

4-1. 선언 적을 때 고려해야 할 것들 (1)

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

3일차 내용

- 선언을 본격적으로 다루기 시작했어요.
 - Specifier, declarator, initializer로 구성돼요. 생략하거나 변형할 수 있긴 해요
 - 이름 + 사용 방법을 정해요
- 약간 중의적으로 보일 수 있는 정의도 구경해 보았어요.
 - 내용을 정의: 이름 정하는(선언 적는) 프로그래머가 해요
 - 위치 상수 값을 정의: 컴파일러가 해요
- 선언의 declarator를 꾸밀 수 있는 세 가지 방법을 확인했어요.
 - *, [], ()가 있었어요
 - 사실 C에는 이것들 말고도 더 있긴 해요. C++에는 좀 더 많이 있어요

3일차 내용

- 어떤 **변수 이름**에 대해 우리가 얻을 수 있는 **값** 세 가지를 확인했어요.
 - 사용 빈도 1위: 그냥 **이름** 적으면 value of
 - 사용 빈도 2위: & **연산자** 붙이면 address of
 - 사용 빈도 3위: sizeof **연산자** 붙이면 size of
- 위의 두 **연산자**는 **이름**을 사용할 때 붙일 수 있는 것들이고, 그래서 사실은 **수식**에 붙여(서 더 큰 **수식**을 만들어)요.
 - '연산자'니까 당연
 - 아무튼 우리는 **수식** sizeof 3을 유효하게 적을 수도 있어요

3일차 내용

- = **수식** 이야기가 꽤 자주 등장했어요.
 - Data 흐름 중 write를 담당해요
 - '어디에' '무엇을' 담아야 할 때 적어요
- 일단 어디에 담을 것인지를 먼저 정해 놓고(정해서 담아 놓고)
= **수식 계산** 자체는 나중에 하고 싶을 때,
우리는 해당 칸에 대한 **포인터 값을 포인터 변수에** 담아 뵈다가
나중에 사용할 수 있어요.
 - 여기서 말하는 '나중'에는 다른 함수에게 = **수식 계산**을 부탁하는 상황도 있어요

오늘 내용

- 소입설 내용 복습 마지막 날이에요.
- 복습을 마무리하기 전에 몇 가지 요소들을 더 짚어보고 퇴근하려 해요.
 - 선언 적을 때 고려해야 할 것들
 - 4-1, 4-2로 나누어 두었어요
 - 포인터 변수, 배열, 함수 이름에 대한 컴파일러의 deduction 방식
- 최종 목표는 없어요. 대신 예시 코드가 좀 많아요.
- 내용이 두툼하긴 하지만... 한 번 천천히 구경해 보도록 합시다.

이번 시간에는

- 선언 적을 때 고려해야 할 것들 (1)
 - 선언 및 (위치) 정의와 연관되어 있는 6칸짜리 표를 구경해 봐요
 - 다음 슬라이드부터 바로 시작해요
 - 중간에 이런저런 곁가지 이야기들이 등장할 예정이에요
- 가벼운 마음으로 시작해 봅시다

6칸짜리 표

- 빠른 진행을 위해 바로 들고 와 봤어요:

| 분류 기준 | 옵션 1 | 옵션 2 |
|-------------------------|--------------|--------------|
| 이름의 모듈(파일)간 visibility | 외부(external) | 내부(internal) |
| 정의되는 위치의 특성 | static | automatic |
| 이름의 모듈(파일) 내 visibility | global | local |

6칸짜리 표

- 빠른 진행을 위해 바로 들고 와 봤어요:

| 분류 기준 | 옵션 1 | 옵션 2 |
|-------------------------|--------------|--------------|
| 이름의 모듈(파일)간 visibility | 외부(external) | 내부(internal) |
| 정의되는 위치의 특성 | static | automatic |
| 이름의 모듈(파일) 내 visibility | global | local |

- Python 다루던 앞 수업에서 본 것도 있고, 처음 나온 것도 있어요

잠시, Python 수업 내용 회상

- Python에는 '**이름** 사전'이라는 실물이 존재합니다.
 - 모듈(파일)마다, 그리고 매 **함수 실행**마다 서로 다른 **이름** 사전을 가지고 있었어요

