 쉽게 적을 수 있어요!
직접 시도해 봅시다.

2차 개선: 숫자 대신 단어로

- 이제 그럭저럭 '코드만 봐도 읽을 만한' 느낌이 된 것 같습니다.
- `#define`에 대한 자세한 설명은 지금은 생략하고,
여기서 한 단계만 더 '조금 더 쉬운 표현'을 추구해 보도록 합시다.

3차 개선: 복잡한 형식을 별도의 이름으로

- 이제 이 코드에서 가장 복잡한 부분을 꼽자면 아마 stats **선언** 부분을 들 수 있을 것입니다.
 - 의미 자체는 '캐릭터 정보 네 개의 나열'인데, 이게 눈에 확 들어오지 않아요

3차 개선: 복잡한 형식을 별도의 이름으로

- 이제 이 코드에서 가장 복잡한 부분을 꼽자면 아마 stats **선언** 부분을 들 수 있을 것입니다.
 - 의미 자체는 '캐릭터 정보 네 개의 나열'인데, 이게 눈에 확 들어오지 않아요
- 여기도 마찬가지로, C에서 제공하는 specifier 하나를 추가로 도입해서 약간 특이한 **이름**을 **선언**해 쓸 수 있어요!

3차 개선: 복잡한 형식을 별도의 이름으로

- 지금 코드의 맨 윗 부분에 아래 내용을 추가해 봅시다:

```
typedef int stat[3];
```

3차 개선: 복잡한 형식을 별도의 이름으로

- 지금 코드의 맨 윗 부분에 아래 내용을 추가해 봅시다:

```
typedef int stat[3];
```

```
// int stats[4][3] = { ... };
```

```
stat stats[4] = { ... };
```

이제 이런 느낌으로 선언할 수 있게 됐어요!
이번에는 다른 코드는 하나도 안 바뀌어도 됨!

3차 개선: 복잡한 형식을 별도의 이름으로

- 오오... 뭔가 차원 하나를 넘은 기분입니다.
 - 내가 정한 이름이 int 대신 specifier 자리에 적을 수 있는 '**형식 이름**'이 되었어요
 - 실제 typedef specifier의 효과 또한 지금 느낀 것과 동일해요

3차 개선: 복잡한 형식을 별도의 이름으로

- 오오... 뭔가 차원 하나를 넘은 기분입니다.
 - 내가 정한 이름이 int 대신 specifier 자리에 적을 수 있는 '형식 이름'이 되었어요
 - 실제 typedef specifier의 효과 또한 지금 느낀 것과 동일해요
- 복잡하게 얽힌 **선언**이 필요하고, 그 선언의 의미를 명확히 이해하고 있다면,
방금 봤던 방식을 사용해서 다음 번 **선언**의 복잡도를 약간 줄여줄 수 있어요
 - 이렇게 한다 해도 컴파일 도중 컴파일러가 알아서 잘 deduce해 줘요

3차 개선: 복잡한 형식을 별도의 이름으로

- 오오... 뭔가 차원 하나를 넘은 기분입니다.
 - 내가 정한 이름이 int 대신 specifier 자리에 적을 수 있는 '형식 이름'이 되었어요
 - 실제 typedef specifier의 효과 또한 지금 느낀 것과 동일해요
 - 복잡하게 얽힌 **선언**이 필요하고, 그 선언의 의미를 명확히 이해하고 있다면,
방금 봤던 방식을 사용해서 다음 번 **선언**의 복잡도를 약간 줄여줄 수 있어요
 - 이렇게 한다 해도 컴파일 도중 컴파일러가 알아서 잘 deduce해 줘요
 - 물론, 이렇게 한다고 해서
선언 stat number; 의 number가 변수가 되는 것은 절대 아니에요(여전히 **배열**임)
 - stats의 경우엔 원래대로 적든 축약해서 적든 **배열**이 맞으니 크게 걱정 안 해도 되긴 해요

잠시 정리

- 여기까지 나온 내용을 정리하면...
 - '배열의 배열'에 겁 먹지 않아도 돼요.
'캐릭터 정보 하나'를 **배열**로 다룬다면,
'캐릭터 정보 여럿'은 **배열들의 배열**이 되는 게 당연해요
 - 계산 순서만 잘 확인하면 실수 없이 ^^ 가능해요
 - **Offset 값**을 외우기 불편하다면 #define을 통해 단어를 도입해 쓸 수 있어요
 - **Offset 값** 뿐만 아니라 '색상 값'과 같은 다양한 요소들에 적용 가능해요
 - typedef specifier를 붙인 **선언**을 적어 새 **형식 이름**을 도입할 수 있어요

잠시 휴식

- 여기까지 나온 내용을 정리하면...
 - '배열의 배열'에 겁 먹지 않아도 돼요.
'캐릭터 정보 하나'를 **배열**로 다룬다면,
'캐릭터 정보 여럿'은 **배열들의 배열**이 되는 게 당연해요
 - 계산 순서만 잘 확인하면 실수 없이 ^^ 가능해요
 - **Offset 값**을 외우기 불편하다면 #define을 통해 단어를 도입해 쓸 수 있어요
 - **Offset 값** 뿐만 아니라 '색상 값'과 같은 다양한 요소들에 적용 가능해요
 - typedef specifier를 붙인 **선언**을 적어 새 **형식 이름**을 도입할 수 있어요
- 잠시 쉬면서,
지금 완성해 둔 코드를 잠깐 구경해 보고 다음으로 넘어가 봅시다.

