

## 3-2. 선언과 정의

창의적소프트웨어프로그래밍  
2022년도 여름학기  
Racin

# 이번 시간에는

- 선언 둘러보기
  - 선언을 어떻게 적을 수 있는지, 어떤 의미를 갖는지 확인해요
- 사잇단계: 수식과 문장
  - 수식과 문장을, 적는 프로그래머의 의도 측면, 읽는 컴파일러 측면에서 구분해 봅니다
  - 코드 적는 시점, compile time, **runtime**의 차이를 짚어 봅니다
    - 나름 중요한 키워드인 상수가 등장할 예정
- 정의
  - 단어 정의에 내포된 두 가지 의미를 확인해 봐요
  - '어떤 이름에 대해 얻을 수 있는 값' 세 가지를 확인해 봐요

# 선언 둘러보기

---

- 일단은 단출하게 시작해 볼까요?

# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

```
int number;  
  
float rate;  
  
char choice;
```

# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

```
int number;
```

```
float rate;
```

```
char choice;
```

이 세 줄은 모두 선언입니다.

# 선언 둘러보기

---

- Python에서는,  
(이름 사전에)**이름**이 적절히 등재되지 않은 상태에서  
(수식으로써 적어 둔)그 **이름**을 **계산**하면  
NameError가 뜨는 것을 관찰할 수 있습니다.
- C에서도,  
**선언**되어 있지 않은 **이름**은 사용할 수 없습니다.

# 선언 둘러보기

- VS에서도 거의 즉시 빨간 줄을 쳐 줘요!

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

```
int main()
```

```
{
```

```
    number = 3;
```

```
    return 0;
```

```
}
```

식별자 "number"이(가) 정의되어 있지 않습니다.

# 선언 둘러보기

---

- Python에서는 '이름 사전 변경 권한'을 가진 문장들이 존재해서, 그 **문장을 실행**하며 사전에 **이름**을 등재합니다.
  - 기본적으로 '가장 가까운' 사전에 등재해요
  - 이를 다르게 의도하는 방법도 존재하긴 해요
- 반면 C에서는,  
**이름**을 사용하기 위해 반드시 **선언**을 적어 두어야 합니다.



# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

```
number = 3
```

```
print str(number) + '입니다.'
```

```
int number;
```

```
number = 3;
```

```
printf("%d 입니다.", number);
```

# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

number는 이 줄 다음부터 사용할 수 있습니다.

```
number = 3
```

```
print(str(number) + '입니다.')
```

number는 이 줄 다음부터 사용할 수 있습니다.

```
int number;
```

```
number = 3;
```

```
printf("%d 입니다.", number);
```

# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

위에 다른 코드가 없는 경우,  
만약 이 **문장**이 없다면 오류가 납니다!

```
#number = 3
```

```
print(str(number) + '입니다.')
```

위에 다른 코드가 없는 경우,  
만약 이 **선언**이 없다면 오류가 납니다!

```
/*int number;*/
```

```
number = 3;
```

```
printf("%d 입니다.", number);
```

# 선언 둘러보기

---

- 일단 이 쪽은 '**이름을 선언**' 정도만 가지고 있으면 될 듯 해요!
  - Python에서도 **이름**은 프로그래머가 자신의 의도를 반영하여 적는 것이었어요
  - C **선언** 또한 우리의 **이름** 사용 계획을 직접 적는 것이라 생각하면 돼요
  - 여기서 중요한 언급이 하나 나왔는데...

# 선언 둘러보기

---

- (중요)선언은,  
이 **이름**이 어떤 의미를 갖는지(어떻게 사용할 수 있는지 등)를  
정확하게 설명합니다.

# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

```
int number;  
  
float rate;  
  
char choice;
```

# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

`int number;`

number는 int **형식 변수의 이름**입니다.

`float rate;`

rate는 float **형식 변수의 이름**입니다.

`char choice;`

choice는 char **형식 변수의 이름**입니다.

# 선언 둘러보기

- 아래 코드를 살펴 봅시다:

`int number;`

수식 `number`를 계산하면,  
`int` 형식 값을 얻을 수 있습니다.

`float rate;`

수식 `rate`를 계산하면,  
`float` 형식 값을 얻을 수 있습니다.

`char choice;`

수식 `choice`를 계산하면,  
`char` 형식 값을 얻을 수 있습니다.



# 선언 둘러보기

- 이번엔 이 코드를 한 번 살펴 봅시다:

```
int number;
```

```
int func();
```

# 선언 둘러보기

- 이번엔 이 코드를 한 번 살펴 봅시다:

```
int number;
```

number는 int **형식 변수의 이름**입니다.

```
int func();
```

func는 '인수는 뭐 몇 개 담는지 모르겠지만  
int **형식 값을 return하는 함수**'의 **이름**입니다.

# 선언 둘러보기

- 이번엔 이 코드를 한 번 살펴 봅시다:

```
int number;
```

수식 `number`를 계산하면,  
`int` 형식 값을 얻을 수 있습니다.

```
int func();
```

수식 `func()`를 계산하면,  
`int` 형식 값을 얻을 수 있습니다.