잠시, Python 수업 내용 회상

- Python에는 '**이름** 사전'이라는 실물이 존재합니다.
 - 모듈(파일)마다, 그리고 매 **함수 실행**마다 서로 다른 **이름** 사전을 가지고 있었어요
- C에서도 대충 비슷합니다.
 - 어떤 **이름**에 대한 **선언**을 어디에, 어떻게 적어 두었는지에 따라 어떤 **문장**을 적고 있는 내가 여기서 그 **이름**을 사용할 수 있는지 여부가 달라집니다
 - 내가 지금 키보드 치고 있는 모듈(.c 파일) / 다른 모듈
 - 내가 지금 **문장** 적고 있는 **함수 정의** / 다른 **함수 정의**
/ 어떤 **함수 정의** 중괄호 안에도 속하지 않는 **선언**일 수도
 - 그냥 **선언** / 뭔가 키워드(specifier)를 붙여 둔 **선언**

global 이름과 local 이름

- 일단 쉬운 것부터 본다면,
{ } 바깥에서 선언한(선언이 어떤 { }에도 포함되지 않는) **이름**은 global,
{ } 안에서 선언한 **이름**은 local입니다.
- 글자 수 절약을 위해 '선언'을 동사처럼 쓰고 있는데,
'선언한 이름'을 '(그) 선언을 적어 새로 도입한 이름'으로 봐 주면 좋을 것 같아요

global 이름과 local 이름

- 일단 쉬운 것부터 본다면,
{ } 바깥에서 선언한(선언이 어떤 { }에도 포함되지 않는) 이름은 global,
{ } 안에서 선언한 이름은 local입니다.
 - Local 이름은 그 이름을 선언한 { } 안에서만 볼 수 있어요
 - 다른 함수, 다른 모듈에선 그 이름을 못 봐요(사용할 수 없어요)
 - 예외적으로,
함수 정의 괄호 안에 적는 인수 이름은 그 함수 정의의 { } 안에서 선언한 것으로 간주해요
- Global 이름은 이러한 제약 조건이 없어요

global 이름과 local 이름

- 일단 쉬운 것부터 본다면,
{ } 바깥에서 선언한(선언이 어떤 { }에도 포함되지 않는) 이름은 global,
{ } 안에서 선언한 이름은 local입니다.
 - Local 이름은 그 이름을 선언한 { } 안에서만 볼 수 있어요
 - 다른 함수, 다른 모듈에선 그 이름을 못 봐요(사용할 수 없어요)
 - 예외적으로,
함수 정의 괄호 안에 적는 인수 이름은 그 함수 정의의 { } 안에서 선언한 것으로 간주해요
- Global 이름은 이러한 제약 조건이 없어요
- 이것까진 Python에서랑 거의 비슷해 보이지요?
예시 코드를 보면...

global 이름과 local 이름

```
#include <stdio.h>

int common = 0;

void fill_with_three(int *p)
{
    *p = 3;
}

int main()
{
    int number = 0;

    fill_with_three(&number);
    fill_with_three(&common);

    return 0;
}
```

global 이름과 local 이름

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

```
int common = 0;
```

```
void fill_with_three(int *p)
{
    *p = 3;
}
```

```
int main()
{
    int number = 0;

    fill_with_three(&number);
    fill_with_three(&common);

    return 0;
}
```

굵은 글씨 **이름**들은 모두 global **이름**입니다.

global 이름과 local 이름

```
#include <stdio.h>

int common = 0;

void fill_with_three(int *p)
{
    *p = 3;
}

int main()
{
    int number = 0;

    fill_with_three(&number);
    fill_with_three(&common);

    return 0;
}
```

같은 모듈 안에서는, ('미리' 선언해 두기만 했다면)
그 global 이름을 사용할 수 있습니다.

(함수 정의 첫 줄 보면 '함수 선언'에 적을만한 모든 것들이 다 있으니 인정)

global 이름과 local 이름

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

```
int common = 0;
```

```
void fill_with_three(int *p)
{
    *p = 3;
}
```

```
int main()
{
```

```
    int number = 0;
```

```
    fill_with_three(&number);
    fill_with_three(&common);
```

```
    return 0;
```

```
}
```

반면, 함수 정의의 인수 이름은 local 이름입니다.