다시 시작

- 이번에는 **구조체** 이야기를 해 보려 해요.
 - '한 칸의 내용'을 직접 **정의**하고, **offset** 개념을 써서 각 부분을 사용할 수 있어요
 - 이를 납득하기 위해, 잠시 **offset** 개념을 약간 더 짚어 놓고 갈게요

Offset 다시보기

- 단어 **offset**은 '기준 위치' 및 그와의 거리를 중시함을 내포하고 있습니다.
 - 수식 `arr[100]`에서 `arr`은 기준 위치, 100은 **offset**이라 말할 수 있어요
 - `arr`에 대한 선언에 따라 **offset 값** 100이 '유효한 index'가 아닐 가능성이 존재하며, 두 단어의 뉘앙스 차이를 여기서 짚어 볼 수도 있어요

Offset 다시보기

- 뭐 그렇긴 하지만, **offset**을 쓰는 기본적인 이유는 '기준 **위치**에서 얼마만큼 떨어진 다른 **위치**'를 특정하기 위해서예요.
 - 수식 `arr[100]`에서 `arr`은 '(메모리 위) 한 칸의 위치', 100은 **offset**을 의미하며, 전체 수식은 '**offset**이 적용된 (메모리 위) 한 칸'을 의미해요
 - 무슨 **형식** 한 칸인지는 `arr` 선언에 의해 결정돼요
 - '칸' 개념 적용이 어려울 땐 뭐 deduction이 발생하거나 할 듯(지금은 그러려니 합시다)
- 이러한 특성을 바탕으로,
우리는 '유효한 **offset**'을 고정시켜 둔 채 기준 **위치**를 그 때 그 때 바꾸어 가면서
그 때 그 때 서로 다른 한 칸을 특정하여 사용할 수 있어요
 - 수식 `stats[0][STR]`에서
`stats[0]`은 '0번째 캐릭터 정보가 담긴 기준 **위치**', `STR`은 **offset**을 의미했지요.
이 `int` 한 칸은 수식 `stats[1][STR]`로 특정되는 `int` 한 칸과 서로 달랐어요

Offset 조금 더 보기

- 일반적으로, **offset 값**은 '기준 위치 값'에 비하면 그 범위가 작은 편이에요.
 - 지금 작성한 코드도,
캐릭터 수는 대폭 증가할 수 있겠지만 능력치 종류는 그렇지 않을 거예요
- 이러한 보편적 사실을 바탕으로,
CPU는 위의 **offset** 개념을 보다 적극적으로 활용하도록 설계되어 있어요.
 - 방금 전 파일은 살짝 빼 두고, CSP_5_2_yeshi_1.c를 살짝 구경해 봅시다
 - 이 부분은 그냥 수업자료만 구경해도 좋아요

Offset 조금 더 보기

- 탭재가 끝났다면 F10을 누른 다음 디스어셈블리 탭을 열어 보세요.
 - 이번에도 편의를 위해 x86 모드로 진행해 볼게요

Offset 조금 더 보기

- 디스어셈블리 창을 보면...

- main() 안 **문장들**(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0

// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0

// 함수 호출(F11을 누르면 노란 화살표가 함수 안으로 이동하는 것을 따라갈 수 있음)
func();
000E173F E8 78 FB FF FF      call     _func (0E12BCh)
```

- func() 안 **문장들**(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

Offset 조금 더 보기

- 디스어셈블리 창을 보면...

- main() 안 **문장들**(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0
```

```
// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0
```

```
// 함수 호출(F11을 누르면 노란 화살표가 함)
func();
000E173F E8 78 FB FF FF call     _func
```

일단 **상수 0**은 각 명령어들에 잘 내장되어 있습니다.

- func() 안 **문장들**(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

Offset 조금 더 보기

- 디스어셈블리 창을 보면...

- main() 안 **문장들**(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 00 mov     dword ptr ds:[0E9138h],0
```

static_number에 대해 정의된 위치 (상수) 값도
명령어 안에 통째로 내장되어 있는 것을 볼 수 있습니다.

```
func()
000E173F E8 78 FB FF FF call     _func (0E12BCh)
```

- func() 안 **문장들**(return문 제외)

```
// automatic 위치에 상수 0을
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

어셈블리 코드에 적혀 있는 값과 동일해요!
조사식 써서 &static_number 계산해도 이 값 나와요!

Offset 조금 더 보기

- 디스어셈블리 창을 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0
```

```
// automatic 위치에 상수 0을 담음
automatic_number = 0;
```

000E172E

애초에 '어셈블리 코드' 자체가
CPU가 이해 가능한 숫자들에 대한 글자 표현에 해당하므로
여기서 식별 가능한 모든 정보는
왼쪽에 보이는 숫자들에 '전부 다' 포함되어 있다고 볼 수 있어요!

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

Offset 조금 더 보기

- 디스어셈블리 창을 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0

// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0

// 함수 호출(F11을 누르면)
func();
000E173F E8 00 00 00 00 call    func
```

함수 호출(F11을 누르면)을 누르면, 함수 호출표가 함수 안으로 이동하는 것을 따라갈 수 있음)

그 말은...
automatic_number의 위치 또한
이 작은 숫자들로 특정 가능하다는 뜻이 되는데...

