

4-2. 선언 적을 때 고려해야 할 것들 (2)

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

이번 시간에는

- 선언 적을 때 고려해야 할 것들 (2)
 - 선언 읽는 법
 - 일단은 세 가지 규칙만 가져가면 돼요
 - 읽는 법을 먼저 익혀 두면, 좀 더 복잡한 선언을 적을 때 도움이 많이 될 거예요
 - **Lvalue**, `const` 이야기
 - 애는 명칭에 비해 좀 이질적인 의미를 가져요

선언 읽는 법

- 일단 쉬운 것부터 가 봅시다!

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int x ;

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int

x

;

이 선언에 의하면 x는 무엇의 이름인가요?

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int

x

;

이 선언에 의하면 x는 무엇의 이름인가요?
int 변수 이름입니다.

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int *X ;

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int

*X

;

int 포인터 변수 이름입니다.

당분간은 단어 '**포인터**'는 개념을 가리키는 명칭으로 생각해 두고,
구체적인 무언가를 논의할 때는
'**포인터 형식**', '**포인터 변수**', '**포인터 값**' 정도로 풀어서 서술해 봅시다.

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

```
int      x[3]      ;
```

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int x[3] ;

(세 칸 짜리) int 배열 이름입니다.

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

```
int          x()          ;
```

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int x() ;

(인수는 뭔지 모르지만) int 값을 return하는 함수 이름입니다.

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

```
int *x[3];
```

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

int *x[3] ;

?

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

지난 수업에서 약간 힌트가 나오기는 했어요.
어떤 이름에 대해 '그 이름 자체를 사용하기 위한' 연산자가 양 쪽에 붙어 있다면
기본적으로 오른쪽에 있는 게 먼저 붙어요!

int *x[3] ;

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

그래서 이 선언의 x는 '일단 (세 칸짜리) 배열'이에요!

(중요)이 이름이 본질적으로 무엇인지는 '맨 처음 뭐가 붙는지'에 따라 결정돼요.

int *x[3];

①

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

무언가가 **배열**이라면,
그 다음에 붙는 것은 '각 칸에 무엇을 담나'를 의미해요.

여기서 x의 각 칸에는 '포인터 값'이 담겨요(각 칸은 딱 그만한 크기를 가져요).

int *x[3] ;

 (2) (1)

선언 읽는 법

- 자주 보던 선언을 들고 와 봤어요:

무언가가 **포인터**라면,
그 다음에 붙는 것은 '무엇에 대한 **포인터**인가'를 의미해요.

여기서 x의 각 칸에 담기는 **포인터 값**들은 모두 'int (한 칸에 대한) **포인터 값**'들이에요.

int

③

*x[3]

②

①

;

선언 읽는 법

- 방금 나온 내용을 잠깐 정리하면...
 - 선언에서 **이름** 좌/우에 기호가 붙어 있다면 기본적으로 오른쪽 것이 먼저 붙어요
 - '이 **이름**은 뭐 **이름**이냐'를 물으면 가장 먼저 붙는 기호만 보면 돼요
 - 방금 붙인 기호가...
 - *이라면, 다음에 붙는 것은 '이 **위치 값**은 뭐 한 칸에 대한 **위치 값**인가'를 의미해요
➢ 보통은 '따라가면 뭐 나오나' 정도로 축약해서 부르곤 해요
 - []라면, 다음에 붙는 것은 '각 칸은 뭐 담는 칸인가'를 의미해요
 - ()라면, 다음에 붙는 것은 '무슨 **형식 값**을 return하나'를 의미해요