{ } 안에서 선언한 이름도 local 이름입니다.

global 이름과 local 이름

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

```
int common = 0;
```

```
void fill_with_three(int *p)
{
    *p = 3;
}
```

따라서, 여기 있는 p는 이 { } 안에서만 볼 수 있습니다.

```
int main()
```

```
{
    int number = 0;
    fill_with_three(&number);
    fill_with_three(&common);

    return 0;
}
```

따라서, 여기 있는 number는 이 { } 안에서만 볼 수 있습니다.

global 이름과 local 이름

- 우리가 특정 **이름**을 사용하면,
컴파일러는 그 **이름**에 대한 **선언**이 어디 적혀 있는지 찾고,
그 **선언**에 대해 자기가 **정의**해 둔 **위치 상수 값**을 바탕으로
적절한 Code를 만들어 줍니다.
 - 다음 슬라이드 예시를 살펴 볼게요

global 이름과 local 이름

```
#include <stdio.h>

int number = 3;

void func()
{
    number = 0;
}

int main()
{
    int number = 5;

    func();
    printf("%d\n", number);

    return 0;
}
```

global 이름과 local 이름

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

```
int number = 3;
```

```
void func()
```

```
{
```

```
    number = 0;
```

```
}
```

```
int main()
```

```
{
```

```
    int number = 5;
```

```
    func();
```

```
    printf("%d\n", number);
```

```
    return 0;
```

```
}
```

이 코드에는 **이름** number에 대한 선언이 두 곳에 적혀 있고,
이들은 각각 다른 **위치 값**을 갖도록 **정의**되어 있습니다.
(**이름** 글자는 같지만 그에 대해 **정의된 위치 값**이 다르기 때문에
이 둘은 서로 다른 **변수 이름**입니다)

global 이름과 local 이름

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

```
int number = 3;
```

```
void func()
```

```
{
```

```
    number = 0;
```

```
}
```

```
int main()
```

```
{
```

```
    int number = 5;
```

```
    func();
```

```
    printf("%d\n", number);
```

```
    return 0;
```

```
}
```

여기서 사용한 number 는...

global 이름과 local 이름

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

```
int number = 3;
```

```
void func()
```

```
{
```

```
    number = 0;
```

```
}
```

```
int main()
```

```
{
```

```
    int number = 5;
```

```
    func();
```

```
    printf("%d\n", number);
```

```
    return 0;
```

```
}
```

여기서 사용한 number는...
{ } '깊이' 기준으로 더 가까운 여기에 선언된 number입니다.

저 위에 선언된 global 이름 number는
{ } 하나 밖에 있기 때문에 더 멀어요.

global 이름과 local 이름

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

```
int number = 3;
```

```
void func()
```

```
{
```

```
    number = 0;
```

```
}
```

```
int main()
```

```
{
```

```
    int number = 5;
```

```
    func();
```

```
    printf("%d\n", number);
```

```
    return 0;
```

```
}
```

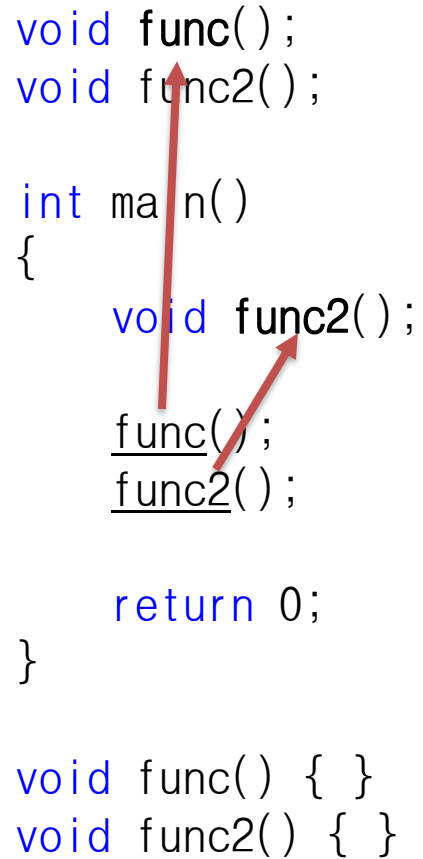
반면 여기서 사용한 number는
선언이 하나밖에 안 보이기 때문에 이렇게 연결되지요!

global 이름과 local 이름

```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```

global 이름과 local 이름

```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```



이번에는 **함수 이름**입니다.
규칙은 아까랑 동일해요.