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

Offset 조금 더 보기

- 디스어셈블리 창을 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 00 mov     dword ptr ds:[0E9138h],0
```

```
// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0
```

```
// 함수 호출(F11을 누르면 노란 화살표가 함수 안으로 이동하는 것을 따라갈 수 있음)
func();
000E173F E8 00 00 00 00
```

어? 지금 보니까
여기도 위와 똑같은 명령어가 적혀 있어요!

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

Offset 조금 더 보기

- 디스어셈블리 창을 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0
```

```
// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0
```

// 함수 호출(F11을 누르면 노란 화살표가 함수 안으로 따라갈 수 있음)

이 두 automatic 변수들의 공통점은,
각 함수 정의 안에서 가장 처음 선언된 automatic 변수라는 점입니다.
둘 다 '첫 automatic 위치'를 가지고 있다고 가정할 수 있겠군요.

- func()

```
// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

Offset 조금 더 보기

- 디스어셈블리 하오 나면

- main() 안

```
// s
stat
000E172E
```

```
// automatic 위치
automatic_number = 0;
```

```
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number], 0
```

```
// 함수 호출(F11을 누르면 노란 화살표가 함수 안으로 이동하는 것을 따라갈 수 있음)
func();
```

```
000E173F E8 78 FB FF FF      call     _func (0E12BCh)
```

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음
number = 0;
```

```
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number], 0
```

맞아요. 여기 적힌 2B의 내용은,
'이번 함수 호출에서 쓰이는 기준 위치'에서 F8만큼 옆
...을 의미해요.

잠시 VS 화면 스샷을 구경해 보면...

Offset 조금 더 보기

주소(A): func(...)

보기 옵션

```
000E16DC 8D BD 34 FF FF FF lea edi,[ebp-0CCh]
000E16E2 B9 33 00 00 00
000E16E7 B8 CC CC CC CC
000E16EC F3 AB
// automatic 위치를 갖는 int number;
// automatic 위치에 상수 0
number = 0;
000E16EE C7 45 F8 00 00 00
return number;
000E16F5 8B 45 F8
}
000E16F8 5F pop edi
000E16F9 5E pop esi
000E16FA 5B pop ebx
000E16FB 8B E5 mov esp,ebp
000E16FD 5D pop ebp
```

이 친구가 그 '기준 위치'를 나타내는 레지스터예요.
(레지스터 창은 Ctrl + D, R 누르면 열 수 있음)

EBP는 extended base pointer의 약자고,
여기서의 pointer는 '(하드웨어적) 위치 값'을 뜻해요.

조사식 1

이름	값	형식
&static_number	9L_Yeshi_1.exe!0x000e9138 {0}	int *
&automatic_number	0x00b3f878 {-858993460}	int *
&number	0x00b3f798 {-858993460}	int *

레지스터

EAX = CCCCCC EBX = 009F8000 ECX = 00000000 EDI = 00B3F7A0 ESI = 000E1055
EIP = 000E16EE ESP = 00B3F6C0 EBP = 00B3F7A0 EFL = 00000216

0x00B3F798 = CCCCCC

출력 로컬 조사식 1 스레드

레지스터 호출 스택 예외 설정 직접 실행 창

Offset 조금 더 보기

주소(A): func(...)

보기 옵션

```
000E16DC 8D BD 34 FF FF FF lea     edi,[ebp-0CCh]
000E16E2 B9 33 00 00 00     mov     ecx,33h
000E16E7 B8 CC CC CC CC     mov     eax,0CCCCCCCCh
000E16EC F3 AB             rep stos dword ptr es:[edi]
// automatic 위치를 갖는 int 변수
int number;

// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0

return number;
000E16F5 8B 45             mov     eax,dword ptr [number]
}
000E16F8 5B             pop     edi
000E16F9 5E             pop     esi
```

지금 func() 내용물을 실행하는 시점이고,
EBP 또한 그에 맞게 세팅되어 있어요.

레지스터

EAX = CCCCCC EBX = 009F8000 ECX = 00000000 EDX = 000E9588 ESI = 000E1055
EDI = 00B3F7A0 EIP = 000E16EE ESP = 00B3F6C8 EBP = 00B3F7A0 EFL = 00000216

0x00B3F798 = CCCCCC

출력 로컬 조사식 1 스레드

레지스터 호출 스택 예외 설정 직접 실행 창

Offset 조금 더 보기

주소(A): func(...)

보기 옵션

```
000E16DC 8D BD 34 FF FF FF lea     edi,[ebp-0CCh]
000E16E2 B9 33 00 00 00     mov     ecx,33h
000E16E7 B8 CC CC CC CC     mov     eax,0CCCCCCCCh
000E16EC F3 AB             rep stos dword ptr es:[edi]
// automatic 위치를 갖는 int 변수
int number;

// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
// automatic 위치를 갖는 int 변수
number;
000E16F5 45 F8             mov     eax,dword ptr [number]
000E16F9 5F             pop     edi
000E16FE 5E             pop     esi
```

여기 있는 F8은
automatic 변수 number에 대해 정의된 **offset** (상수) 값이고,
-8을 의미해요.

number에 대한 실제 위치는 EBP - 8로 얻을 수 있어요.

레지스터 | 호출 스택 | 예외 설정 | 직접 실행 창

출력 | 로컬 | 조사식 1 | 스레드

EBX = 009F8000 ECX = 00000000 EDX = 000E9588 ESI = 000E1055
EIP = 000E16EE ESP = 00B3F6C8 EBP = 00B3F7A0 EFL = 00000216

Offset 조금 더 보기

주소(A): func(...)