선언 읽는 법

- 우리보다 선언 더 잘 읽는 VS의 도움을 받아 봅시다.
평소 쓰던 그냥 프로젝트/파일 연 다음에 아래 내용을 복붙해 보세요:

```
#include <stdio.h>

int main()
{
    int number;
    int *ptr;
    int arr[3];
    int main();

    int *x[3];

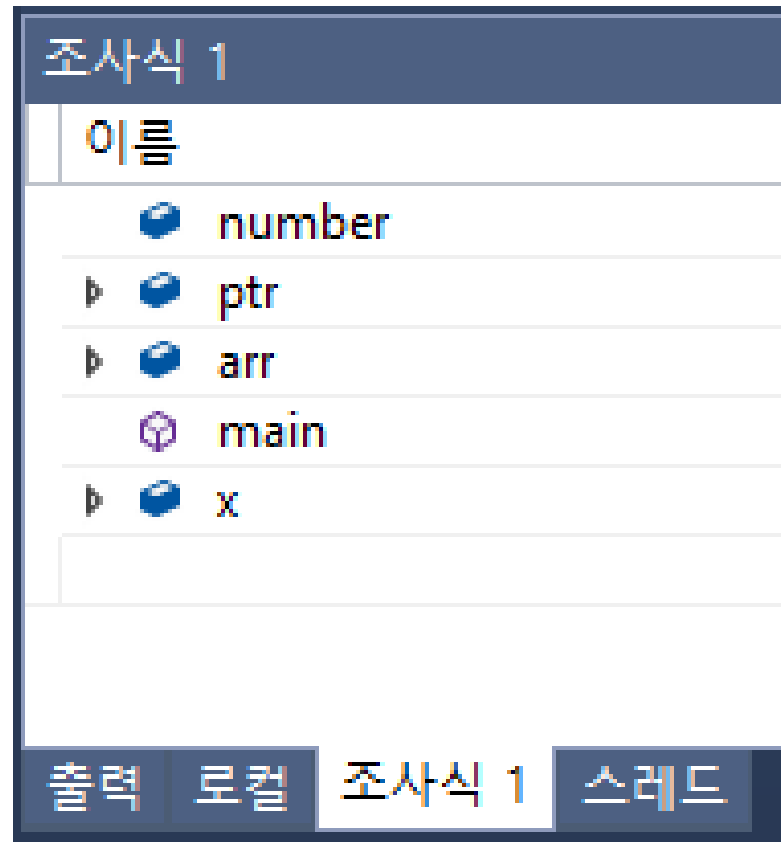
    // 나중에 여기에 더 적을 예정
}
```

선언 읽는 법

- 우리보다 **선언** 더 잘 읽는 VS의 도움을 받아 봅시다.
평소 쓰던 그냥 프로젝트/파일 연 다음에 아래 내용을 입력해 보세요:
- 다 적었으면 F10을 눌러 디버깅을 시작한 다음 조사식 탭을 준비해 보세요.
 - 조사식 탭이 없다면 디버그(D) → 창(W) → 조사식(W) → 조사식 1(1) 눌러요

선언 읽는 법

- 이제 조사식 탭의 **이름** 칸에 아래 내용을 적어 봅시다:
 - 일단 **이름** 적고 엔터 치면 아래처럼 아이콘 등을 그려줄 거예요



선언 읽는 법

- 이제 조사식 탭의 **형식** 칸을 구경해 봅시다.

선언 읽는 법

- 이제 조사식 탭의 **형식** 칸을 구경해 봅시다.
 - 선언에 적어 둔, **이름**을 제외한 다른 글자들이 똑같이 적혀 있으면 성공이에요!
 - 함수 이름의 경우 괄호 안이 조금 오묘하게 나오는 게 정상이니 걱정 ㄴㄴ

선언 읽는 법

- 이제 조사식 탭의 **형식** 칸을 구경해 봅시다.
 - 선언에 적어 둔, **이름**을 제외한 다른 글자들이 똑같이 적혀 있으면 성공이에요!
 - 함수 이름의 경우 괄호 안이 조금 오묘하게 나오는 게 정상이니 걱정 ㄴㄴ
- 이번에는 조사식 탭의 **이름** 칸에 수식 $x[0]$ 을 적고 엔터 키를 쳐 봅시다.