global 이름과 local 이름

```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```

다만, 일반적인 **변수 선언**과 달리 **함수 선언**의 경우에는
그 **함수**의 내용물이 무엇인지 프로그래머가 직접 **정의**해야 합니다.

변수야 뭐 '그 형식 한 칸'으로서 내용물이 간단하지만
어떤 **함수**를 구성하는 **문장**들이 무엇일지는
그 **함수 선언**만 읽어가지고는 컴파일러는 파악할 수 없어요.

global 이름과 local 이름

```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```

그래서 별도의 **함수 정의** 문법을 통해
프로그래머가 수동으로 적어 주어야 하지요.

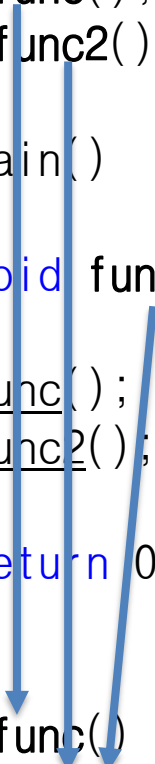
global 이름과 local 이름

```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```

우리가 별로 자각하지는 않고 있었지만, 컴파일러는, 각 선언의 이름들이 어떤 위치를 나타내는지를 직접 판단해서 그 둘을 '연결'해 줍니다.

global 이름과 local 이름

```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```



우리가 별로 자각하지는 않고 있었지만, 컴파일러는, 각 선언의 이름들이 어떤 위치를 나타내는지를 직접 판단해서 그 둘을 '연결'해 줍니다.

이 코드의 경우 이렇게 연결되어 있어요!

global 이름과 local 이름

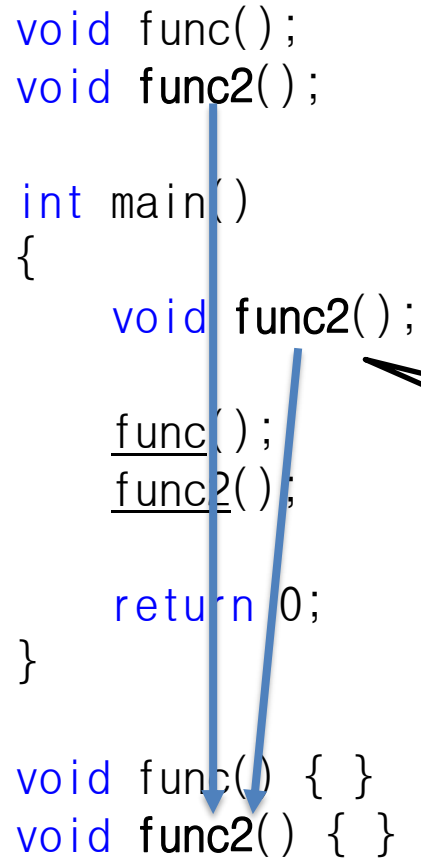
```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```

그리고 방금 전 봤던
'여기서 사용한 **이름**은 어디에 **선언**된 **이름**인가'와,
'여기서 **선언**하는 **이름**은 어디에 **정의**된 **위치**와 연결되어 있는가'는
모두 컴파일러에 의해 잘 체크됩니다.
(컴파일 도중 수식으로써 적어 둔 Data **이름**, Code **이름**을 다룰 때
그 **이름**에 대한 **위치 상수 값**을 명령어 안에 담거나 하게 돼요)

이 '연결' 작업을 컴파일과 구분하여 '링크'라고 부를 때도 있기는 한데,
요즘 컴퓨터들은 용량이 넉넉하니 뭐 지금은 그러려니 해도 좋아요.

global 이름과 local 이름