보기 옵션

```
000E16DC 8D BD 34 FF FF FF lea edi,[ebp-0CCh]
000E16E2 B9 33 00 00 00 mov ecx,33h
000E16E7 B8 CC CC CC CC mov eax,0CCCCCCCCh
000E16EC F3 AB rep stos dword ptr es:[edi]
// automatic 위치를 갖는 int 변수
int number;
```

현재 EBP 값은 0x00B3F7A0이에요.

조사식 1

이름	값	형식
&static_number	9L_Yeshi_1.exe!0x000e9138 {0}	int *
&automatic_number	0x00b3f878 {-858993460}	int *
&number	0x00b3f798 {-858993460}	int *

레지스터

EAX = CCCCCC EBX = 009F8000 ECX = 33 EDI = 00B3F7A0 EIP = 000E16EE ESP = 00B3F6C8 EBP = 00B3F7A0 EFL = 00000216

0x00B3F798 = CCCCCC

출력 로컬 조사식 1 스레드 레지스터 호출 스택 예외 설정 직접 실행 창

Offset 조금 더 보기

The screenshot shows a debugger window with the following components:

- 주소(A): func(...)**
 - 보기 옵션**
 - 000E16DC 8D BD 34 FF FF FF lea edi,[ebp-0CCh]
 - 000E16E2 B9 33 00 00 00 mov ecx,33h
 - 000E16E7 B8 CC CC CC CC mov eax,0CCCCCCCCh
 - 000E16EC F3 AB rep stos dword ptr es:[edi]
 - // automatic 위치를 갖는 int 변수
 - int number;
- 조사식 1**

이름	형식
&static_number	int *
&automatic_number	int *
&number	int *
- 레지스터**

EAX = CCCCCC EBX = 009F8000 ECX = 00000000 EDX = 000E9588 ESI = 000E1055
EDI = 00B3F7A0 EIP = 000E16EE ESP = 00B3F6C8 EBP = 00B3F7A0 EFL = 00000216
0x00B3F798 = CCCCCC

출력 로컬 조사식 1 스레드 레지스터 호출 스택 예외 설정 직접 실행 창

현재 EBP 값은 0x00B3F7A0이에요.

그래서, '이번 호출에서의' number의 실제 위치는
 $0x00B3F7A0 - 0x00000008 \rightarrow 0x00B3F798$ 이 되지요!

Offset 조금 더 보기

주소(A): func(...)

보기 옵션

```
// static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0

// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0

// 함수 호출(F11을 누르면 노란 화살표가 함수 안으로 이동하는 것을 따라갈 수 있음)
func();
000E173F E8 78 FB FF FF      call     _func (0E12BCh)

return 0;
000E1744 33 C0      xor     eax, eax  경과 시간 1ms 이하
}
```

46 5F pop edi
747 5E pop esi
748 5B pop ebx

func() 내용물 실행이 끝나고 main()으로 돌아오면...

레지스터

EAX	=	00000000	EBX	=	009F8000	ECX	=	00000000	EDX	=	000E9588	ESI	=	000E1055
EDI	=	00B3F880	EIP	=	000E1744	ESP	=	00B3F7A8	EBP	=	00B3F880	EFL	=	00000216

출력 로컬 조사식 1 스레드

레지스터 호출 스택 예외 설정 직접 실행 창

Offset 조금 더 보기

The screenshot shows a debugger window with the following assembly code:

```
주소(A): func(...)
보기 옵션
// static 변수 초기화
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0

// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0

// 함수 호출(F11을 누르면 노란 화살표가 함수 안으로 이동하는 것을 따라갈 수 있음)
func();
000E173F E8 78 FB FF FF      call    _func (0E12BCh)

return 0;
000E1744 33 C0                  xor     eax, eax
}
000E1746 5F                  pop     edi
000E1747 5E                  pop     esi
000E1748 5B                  pop     ebx
```

A callout box points to the EBP register in the register window, containing the text:

func() 내용물 실행이 끝나고 main()으로 돌아오면...
EBP도 원래 main() 호출에 대해 쓰이던 값으로 복구돼요.
0x00B3F7A0이었는데 이젠 0x00B3F880이 되어 있어요.

The register window shows the following values:

레지스터	값
EAX	00000000
EBX	009F8000
ECX	00000000
EDX	000E9588
ESI	000E1055
EDI	00B3F880
EIP	000E1744
ESP	00B3F7A0
EBP	00B3F880
EFL	00000216

The EBP register value is highlighted with a red box. The bottom status bar shows tabs for '출력', '로컬', '조사식 1', '스레드', '레지스터', '호출 스택', '예외 설정', and '직접 실행 창'.

Offset 조금 더 보기

주소(A): func(...)

보기 옵션

```
// static 위치에 상수 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0

// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0
```

(F11을 누르면 노란 화살표가 함수 안으로 이동하는 것을 따라갈 수 있음)

이미 확인했듯,
automatic 변수 automatic_number에 대해 정의된 **offset (상수) 값** 또한
func()의 number와 동일한 F8이에요.

이름	값	형식
&static_number	9L_Yeshi_1.exe!0x000e9138 {0}	int *
&automatic_number	0x00b3f878 {0}	int *
&number	0x00b3f878 {0}	int *

레지스터

EAX = 00000000 EBX = 009F8000 ECX = 00000000 EDX = 000E9588 ESI = 000E1055
EDI = 00B3F880 EIP = 000E1744 ESP = 00B3F7A8 EBP = 00B3F880 EFL = 00000216

출력 로컬 조사식 1 스레드 레지스터 호출 스택 예외 설정 직접 실행 창

Offset 조금 더 보기

주소(A): func(...)