선언 읽는 법

- 이제 조사식 탭의 **형식** 칸을 구경해 봅시다.
 - 선언에 적어 둔, **이름**을 제외한 다른 글자들이 똑같이 적혀 있으면 성공이에요!
 - 함수 이름의 경우 괄호 안이 조금 오묘하게 나오는 게 정상이니 걱정 ㄴㄴ
- 이번에는 조사식 탭의 **이름** 칸에 **수식** $x[0]$ 을 적고 엔터 키를 쳐 봅시다.
 - x 에 대한 선언에서 $x[3]$ 부분을 제외한 다른 글자들이 똑같이 적혀 있을 거예요!

선언 읽는 법

- 오옹... 방금 본 내용을 요약하면...
 - 선언 읽는 순서에 따라 수식을 잘 적는다면(연산자를 순서대로 잘 붙인다면), '선언 내용들 중 내가 적은 부분'을 계산했을 때 얻을 수 있는 칸 / 값의 형식은 '선언 내용들 중 내가 안 적은 부분'과 동일해요
 - 적어도 declarator 부분에 대해서는 그래요.
앞에서 본 extern 같은 몇몇 specifier는 여기서는 취급 안 함
- 이 사실만 알고 있어도 C 선언을 좀 더 능동적으로 다룰 수 있을 거예요!

선언 읽는 법

- 조금 더 진행해 봅시다.

```
int *x();
```

선언 읽는 법

- 조금 더 진행해 봅시다.

int *x();

(인수는 뭔지 모르지만) int * 값을 return하는 함수 이름입니다.

선언 읽는 법

- 조금 더 진행해 봅시다.

int **X ;

선언 읽는 법

- 조금 더 진행해 봅시다.

int

**X

;

int * 포인터 변수 이름입니다?

선언 읽는 법

- 조금 더 진행해 봅시다.

int **X ;

int * 포인터 변수 이름입니다?

맞아요. 메모리 어딘가에 있을 'int * 한 칸'에 대한 **위치 값**을 담을 수 있어요.
(예를 들면 방금 본 'int * **배열**'의 한 칸을 골라 그 **위치 값**을 담아 둘 수 있어요)

선언 읽는 법

- 조금 더 진행해 봅시다.

```
int      x[3][2];
```

선언 읽는 법

- 조금 더 진행해 봅시다.

int x[3][2];

얘는 일단 세 칸짜리 배열이에요.

선언 읽는 법

- 조금 더 진행해 봅시다.

int x[3][2];

세 칸의 각 칸은 두 칸으로 되어 있고...

선언 읽는 법

- 조금 더 진행해 봅시다.

int x[3][2];

세 칸의 각 칸은 두 칸으로 되어 있고...
그 두 칸의 각 칸에는 int 값 하나가 담겨요.
그래서 결국 이름 x를 통해 총 여섯 칸을 쓸 수 있게 돼요.

선언 읽는 법

- 방금 규칙 하나가 또 나왔어요.
 - 선언을 읽을 때는 항상 '**이름**과 더 가까운 것'을 먼저 읽어요.
 - 선언 `int x[3][2];`에서는 `[3]`을 `[2]`보다 먼저 읽어요.
- 따라서, 방금 전 예시의 **이름**을 사용할 때...
 - 수식 `x`의 형식은 `int [3][2]` 형식 (`int` 기준으로는 아직은 여러 칸),
수식 `x[0]`의 형식은 `int [2]` 형식 (`int` 기준으로는 아직은 여러 칸),
수식 `x[0][0]`의 형식은 `int` 형식이라 말할 수 있어요!

선언 읽는 법

- 조금 더 진행해 봅시다.

```
int    (*x)[3]    ;
```

선언 읽는 법

- 조금 더 진행해 봅시다.

int (*x)[3] ;

굳이 괄호를 친 의도를 생각해 본다면,
여기서는 []보다 *이 먼저 붙는 게 맞을 거예요.

선언 읽는 법

- 조금 더 진행해 봅시다.