```
void func();  
void func2();  
  
int main()  
{  
    void func2();  
  
    func();  
    func2();  
  
    return 0;  
}  
  
void func() { }  
void func2() { }
```



아무튼 그래서 실질적으로 이 두 선언은
이름(및 사용 방법)도 같고 연결된 **위치**도 같은,
다시 말하면 '동일한 **함수**'에 대한 **선언**이라고 말할 수 있지요!

global 이름과 local 이름

- 요점은 이렇습니다:
 - 수식으로써 적어 둔 **이름**이 있을 때, 컴파일러는 그 **이름**에 대한 **선언**을 찾아요
 - **선언된 이름**들은, 그 **이름**에 대한 **위치**와 연결돼요
 - 컴파일러가 모든 **선언**에 대한 **위치 값**을 그 자리에서 **정의**하는 건 아니에요(예: 함수 선언)
 - '정의를 수반하지 않는 선언'도 존재해요
 - **이름**(및 사용 방법)도 같고 연결된 **위치**도 같은 두 선언은 '동일한 Data / Code **이름 선언**'이 돼요
 - 여기까지 컴파일러가 자동으로 해 주기 때문에, 우리는 **선언**해 둔 **이름**을 사용함으로써 **정의**해 둔 **위치 값**을 얻고 활용할 수 있어요!
 - 그리고, '모든 선언이 정의를 수반하는 것은 아니다'를 확인했다면 다음 개념들을 이해하기 조금 더 수월해질 거예요

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의된 값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
 - Storage class라고도 불러요
 - 사실 다른 옵션들도 있는데 지금은 이 둘만 봅시다

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 일단, 모든 global **이름**들은 static **위치**를 가집니다. (예외 없음)

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 일단, 모든 global **이름**들은 static **위치**를 가집니다. (예외 없음)
 - 위치가 static하다는 것은...
 - 프로그램 실행하는 내내 항상 동일한 **위치 값**을 가짐
 - 따라서, 그 메모리 영역은 항상 이 **이름**만을 위해 사용됨
(다른 **이름**에 대해 해당 영역이 점유되지 않음)
...을 의미해요

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 그렇다면, 'automatic 위치'는 어떤 것을 의미하는 걸까요?

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 그렇다면, 'automatic 위치'는 어떤 것을 의미하는 걸까요?
 - 프로그램 실행 도중 필요에 따라
해당 **이름**에 대한 칸이 '자동'으로 마련되는 것을 의미합니다

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 그렇다면, 'automatic 위치'는 어떤 것을 의미하는 걸까요?
 - 프로그램 실행 도중 필요에 따라
해당 **이름**에 대한 칸이 '자동'으로 마련되는 것을 의미합니다
 - Static 위치를 갖는 **변수**(static 변수)와 비교했을 때...
 - Automatic 변수들은 실제 위치가 프로그램 실행 도중 그 때 그 때 달라질 수 있습니다
 - Automatic 변수들을 위한 메모리 영역(칸)들은,
그 **변수 이름**이 선언된 { }의 **실행**이 끝나면 '그 변수 것'이 아니게 됩니다

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 그렇다면, 'automatic 위치'는 어떤 것을 의미하는 걸까요?
 - 프로그램 실행 도중 필요에 따라
해당 **이름**에 대한 칸이 '자동'으로 마련되는 것을 의미합니다
 - Static 위치를 갖는 **변수**(static 변수)와 비교했을 때...
 - Automatic 변수들은 실제 위치가 프로그램 실행 도중 그 때 그 때 달라질 수 있습니다
 - Automatic 변수들을 위한 메모리 영역(칸)들은,
그 **변수 이름**이 선언된 { }의 실행이 끝나면 '그 변수 것'이 아니게 됩니다
 - 아무튼, 요약하면 시한부 변수라 할 수 있습니다.

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 기본적으로 local Data 선언들 중 '(위치) 정의를 하는' 선언들은
automatic **위치**를 가집니다.