보기 옵션

```
// static 변수에 0을 담음
static_number = 0;
000E172E C7 05 38 91 0E 00 00 00 00 mov     dword ptr ds:[0E9138h],0

// automatic 위치에 상수 0을 담음
automatic_number = 0;
000E1738 C7 45 F8 00 00 00 00 mov     dword ptr [automatic_number],0

// 함수 호출(F11을 누르면 노란 화살표가 함수 안으로 이동하는 것을 따라갈 수 있음)
func();
```

하지만, '방금 전과 EBP 값이 다르기 때문에',
automatic_number의 실제 위치는
방금 전 number의 실제 위치와 다른 것을 볼 수 있어요.
(아까 &number는 0x00B3F798이었고,
지금은 이미 증발했으니 요상하게 보이는 게 정상임)

이름	값	형식
&static_number	9L_Yeshi_1.exe:000E9138 {0}	int *
&automatic_number	0x00b3f878 {0}	int *
&number	0x00b3f878 {0}	int *

레지스터
EAX = 00000000 EBX = 009F8000 ECX = 00000000 EDX = 000E9588 ESI = 000E1055
EDI = 00B3F880 EIP = 000E1744 ESP = 00B3F7A8 EBP = 00B3F880 EFL = 00000216

출력 로컬 조사식 1 스레드 레지스터 호출 스택 예외 설정 직접 실행 창

Offset 조금 더 보기

- 이 코드에 대한 컴파일 결과(디스어셈블리 창)를 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
```

```
static_n  
000E172E C7
```

```
// autom  
automatic  
000E1738 C7
```

```
// 함수 호출(F11을 누르면 노란  
func();  
000E173F E8
```

```
78 FB FF FF
```

```
call
```

```
_func (0E12BCh)
```

비슷한 느낌으로,
call 명령어처럼 '노란 화살표 자체'의 위치를 바꾸는 상황에서는
'현재'의 노란 화살표 값(얘는 EIP에 담김!)을 기준 위치로 삼아요.

후 안으로 이동하는 것을 따라갈 수 있음)

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음  
number = 0;
```

```
000E16EE C7 45 F8 00 00 00 00 mov
```

```
dword ptr [number],0
```

Offset 조금 더 보기

- 이 코드에 대한 컴파일 결과(디스어셈블리 창)를 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
```

```
static_n  
000E172E C7
```

```
// autom  
automatic  
000E1738 C7
```

실제 **함수** 호출을 하는 시점은 '이 명령어를 다 읽은 시점' 이후가 되므로,
그 시점의 EIP는 $0x000E173F + 5 \rightarrow 0x000E1744$ 가 돼요.
(일단 읽어야 이게 call인지 mov인지 등등을 알 수 있게 됨!)

```
// 함수 호출(F11을 누르면 실행되는 함수로 이동하는 것을 따라갈 수 있음)  
func();
```

```
000E173F E8 78 FB FF FF call _func (0E12BCh)
```

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음  
number = 0;
```

```
000E16EE C7 45 F8 00 00 00 00 mov dword ptr [number],0
```

Offset 조금 더 보기

- 이 코드에 대한 컴파일 결과(디스어셈블리 창)를 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
static_n
000E172E C7

// autom
automatic
000E1738 C7

// 함수 호출(F11을 누르면)
func();
000E173F E8 78 FB FF FF call _func (0E12BCh)
```

0xFFFFFB78은 -1160을 의미해요.
호출 시점의 EIP는 0x000E1744니까...

이것이 함수 안으로 이동하는 것을 따라갈 수 있음)

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음
number = 0;
000E16EE C7 45 F8 00 00 00 00 mov dword ptr [number],0
```

Offset 조금 더 보기

- 이 코드에 대한 컴파일 결과(디스어셈블리 창)를 보면...

- main() 안 문장들(return문 제외)에 대한 명령어들

```
// static 위치에 상수 0을 담음
```

```
static_n  
000E172E C7
```

```
// autom  
automatic  
000E1738 C7
```

0xFFFFFB78은 -1160을 의미해요.
호출 시점의 EIP는 0x000E1744니까...
이 둘을 더하면 정확히 여기 적힌 위치 값이 나와요!

```
// 함수 호출(F11을 누르면 노란 화살표가 함수로 이동하는 것을 따라갈 수 있음)  
func();  
000E173F E8 78 FB FF FF      call     _func (0E12BCh)
```

- func() 안 문장들(return문 제외)에 대한 명령어들

```
// automatic 위치에 상수 0을 담음  
number = 0;  
000E16EE C7 45 F8 00 00 00 00 mov     dword ptr [number],0
```

Offset 조금 더 보기

- 방금 구경한 내용은 지금은 직접 살펴보기 조금 어려울 거예요.
 - EBP 값을 변경하는 것 또한 실제 명령어를 써서 수행되므로,
'내 문장'에 대한 명령어에 노란 화살표가 가 있을 때만 그 값을 구경해 보는 게 좋아요
 - 함수 정의의 각 중괄호에 대한 명령어가 두툼한 이유 중 하나도 여기에 있음!
- 일단 지금은 그냥,
'automatic 변수의 위치를 특정하기 위한 기준 위치로 EBP를 사용한다'
...정도만 가져가 두세요
 - 이 사실을 가지고 예전에 피보나치 수 구하는 예제를 다시 보면
왜 그 변수들의 위치가 그렇게 보였는지 납득할 수 있을 거예요!
- call 명령어의 동작은, 지금은 뭐 크게 고민하지 않아도 돼요.
컴파일러는 call 명령어의 위치, 실제 함수의 위치(항상 static함)를 모두 알고 있으므로
적절한 offset 값을 계산해서 미리 담아 둘 수 있어요!