`int (*x)[3] ;`

그렇게 본다면 x는 'int [3]' **포인터 변수**로 읽을 수 있을 거예요.
메모리 어딘가에 있을 'int 세 칸짜리 **배열**'에 대한 **위치 값**을 담을 수 있어요.

선언 읽는 법

- 조금 더 진행해 봅시다.

```
int    (*X)()    ;
```

선언 읽는 법

- 조금 더 진행해 봅시다.

`int (*x)();`

여기서도 마찬가지로, x는 'int ()' 포인터 변수예요.
메모리 어딘가에 있는 'int 값 return하는 함수'에 대한 위치 값을 담을 수 있어요.
→ 보통 이 변수에 담을 만한 값을 '함수 포인터 값'이라고 축약해서 부르는 편이긴 해요.

선언 읽는 법

- 이제 마지막이에요!

```
int    (*x[3])();
```

선언 읽는 법

- 이제 마지막이에요!

int (*x[3])();

이제 애는 []가 가장 먼저 붙으므로 일단 (세 칸짜리) **배열**이에요.
배열의 각 칸은 int (*)() 형식이고, 이건 방금 전 예시의 **형식**과 동일해요.
개는 (한 칸짜리) **변수**였지만, 애는 (세 칸짜리) **배열**이지요!

선언 읽는 법


- 방금 나온 친구들 중 일부를 가져오면...

```
int      f      ();  
int      (*p     )();  
int      (*a[3])();
```

선언 읽는 법

- 방금 나온 친구들 중 일부를 가져오면...

int f ();
int f ();



이런 **형식** 뭔가가 존재할 때 '**애의 위치 값**'을 다루는 **형식**을 적으려면...

선언 읽는 법

- 방금 나온 친구들 중 일부를 가져오면...

```
int f();  
int (*p)();  
:  
*a[3])();
```

원래 선언의 이름 자리를 이런 식으로 치환해서 적기만 하면 돼요.
이제 수식 $p = \&f$ 는 형식 측면에서 유효해요.

선언 읽는 법

- 방금 나온 친구들 중 일부를 가져오면...

```
int      f      ();  
int      ( *p    ) ();  
int      ( *a[3] ) ();
```

비슷한 느낌으로, 원래 **형식**에 대한 세 칸짜리 **배열**을 선언하고 싶을 때도
이런 식으로 그 이름 자리를 치환해 주면 간단히 끝나요! (여기선 괄호는 불필요)
이제 수식 `a[0]`와 수식 `p`는 동일한 **형식**이라 볼 수 있어요.

선언 읽는 법

- 방금 나온 친구들 중 일부를 가져오면...

```
int      f      ();  
int      ( *p    ) ();  
int      ( *f2() ) ();
```

예시엔 없었지만, '원래 **형식 값**을 return하는 **함수**'를 선언하는 것도 가능해요.
이제 수식 f2()와 수식 p는 동일한 **형식**이라 볼 수 있어요.

선언 읽는 법

- 요약하면, 전체를 관망하면 노답인데, 조목조목 뜯어 보면 그럭저럭 쉬워요.
 - 세 가지 규칙을 일단 외워 두세요
 - 괄호 안 먼저
 - 가까운 것 먼저
 - 거리가 같으면 오른쪽 먼저
 - (중요)**이름**의 정체는 가장 먼저 붙는 기호에 의해 결정돼요
 - 따라서, 예를 들어 '저 선언에 대해 정의된 위치 값을 담아 둘 포인터 변수'를 선언하고 싶다면
저 선언의 **이름** 자리를 * 붙인 새 **이름**으로 치환해 적으면 돼요.
 - 붙는 순서를 따져서, 만약 필요하다면 괄호를 씌워야 할 수 있음!