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 기본적으로 local Data 선언들 중 '(위치) 정의를 하는' 선언들은
automatic **위치**를 가집니다.
 - 규칙이 조금 길지요?

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의**된 **값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 기본적으로 local Data 선언들 중 '(위치) 정의를 하는' 선언들은 automatic 위치를 가집니다.
 - 특별한 specifier를 써서 '기본'을 깰 수 있습니다
 - 특별한 specifier를 써서 '**정의를 안 하게**' 할 수 있습니다

static 위치와 automatic 위치

- 이 둘은,
그 **이름**에 대한 **위치 값**이 어떻게 **정의**되는지,
그 **정의된 값**으로 특정되는 메모리 영역이 얼마나 유효한지를 구분합니다.
- 기본적으로 local Data 선언들 중 '(위치) 정의를 하는' 선언들은 automatic 위치를 가집니다.
 - 특별한 specifier를 써서 '기본'을 깰 수 있습니다
 - Local **변수 선언**을 static int x;와 같이 static specifier를 붙여 적으면,
그 **변수**는 static 위치를 갖도록 **정의**됩니다
 - 특별한 specifier를 써서 '**정의를 안 하게**' 할 수 있습니다
 - **변수 선언**을 extern int x;와 같이 extern specifier를 붙여 적으면,
그 **변수**는 이 **선언**을 가지고는 **정의**되지 않습니다. (다른 **선언**을 통해 **정의**되어 있어야 함)

static 위치와 automatic 위치

- 예시 코드를 잠깐만 보고 갈게요.
 - 조금 길어서, CSP_4_1_yeshi.c에 담아 두었어요
 - 집에서 직접 실행해 보는 것을 추천해요!
 - 피보나치 수열의 규칙 세 가지, 다들 알지요?
 - 규칙 1: $\text{fib}[1] = 1$ 이다
 - 규칙 2: $\text{fib}[2] = 1$ 이다
 - 규칙 3: $\text{fib}[n] = \text{fib}[n - 2] + \text{fib}[n - 1]$ 이다. ($n > 2$ 일 때)
 - 예제 코드의 `fib()`도, 이러한 규칙에 따라서 필요한 경우 자기 자신을 다시 한 번 호출하도록 구성되어 있어요
 - 그리고 이 함수는 매 번 호출할 때마다 자신이 사용하는 이름들에 대한 위치 값을 출력해요
 - static 위치로 정의된 이름들과 automatic 위치로 정의된 이름들에 & 연산자를 붙여 계산할 때, 매 호출마다 어떤 결과값이 나오는지 한 번 확인해 봅시다!

static 위치와 automatic 위치

- 요점은 이렇습니다:
 - static 위치에 정의되는 이름들에 대한 위치 값은 프로그램 실행 내내 항상 동일해요

static 위치와 automatic 위치

- 요점은 이렇습니다:
 - automatic 위치에 정의되는 Data 이름들은...
 - 그 함수가 호출될 때 정확한 위치 값이 정해져요
 - 그 함수 내용물 실행이 끝나기 전까지는
그 위치 값으로 특정되는 칸은 '이번 호출에서의' 그 이름 것으로 유지돼요
 - 함수 정의 내용물 실행이 끝나면 그 위치는 더 이상 그 변수 것이 아니게 되어요
 - 예시에서는, fib(2) 끝난 바로 다음에 fib(1)을 다시 호출하면
이전에 다 쓰고 놔 둔 바로 그 영역을 다시 활용하는 것을 볼 수 있었어요
 - (중요)'지금 호출되지 않은' 함수의 automatic 변수들은 아무 위치도 점유하고 있지 않아요
 - 프로그램 안에 함수 정의가 100개 1000개 있다 해도,
메모리 영역은 '호출되었고, 아직 내용물 실행이 끝나지 않은' 함수들만 사용해요
 - 어떤 시점에 한 함수에 대한 '아직 안 끝난 함수 호출'이 여러 번 중첩되어 있을 수 있기는 해요

static 위치와 automatic 위치

- 요점은 이렇습니다:

- automatic 위치에 정의되는 Data 이름들은...

- 그 함수가 호출될 때 위치가 지정돼요

이런 복잡한 작업이 컴파일러가 만든 Code에 의해 자동으로 수행되기 때문에, 그리고 그 당시 프로그래밍 언어들 중에서는 이런 서비스를 제공하는 언어가 몇 없었기 때문에, C에서는 엘리컨트하게 'automatic'이라는 용어를 써서 이 기능을 표현해 놓았어요!

- 함

(그러니 언급할 때 존중을 담아 조금 신경써서 발음해 보도록 합시다)

- (중요)'지금 호출되지 않은' 함수의 automatic 변수들은 아무 위치도 점유하고 있지 않아요