Offset 조금 더 보기

- 한 걸음 밖으로 나와서 요약하면...
 - 메모리 자체는 매우 크기 때문에, 그걸 그냥 다루기보다는 몇 가지 기준 **위치**를 잡고 **offset**을 다루는 편이 더 효율적이예요

Offset 조금 더 보기

- 한 걸음 밖으로 나와서 요약하면...
 - 메모리 자체는 매우 크기 때문에, 그걸 그냥 다루기보다는 몇 가지 기준 **위치**를 잡고 **offset**을 다루는 편이 더 효율적이예요
 - 프로그래머가 **수식**으로써 적을 수 있는 **이름**은 전부 다 '지금 당장 보이는 **이름**'들이예요
 - 다른 **함수**(심지어 이 **함수**를 호출한 **함수**) **정의** 안에서 **선언**한 이름은 전혀 보이지 않으므로 그 **이름**들에 대해 **정의**된 칸들은 현 시점에는 전혀 중요하지 않아요
(**위치 값**을 별도로 **계산**해서 호출할 때 인수 자리에 담아 준다면 뭐 그건 이야기가 다르긴 함)
 - 그러므로 매 **함수** 호출마다 새로운 기준 **위치**를 잡아 활용하도록 설계하는 것은 나름 타당해요

Offset 조금 더 보기

- 한 걸음 밖으로 나와서 요약하면...
 - 메모리 자체는 매우 크기 때문에, 그걸 그냥 다루기보다는 몇 가지 기준 위치를 잡고 offset을 다루는 편이 더 효율적이예요
 - 이런 기준 위치들을 토대로 전체 메모리 범위를 몇 가지 영역으로 나눌 수 있긴 해요
 - Code들이 모여 있는 곳
 - Static Data 친구들이 모여 있는 곳
 - Automatic Data 친구들이 잠시 지내다 증발하는 곳
 - 위의 어느 영역에도 포함되지 않는 곳
 - 일단은 '뭐 다들 다른 데 사나보다' 정도만 생각해 두고, 이들의 정체는 좀 더 어려운 과목 수업에서 좀 더 정확하게 파악해 보면 좋을 듯 해요

Offset 조금 더 보기

- 한 걸음 밖으로 나와서 요약하면...
 - 메모리 자체는 매우 크기 때문에, 그걸 그냥 다루기보다는 몇 가지 기준 위치를 잡고 offset을 다루는 편이 더 효율적이예요
- 컴파일러는 모든 **이름**에 대한 **위치 (상수) 값**을 본인이 직접 **정의**하며 각 명령어에 내장될 **offset 값** 또한 본인이 직접 **계산**할 수 있으므로 컴파일러가 **offset** 개념을 적극적으로 도입해 쓴다 하더라도 문제가 생길 일은 없어요
 - 컴파일러는 실수를 하지 않아요

Offset 조금 더 보기

- 한 걸음 밖으로 나와서 요약하면...
 - 반면, 우리가 직접 **offset** 개념을 사용할 때는 그렇지 않을 수 있어요.
뭐 `#define`을 쓴다거나 해서 어느 정도 예방할 수는 있겠지만
그렇다 해도 [] 연산자의 우항 자리에 항상 적절한 수식을 적는다는 보장은 어려워요
 - 특히, 내 실수(적으면 안 되는 칸에 적음)로 인해
컴파일러가 짜 둔 **offset**들의 동작이 어그러지기 시작하면 문제가 좀 커져요
 - Python이라면 우리를 신뢰하지 않기 때문에 알아서 검사해서 `IndexError`를 보여주지만,
기본적으로 C 컴파일러는 우리를 신뢰하기 때문에 그런 서비스를 제공하지 않아요

Offset 조금 더 보기

- 한 걸음 밖으로 나와서 요약하면...
 - 반면, 우리가 직접 **offset** 개념을 사용할 때는 그렇지 않을 수 있어요.
뭐 `#define`을 쓴다거나 해서 어느 정도 예방할 수는 있겠지만
그렇다 해도 [] 연산자의 우항 자리에 항상 적절한 수식을 적는다는 보장은 어려워요
- ...그리고 이러한 난관을 보다 효과적으로 돌파하기 위해
C에 있는 **구조체** 개념을 도입하여 사용할 수 있어요!
 - 다음 슬라이드부터 설명 시작함!

구조체

- 아래 예시를 봅시다:

```
int;
```

```
typedef int stat[3];
```

```
stat;
```

```
struct StatType;
```

구조체

- 아래 예시를 봅시다:

int;

typedef in

stat;

struct StatType;

예전에 잠시 등장했던,
specifier만 적혀 있는 C 선언입니다.

'int'야... 그냥 그렇다구' 정도의 의미를 갖는 듯 합니다.

구조체

- 아래 예시를 봅시다:

```
int;
```

```
typedef int stat[3];
```

```
stat;
```

```
struct StatType;
```

앞에서 등장한,
typedef specifier가 달린 선언입니다.

'새 형식 이름 stat은 int 세 칸 짜리 배열을 의미해'
...정도로 해석할 수 있습니다.