선언 읽는 법

- 오오... 뭔가 규칙을 깨달은 것 같으니,
방금 열어 둔 파일에 아래 목표를 달성하기 위한 **선언**들을 직접 적어 봅시다:
 - int **변수** number
 - number에 대해 **정의된 위치 값**을 담기 위한 p_number
 - number와 동일한 **형식 값** 세 개를 담을 수 있는 a_number
 - number와 같은 **형식 값**을 return하는 f_number

선언 읽는 법

- 이런 느낌으로 적었다면 성공이에요!

```
int    number ;  
int    *p_number ;  
int    a_number [ 3 ] ;  
int    f_number ( ) ;
```

선언 읽는 법

- 배운 김에 몇 가지를 더 해 봅시다:
 - int **배열** arr (칸 수는 자유)
 - arr에 대해 **정의된 위치 값**을 담기 위한 p_arr
 - arr과 동일한 **형식 값** 세 개를 담을 수 있는 a_arr
 - arr과 같은 **형식 값**을 return하는 f_arr
 - int **값**을 return하는 **함수** func (인수는 자유)
 - func에 대해 **정의된 위치 값**을 담기 위한 p_func
 - func와 동일한 **형식 값** 세 개를 담을 수 있는 a_func
 - func와 같은 **형식 값**을 return하는 f_func
- 윗 덩어리는 지금 같이 해 볼게요. 아래 덩어리는 여러분이 직접 해 봐요

선언 읽는 법

- 어? 뭔가 이상합니다. 빨간 줄이 막 그어져요.

```
int arr[2];  
int(*p_arr)[2];  
int a_arr[3][2];  
int f_arr()[2];
```

```
int func();  
int(*p_func)();  
int a_func[3]();  
int f_func()();
```

선언 읽는 법

- 어? 뭔가 이상합니다. 빨간 줄이 막 그어져요.
 - 각 빨간 줄에 마우스 포인터를 갖다 대어 무슨 오류인지 확인해 봅시다.

```
int arr[2];  
int(*p_arr)[2];  
int a_arr[3][2];  
int f_arr()[2];
```

```
int func();  
int(*p_func)();  
int a_func[3]();  
int f_func()();
```

선언 읽는 법

- 적나라하게 '너 그러면 못 써'라고 말하는 것을 볼 수 있습니다.

 E0091

배열을 반환하는 함수를 사용할 수 없습니다.

 E0088

함수 배열을 사용할 수 없습니다.

 E0090

함수를 반환하는 함수를 사용할 수 없습니다.

선언 읽는 법

- 적나라하게 '너 그러면 못 써'라고 말하는 것을 볼 수 있습니다.
 - *은 별 무리 없이 잘 붙는 것 같은데, []와 ()은 서로 섞으면 안 되는 듯 보입니다.
 - ()의 경우에는 ()()조차도 막는 것 같아요.

 E0091

배열을 반환하는 함수를 사용할 수 없습니다.

 E0088

함수 배열을 사용할 수 없습니다.

 E0090

함수를 반환하는 함수를 사용할 수 없습니다.

선언 읽는 법

- 일단, C에서 **함수**는 '**값** 하나'를 return하도록 규정되어 있으므로, '여러 칸'을 상정하는 **배열**이나 아예 혼자 Code에 해당하는 **함수**를 return할 수 없는 것은 정상입니다.

선언 읽는 법

- 일단, C에서 **함수**는 '**값** 하나'를 return하도록 규정되어 있으므로, '여러 칸'을 상정하는 **배열**이나 아예 혼자 Code에 해당하는 **함수**를 return할 수 없는 것은 정상입니다.
 - **포인터**, **배열**과 달리 **함수**는 '(Data 답을) 칸'으로 표현되지 않아요
 - 달리 말하면, **함수**는 **형식**이 동일하다 하더라도 그 크기(아마도 명령어 전체 길이?)가 일정하지 않아요

선언 읽는 법

- '함수 배열'을 사용할 수 없게 둔 이유는 이따 4-3에서 실마리가 나옵니다. 그리고, 위 제약 사항을 극복하기 위한 방법 또한, 4-3에서 힌트가 나옵니다.
 - 일단 방금 전 선언들을 여러분 손으로 적을 수 있게 되었다면 그거로 충분해요