- 프로그램 안에 함수 정의가 100개 1000개 있다 해도, 메모리 영역은 '호출되었고, 아직 내용물 실행이 끝나지 않은' 함수들만 사용해요
 - 어떤 시점에 한 함수에 대한 '아직 안 끝난 함수 호출'이 여러 번 중첩되어 있을 수 있기는 해요

external 이름과 internal 이름

- 이 둘은 **이름**이라고 적혀 있기는 한데,
정밀하게는 **정의**(연결)와 연관되어 있는 친구들입니다.

external 이름과 internal 이름

- 이 둘은 **이름**이라고 적혀 있기는 한데,
정밀하게는 **정의**(연결)와 연관되어 있는 친구들입니다.
- External **이름**은 '다른 모듈에서 **선언**하여 사용할 수도 있는' **이름**이고,
internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.

external 이름과 internal 이름

- 이 둘은 **이름**이라고 적혀 있기는 한데, 정밀하게는 **정의**(연결)와 연관되어 있는 친구들입니다.
- External **이름**은 '다른 모듈에서 **선언**하여 사용할 수도 있는' **이름**이고, internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.
- 일단 모든 external **이름**은 반드시 일반적인 global **선언**에 의해 **정의**되어 있어야 합니다.
 - local **이름**은 애초에 '같은 모듈의 다른 **함수 정의** 내용물 안'에서도 못 보니 다른 모듈에서 못 보는 것은 당연해요

external 이름과 internal 이름

- Python에 있던 global문을 잠시 회상해 보면...
 - 절대로 **실행** 도중 이 **이름**을 내(함수) **이름** 사전에 등재하지 말 것을 미리 부탁해 둬

external 이름과 internal 이름

- Python에 있던 global문을 잠시 회상해 보면...
 - 절대로 **실행** 도중 이 **이름**을 내(**함수**) **이름** 사전에 등재하지 말 것을 미리 부탁해 둬
- C에서는 local **선언**을 아예 안 적으면 이런 일은 발생하지 않지만...

external 이름과 internal 이름

- Python에 있던 global문을 잠시 회상해 보면...
 - 절대로 **실행** 도중 이 **이름**을 내(**함수**) **이름** 사전에 등재하지 말 것을 미리 부탁해 둬
- C에서는 local **선언**을 아예 안 적으면 이런 일은 발생하지 않지만...
'이 모듈에서 선언한 **이름**'과 '다른 모듈의 **이름**'은 명백히 구분합니다.

external 이름과 internal 이름

- Python에 있던 global문을 잠시 회상해 보면...
 - 절대로 **실행** 도중 이 **이름**을 내(**함수**) **이름** 사전에 등재하지 말 것을 미리 부탁해 둬
- C에서는 local **선언**을 아예 안 적으면 이런 일은 발생하지 않지만...
'이 모듈에서 선언한 **이름**'과 '다른 모듈의 **이름**'은 명백히 구분합니다.
 - Python에서는 '풀 네임'을 부름으로써 '저 모듈의 저 **이름**'을 지칭할 수 있었어요
- C에서는 '모듈 **이름**' 개념은 존재하지 않기에,
다른 모듈에 있는 **이름**을 사용할 때도 그냥 원래 **이름**을 적어요
 - 물론 그 **이름**을 사용하려면 '내 모듈'에 '저 **이름**'에 대한 **선언**을 적어 주어야 하고,
이 때는 extern specifier를 사용해서 내 **선언**을 '정의를 수반하지 않는 **선언**'으로 만들어 뒀요
→ 컴파일러가 나중에 저 모듈에서 **정의**된 위치 값과 연결해 줘요

external 이름과 internal 이름

- 다시 돌아와서,
External **이름**은 '다른 모듈에서 **선언**하여 사용할 수도 있는' **이름**이고,
internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.
- 더 정확히는,
'다른 모듈에서 **선언**하여, 여기서 **정의**한 **위치**와 연결할 수 있는',
또는,
'이 모듈에서 **선언**하여, 다른 모듈에 **정의**된 **위치**와 연결할 수 있는'
...**이름**을 external **이름**이라 합니다.

external 이름과 internal 이름

- 다시 돌아와서,
External **이름**은 '다른 모듈에서 선언하여 사용할 수도 있는' **이름**이고,
internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.
- 더 정확히는,
'다른 모듈에서 선언하여, 여기서 정의한 위치와 연결할 수 있는',
또는,