구조체

- 아래 예시를 봅시다:

```
int;
```

```
typedef int stat[3];
```

```
stat;
```

이제 컴파일러도 형식 이름 stat의 정체를 알고 있으므로,

'(방금 말했던)stat이야... 그냥 그렇다구'
...정도의 의미를 갖는 선언을 적을 수 있습니다.

```
struct StatType;
```


구조체

- 아래 예시를 봅시다:

```
int;
```

```
typedef int
```

```
stat;
```

```
struct StatType;
```

이 선언 또한 문법적으로는 위 둘과 동일합니다.

'구조체 StatType이야... 그냥 그렇다구'
...정도의 의미를 가지지요.

구조체

- 아래 예시를 봅시다:

```
int;
```

```
typedef int
```

```
stat;
```

```
struct StatType;
```

반면, 그 효과는 위의 둘과 약간 다릅니다.

C 컴파일러는 StatType이 '구조체 이름'인 줄 몰랐는데 이 선언을 읽으며 '오, 그래?' 하며 새 이름을 알게 됩니다.

구조체

- 아래 예시를 봅시다:

```
int;
```

```
//typedef int stat[3];
```

```
stat;
```

이 경우는 그렇지 않습니다.

위의 typedef 붙은 선언이 없다면
이 선언의 stat은 절대 '형식 이름'으로 간주되지 않습니다.
이 선언에는 '새 이름 도입' 효과는 들어 있지 않아요!

```
struct StatType;
```

구조체

- 아래 예시를 봅시다:

```
struct StatType;
```

아무튼, 다시 이 선언으로 돌아와서...

```
struct StatType stats[4];
```

구조체

- 아래 예시를 봅시다:

```
struct StatType;
```

위 선언에 declarator를 붙이면
이런 느낌의 **배열 선언** 또한 할 수 있을 것입니다.

배열 선언인 만큼 컴파일 도중에 **정의** 또한 하게 되겠지요.

```
struct StatType stats[4];
```

구조체

- 아래 예시를 봅시다:

```
struct StatType;
```

그래서, 이것 그냥 두면 '정의 오류'가 뜨게 됩니다.

StatType이 **구조체 이름**이라는 것은 알지만,
그 한 칸이 어떻게 생겼는지는 전혀 모르기 때문이지요.

```
struct StatType stats[4];
```

구조체

- 아래 예시를 봅시다:

```
struct StatType;
```

함수 정의나 typedef 붙은 선언이 그러했듯,
StatType이 정확히 무엇인지 그 내용을 정의하는 것은
새 이름을 정한 우리가 해야 할 일입니다.

```
struct StatType stats[4];
```

구조체

- 아래 예시를 봅시다:

```
struct StatType  
{  
    int str;  
    int dex;  
    int con;  
};
```

그리고, 그런 목표에 따라 이렇게 적어 놓는 것을
'구조체 정의'라 부릅니다.

```
struct StatType stats[4];
```


구조체

- 아래 예시를 봅시다:

```
struct StatType  
{  
    int str;  
    int dex;  
    int con;  
};
```

이처럼, C에서 구조체 정의는
중괄호 안에 선언을 적는 것으로 구성됩니다.
C에서는 여기 선언된 친구들을 멤버라고 불러요.

```
struct StatType stats[4];
```

구조체

- 아래 예시를 봅시다:

```
struct StatType  
{  
    int str;  
    int dex;  
    int con;  
};
```

여기 있는 **구조체 정의**를 통해 C 컴파일러는
'StatType 한 칸'이 'int 세 칸'으로 구성됨을 알 수 있게 됩니다.

```
struct StatType stats[4];
```

구조체

- 아래 예시를 봅시다:

```
struct StatType
{
    int str;
    int dex;
    int con;
};
```

여기 있는 **구조체 정의**를 통해 C 컴파일러는
'StatType 한 칸'이 'int 세 칸'으로 구성됨을 알 수 있게 됩니다.
따라서 StatType 네 칸 짜리 **배열**을 정의할 수도 있게 됩니다!

```
struct StatType stats[4];
```

구조체

- 아래 예시를 봅시다:

```
struct StatType  
{  
    int str;  
    int dex;  
    int con;  
};
```

```
int;
```

일단 한 걸음 늦춰서, 여전히 **구조체 정의**는 선언 문법상 'specifier 하나'의 범주에 머무릅니다.

이 큰 덩어리가 int 달랑 하나와 동격이 돼요!

구조체

- 아래 예시를 봅시다:

```
struct StatType
```

```
{
```

```
    int str;
```

```
    int dex;
```

```
    int con;
```

```
} stats[4];
```

```
int arr[4];
```

따라서, 이렇게 바로
새 변수 / 포인터 변수 / 배열 / 함수 이름을 선언하거나...