선언 읽는 법

- '함수 배열'을 사용할 수 없게 둔 이유는 이따 4-3에서 실마리가 나옵니다.
그리고, 위 제약 사항을 극복하기 위한 방법 또한, 4-3에서 힌트가 나옵니다.
 - 일단 방금 전 선언들을 여러분 손으로 적을 수 있게 되었다면 그거로 충분해요
- 그러니 일단은 조금 더 챙겨서,
C 수식을 납득하는데 매우 중요한 **lvalue**에 대해 좀 더 짚어 봅시다.
 - 애는 빨리 끝나요!

lvalue

- **Lvalue**는 옛날 C 설명서에도 나오는 전통 있는 속어입니다.
 - 엘-밸류 라고 발음해요
 - C++에서는 정식 명칭으로 쓰이고 있어요

lvalue

- **Lvalue**는 옛날 C 설명서에도 나오는 전통 있는 속어입니다.
 - 엘-밸류 라고 발음해요
 - C++에서는 정식 명칭으로 쓰이고 있어요
 - 아마 기력이 고갈되어 있을 듯 하니 이 부분은 예제 코드를 올려서 확인해 봅시다.
 - CSP_4_2_yeshi.c를 다운로드해서 VS에 탑재해 주세요

lvalue





- 뭐 이건...
강사가 오류 투성이 코드를 열어보라고 시켰군요.

lvalue

- 뭐 이건...
강사가 오류 투성이 코드를 열어보라고 시켰군요.
 - 각 빨간 줄에 마우스 포인터를 갖다 대어 뭐라고 말하는지 확인해 봅시다

Ivalue

- 이번에는 VS가 걱정하고 매크로 답변을 하고 있습니다.
 - 여기서의 '식'은 우리 수업에서는 수식이라 부르고 있어요
 - = 수식의 좌항 자리에 적은 수식들에 문제가 있는 것 같아요

 E0137	식이 수정할 수 있는 Ivalue여야 합니다.
 E0137	식이 수정할 수 있는 Ivalue여야 합니다.
 E0137	식이 수정할 수 있는 Ivalue여야 합니다.
 E0137	식이 수정할 수 있는 Ivalue여야 합니다.

lvalue

- 주석 내용이랑 함께 보면서 생각하면,
빨간 줄 적힌 부분들은 뭔가 논리적으로 말이 안 되는 듯 보입니다.
 - 상수 3을 위한 칸(메모리 공간)은 없음
 - 이미 즉시값으로써 명령어에 들어가 있을듯
 - arr은 명백히 '세 칸'에 대한 **이름**이지 '한 칸'에 대한 **이름**이 아님
 - func는 **함수 이름**이지 Data **이름**이 아님
 - func()는 **계산**하면 '값'이 나오지 '칸'이 나오는 게 아님
 - 지난번에 구경해 둔 것을 토대로 생각하면
func()의 return값은 메모리 위에 있는 게 아니라 CPU 손(레지스터)에 들려 있을 것임

lvalue





- 맞아요. **lvalue**라는 개념은 이런 의미를 내포하고 있어요:
 - 계산 결과값이 메모리 위 정확히 어디를 의미하는지 특정할 수 있는 **수식**
 - 주의: 명칭에 value가 들어 있긴 하지만 **lvalue**는 명백히 '**수식**'과 관련 있는 단어예요!
- 그래서, 이런 조건을 만족하는 **수식**을 '**lvalue 수식**'이라 부르는 편이에요.
 - '**수식의 형식**' 이야기를 했는데,
어떤 **수식**이 **lvalue 수식**인지 아닌지 여부를 논하는 것은
그 **수식**의 **형식**이 무엇인지 여부를 논하는 것과는 orthogonal해요
 - int **형식 lvalue 수식**이 존재할 수 있고,
int **형식 rvalue? 수식**이 존재할 수도 있어요

lvalue

- **Lvalue** 수식의 예를 들면...
 - Data, Code **이름**만 달랑 적은 것
 - **포인터 값** 나오는 수식에 * **연산자** 붙여둔 것
 - **배열 이름**에 [] **연산자** 붙여둔 것
...정도가 있어요
- 쉽게 본다면,
& **연산자**를 붙일 수 있는 수식은 **lvalue** 수식이라 생각해도 좋아요.
 - 속어인 만큼 동네마다 룰이 조금씩 다를 수 있기는 해요

lvalue

- 음 그렇다면, 왜 굳이 VS는 '수정할 수 있는'이라는 토를 달아 둔 것일까요?