'이 모듈에서 선언하여, 다른 모듈에 정의된 위치와 연결할 수 있는'
...**이름**을 external **이름**이라 합니다.
 - printf(), scanf()가 딱 그런 친구들이었죠?
다음 슬라이드에서 살짝 정리해 볼게요

external 이름과 internal 이름

- 우리가 적어 왔던 첫 줄의 비밀
 - `#include <stdio.h>`를 적어 두면
'헤더 파일' `stdio.h`에 적혀 있는 글자들을 전부 여기에 복붙해 줘요
 - `stdio.h`에는 `printf()` 등에 대한 **선언**이 적혀 있고,
이 때 `printf`는 external **이름**으로써 **선언**되어 있어요
 - 동네에 따라 조금씩 다르긴 한데 그냥 이렇다 칩시다
 - 컴파일러는 '우리가 적는 모듈(`main.c`)'에 적혀 있는 **선언**과
다른 어딘가에 있을 **정의**(된 **위치**)를 연결해 줘요
 - 그 덕분에 우리는 **이름** `printf`를 우리 목표에 맞게 사용할 수 있었어요!

external 이름과 internal 이름

- 다시 돌아와서,
External **이름**은 '다른 모듈에서 **선언**하여 사용할 수도 있는' **이름**이고,
internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.
- 더 정확히는,
'다른 모듈에서 **선언**하여, 여기서 **정의**한 **위치**와 연결할 수 있는',
또는,
'이 모듈에서 **선언**하여, 다른 모듈에 **정의**된 **위치**와 연결할 수 있는'
...**이름**을 external **이름**이라 합니다.
 - printf(), scanf()가 딱 그런 친구들이었죠?
다음 슬라이드에서 살짝 정리해 볼게요

external 이름과 internal 이름

- 다시 돌아와서,
External **이름**은 '다른 모듈에서 **선언**하여 사용할 수도 있는' **이름**이고,
internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.
- 그런데,
다른 모듈에서 **선언**만 해 버리면 여기 **정의된 위치**를 연결해간다는 건데,
그 말은, 누군가가 내 모듈에 **선언/정의된 변수 이름**만 알면
그 **변수** 자리에 담긴 **값**을 지 맘대로 바꿀 수 있다는 것을 의미해요.

external 이름과 internal 이름

- 다시 돌아와서,
External **이름**은 '다른 모듈에서 **선언**하여 사용할 수도 있는' **이름**이고,
internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.
- 그런데,
다른 모듈에서 **선언**만 해 버리면 여기 **정의된 위치**를 연결해간다는 건데,
그 말은, 누군가가 내 모듈에 **선언/정의된 변수 이름**만 알면
그 **변수** 자리에 담긴 **값**을 지 맘대로 바꿀 수 있다는 것을 의미해요.
 - 그게 싫다면, `static int x;`와 같이 **선언함**으로써
선언하는 이름이 internal **이름**이 되도록 강제할 수 있어요
 - Global이면서 internal한 **선언**을 적을 수 있어요

external 이름과 internal 이름

- 다시 돌아와서,
External **이름**은 '다른 모듈에서 **선언**하여 사용할 수도 있는' **이름**이고,
internal **이름**은 '이 모듈에서만 사용할 수 있는' **이름**을 말합니다.
- 그런데,
다른 모듈에서 **선언**만 해 버리면 여기 **정의된 위치**를 연결해간다는 건데,
그 말은, 누군가가 내 모듈에 **선언/정의된 변수 이름**만 알면
그 **변수** 자리에 담긴 **값**을 지 맘대로 바꿀 수 있다는 것을 의미해요.
 - 그게 싫다면, `static int x;`와 같이 **선언함**으로써
선언하는 **이름**이 internal **이름**이 되도록 강제할 수 있어요
 - Global이면서 internal한 **선언**을 적을 수 있어요
 - 이걸 아까 나온 specifier긴 해요. C에서는 뜻이 두 가지임!

일단 잠시 휴식

- 아마 이쯤 오면 다들 정신줄 놓고 있을 것 같아요.
 - 특별히 specifier를 붙여 **선언**하지 않는 이상은
global - static - external 조합 또는
local - automatic - internal 조합을 사용하게 되니,
보통은 그냥 평소 부르던 대로 '로컬 **변수**'와 같이 불러도 되긴 해요
 - 뭐 그렇기는 하지만, C++ 넘어가면 조금 더 복잡해질 예정이니
방금 본 6칸짜리 표는 지금 시점에 외워 두고 가면 좋을 것 같아요
- 일단 잠시 쉬었다가 4-2에서 이어서 진행해 보도록 합시다.