구조체

- 아래 예시를 봅시다:

```
typedef struct StatType
```

```
{
```

```
    int str;
```

```
    int dex;
```

```
    int con;
```

```
} stat;
```

```
stat stats[4];
```

긴 이름을 매 번 부르기 번거롭다면
이렇게 바로 새 **형식 이름**을 선언해 사용할 수도 있습니다!
(아마 몇몇 수업자료에서 이런 표현을 볼 수 있을 거예요)

구조체

- 여기까지 나온 내용을 요약하면...
 - **구조체 이름**이 처음 등장하면 컴파일러는 그 **이름**을 알게 돼요
 - `struct` 뒤에 단어 적으면 그 단어는 **구조체 이름**이 돼요
 - **함수**가 그랬듯, **이름**만 알려준다고 컴파일러가 내용물까지 다 아는 것은 아니므로, 적절한 곳에 **구조체 정의**를 미리 적어 주어야 해요
 - **함수 정의**와 다르게 **구조체 정의**는 '연결'의 대상이 아니므로 '해당 **구조체 형식**을 사용'하는 곳보다 위에 적어 주어야 해요
 - 뭐 이 규칙은 그냥 '파일 윗 부분에 해 두자' 하면 되니 지금은 잊어도 좋아요
 - **구조체 정의** 내용물은 **멤버 선언**으로 구성돼요
 - C 컴파일러는 **멤버 선언**들을 읽음으로써 '**구조체** 한 칸'에 대한 정보를 얻어요
 - 그렇다 하더라도 여전히 **구조체 정의**는 문법상 '**specifier** 하나'에 해당해요

. 연산자

- 여차저차해서 만든 **구조체** 한 칸의 내용물을 다룰 때는 **. 연산자**를 씁니다.

```
stats[0].str
```

```
stats[0].dex
```

```
stats[0].con
```


. 연산자

- 여차저차해서 만든 **구조체** 한 칸의 내용물을 다룰 때는 **. 연산자**를 씁니다.

stats[0].str

stats[0].dex

stats[0].con

Python에서 썼던 **. 연산자**와 느낌이 비슷해요!

. 연산자

- 여차저차해서 만든 **구조체** 한 칸의 내용물을 다룰 때는 **. 연산자**를 씁니다.

stats[0].str

stats[0][0]

방금 전에 직접 적어 본 표기법이랑 비교해 보면...

. 연산자

- 여차저차해서 만든 구조체 한 칸의 내용물을 다룰 때는 . 연산자를 씁니다.

stats[0].str

stats[0][0]

방금 전에 직접 적어 본 표기법이랑 비교해 보면...
이 부분도 뭔가 '한 칸 안에서의 **offset**'을 적용하는 표현이라
가정해 볼 수 있을 거예요.

구조체와 offset

- 일단은, 방금 전 했던 가정은 사실이에요.
 - C에서 . 연산자는 본질적으로 '**offset**'을 적용하는' 연산자예요
 - 우항 자리에 아무 숫자 / 이름이나 적을 수는 없고,
구조체 정의 적을 때 선언해 둔 멤버 이름들만 적을 수 있어요!

구조체와 offset

- 일단은, 방금 전 했던 가정은 사실이에요.
 - C에서 . 연산자는 본질적으로 '**offset**을 적용하는' 연산자예요
 - 우항 자리에 아무 숫자 / 이름이나 적을 수는 없고,
구조체 정의 적을 때 선언해 둔 멤버 이름들만 적을 수 있어요!
- 이 부분은 직접 예시를 구경해 보는 게 좋을 듯 해요.
 - CSP_5_2_yeshi_2.cpp를 열어 확인해 봅시다.
 - C에는 없는 연산자 하나를 써서 ^^하는 예시임
(중간고사 이후에 다시 정식으로 등장할 예정이니 지금은 구경만 해 봅시다)

구조체와 offset

- 복습을 위한 슬라이드
 - **멤버 선언들에 대해, C 컴파일러는 offset (상수) 값을 정의해요**
 - 구조체 한 칸의 맨 앞 위치가 '기준 위치'가 돼요
 - 일반적으로 첫 **멤버 이름**에 대한 **offset 값**은 0이에요
(굳이 앞부분에 빈 공간을 잡아 둘 필요는 없으니 당연)
 - 각 **멤버들에 대한 칸이 항상 딱 붙어 있는 것은 아니에요**
 - 이걸 보장하는 건 **배열**이에요. 구조체는 **배열**이 아니에요
 - 각 **멤버들이 항상 동일한 형식일 필요는 없어요**
 - 이걸 요구하는 건 **배열**이에요. 구조체는 **배열**이 아니에요

구조체와 offset

- (중요)구조체의 이러한 특성은...
 - 형식에 구애받지 않고 '한 가지 의도를 위한 Data'를 한 칸에 몰아 담아 활용할 수 있어요
 - 프로그래머가 **구조체 정의**를 적음으로써 '내용물의 구성'을 의도해 두면, 컴파일러가 각 **멤버**에 대한 적절한 **offset 값**들과 '전체 한 칸의 크기'를 정해줘요
 - 내가 선언하지 않은 **멤버 이름**을 **. 연산자** 우항 자리에서 사용할 수 없으므로, **offset 값**을 실수로 '칸 너머'로 지정하는 실수를 원천적으로 차단할 수 있어요
 - 컴파일러는 절대 '칸 너머'에 해당하는 **offset 값**을 **정의**해 주지 않아요!
 - (이건 덜 중요)**배열**과 달리 '한 칸'으로 간주되므로, '**구조체 값**을 return하는 **함수**'를 만들 수 있어요
 - 이렇게 구성하면 느리기 때문에 C 실전에서는 잘 안 쓰는 편이긴 해요. C++로 넘어가서야 가끔 등장하게 될 거예요

마무리

- 아무튼 이렇다 보니,
C++로 넘어가서 조금 큰 프로그램을 만들기 시작하는 시점에는
아마도 **구조체 정의** 등을 통해 새 **형식**을 만들어 가며 진행하게 될 거예요.
- 그러니 오늘은 이 정도로만 구경해 두어도 좋을 것 같아요!