 E0137	식이 수정할 수 있는 lvalue여야 합니다.
 E0137	식이 수정할 수 있는 lvalue여야 합니다.
 E0137	식이 수정할 수 있는 lvalue여야 합니다.
 E0137	식이 수정할 수 있는 lvalue여야 합니다.

const

- 음 그렇다면, 왜 굳이 VS는 '수정할 수 있는'이라는 토를 달아 둔 것일까요?
 - 여러분이 원한다면 어떤 이름을 '수정할 수 없는' 이름으로 만들 수 있기 때문이에요!

 E0137 식이 수정할 수 있는 lvalue여야 합니다.

 E0137 식이 수정할 수 있는 lvalue여야 합니다.

 E0137 식이 수정할 수 있는 lvalue여야 합니다.

 E0137 식이 수정할 수 있는 lvalue여야 합니다.

const

- main() 정의 안에 적혀 있는 **선언**을 아래와 같이 고쳐 봅시다:

// 이제 number는 int 한 칸에 대한 이름이에요.

const int number = 3;

// 이제 arr은 int 세 칸에 대한 이름이에요.

// (배열에 대한 initializer를 이런 식으로 적을 수 있어요. 애는 수식은 아님!)

const int arr[] = { 3, 3, 3 };

각 **선언**에 const specifier를 달고 있어요.

const

- 오... 이제 이 코드는 완벽히 망한 것 같아요.
 - 그 어떤 = 수식도 유효하지 않아요

// 상수 값 3 자리에 5 담기

3 = 5;

// number 자리에 5 담기

number = 5;

// arr 자리에 5 담기

arr = 5;

// 'arr에 대한 여러 칸들 중 0번째 칸' 자리에 5 담기

arr[0] = 5;

// 함수 func 자리에 5 담기

func = 5;

// 함수 func의 return값 자리에 5 담기

func() = 5;

const

- 맞아요. const specifier는 '수정할 수 없는'으로 만들어요.
- 좀 더 정밀하게 보면,
이제 이 **이름**을 가지고는 int **형식** 칸에 담긴 **값**을 수정할 수 없게 돼요.
 - 뭔가 Python에서 본 '변경 가능성'이 떠오른다면, 그게 맞긴 한데, 일단 지금은 참아용

const

- 맞아요. const specifier는 '수정할 수 없는'으로 만들어요.
- 좀 더 정밀하게 보면,
그 **이름**을 가지고는 그 **형식** 칸에 담긴 **값**을 수정할 수 없게 돼요.
 - 뭔가 Python에서 본 '변경 가능성'이 떠오른다면, 그게 맞긴 한데, 일단 지금은 참아용
- (중요)'수정'만 불가능할 뿐, initialize는 여전히 가능해요.
 - 사실, Data **이름**에 const를 붙인다면
꼭 initializer까지 적어 주어야 의미를 갖게 될 거예요!
 - 정상적인 방법으로는 나중에 = **수식** 써서 다른 **값**을 담는 게 불가능해요

마무리

- 좋아요. 여기까지 보면
여러분이 새로운 C 선언을 적을 때 신경 써야 할 요소들은
거의 다 다룬 셈이 돼요.
 - const 이야기는 4-3에서 좀 더 구경해 볼게요
- 잠시 쉬었다가,
이번엔 프로그래머 쪽이 아닌 컴파일러 쪽 이야기를 구경해 보도록 합시다.