# 선언 둘러보기

---

- 오... 뭔가 감이 오는 듯 합니다.
  - 시작하는 단계긴 하지만, 우리는 **선언**을 더 쉽게 이해하기 위한 비법을 배웠어요!
  - 여기까지 나온 내용을 다음 슬라이드에서 살짝 정리해 볼게요

# 선언

- 지금까지 살펴 본 **선언(declaration)**의 의미
  - '**이름을 선언**'해요
    - 이 **이름**이 무슨 **이름**인지를 **선언**해요
      - **변수 이름**이라면, 어떤 **형식 값** 담을 **변수 이름**인지를 같이 적게 돼요
  - 어떤 **이름**을 **수식**으로써 적었을 때의 계산 방법은 그 **이름**에 대한 **선언**에 의해 결정돼요
    - 일단은, **선언**에 적혀 있는 대로 **수식**을 적으면, **선언**에 적어 둔 그 **형식 값**을 얻을 수 있어요
- 기본적으로, **선언**되어 있지 않은 **이름**은 **수식**으로써 적을 수 없어요

# 선언

- 지금까지 살펴 본 **선언**(declaration)의 의미
  - '이름을 선언'해요
    - 이 **이름**이 무슨 **이름**인지를 선언해요
      - 변수 이름이라면, 어떤 **형식 값** 담을 변수 이름인지를 같이 적게 돼요
  - 어떤 **이름**을 수식으로써 적었을 때의 계산 방법은 그 **이름**에 대한 **선언**에 의해 결정돼요
    - 일단은, **선언**에 적혀 있는 대로 수식을 적으면, **선언**에 적어 둔 그 **형식 값**을 얻을 수 있어요

- 기본적으로

이 부분을 조금 더 단단하게 가져가기 위해  
**선언** 적는 방법을 간단하게 살펴보고 갈게요

# 선언 둘러보기

- 선언은 이렇게 세 부분으로 구성됩니다:



# 선언 둘러보기

- 선언은 이렇게 세 부분으로 구성됩니다:
  - 각 자리에 적절한 내용을 적으면 돼요!

Specifier(s)	Declarator	=	Initializer	;
int	number	=	3	;
double	rate	=	3.5	;



# 선언 둘러보기

- 선언은 이렇게 세 부분으로 구성됨

- 각 자리에 적절한 내용을 적으면 됨

이 자리에 적는 내용은,  
지금 당장 유추 가능한 그 효과를 낸다고 봐도 좋아요.

Specifier(s)	Declarator	=	Initializer	;
int	number	=	3	;
double	rate	=	3.5	;

# 선언 둘러보기

- 여기서 약간의 변형이나 생략을 할 수도 있어요.

Specifier(s)	Declarator	=	Initializer	;
int	number	=	3	;
double	rate	=	3.5	;

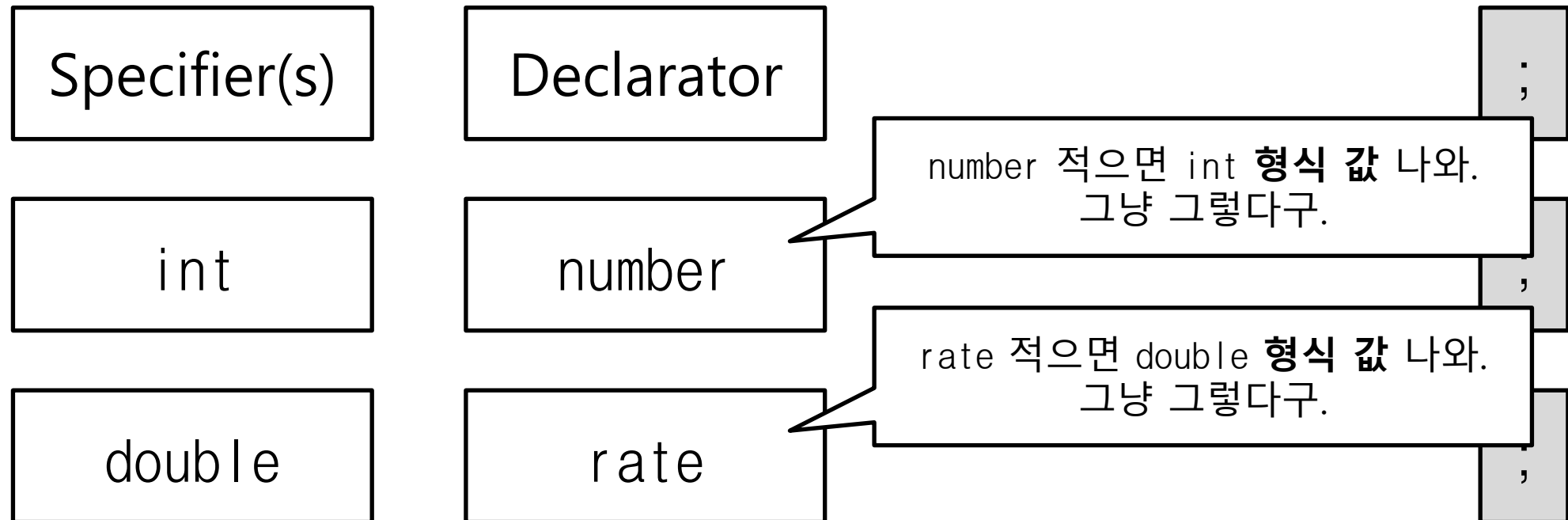
# 선언 둘러보기

- 여기서 약간의 변형이나 생략을 할 수도 있어요.
  - '= Initializer' 부분은 생략해도 좋아요.

Specifier(s)	Declarator	
int	number	;
double	rate	;

# 선언 둘러보기

- 여기서 약간의 변형이나 생략을 할 수도 있어요.
  - '= Initializer' 부분은 생략해도 좋아요.



# 선언 둘러보기

- 여기서 약간의 변형이나 생략을 할 수도 있어요.
  - 이건 비밀인데, declarator 부분도 사실 생략해도 좋아요.

Specifier(s)

;

int

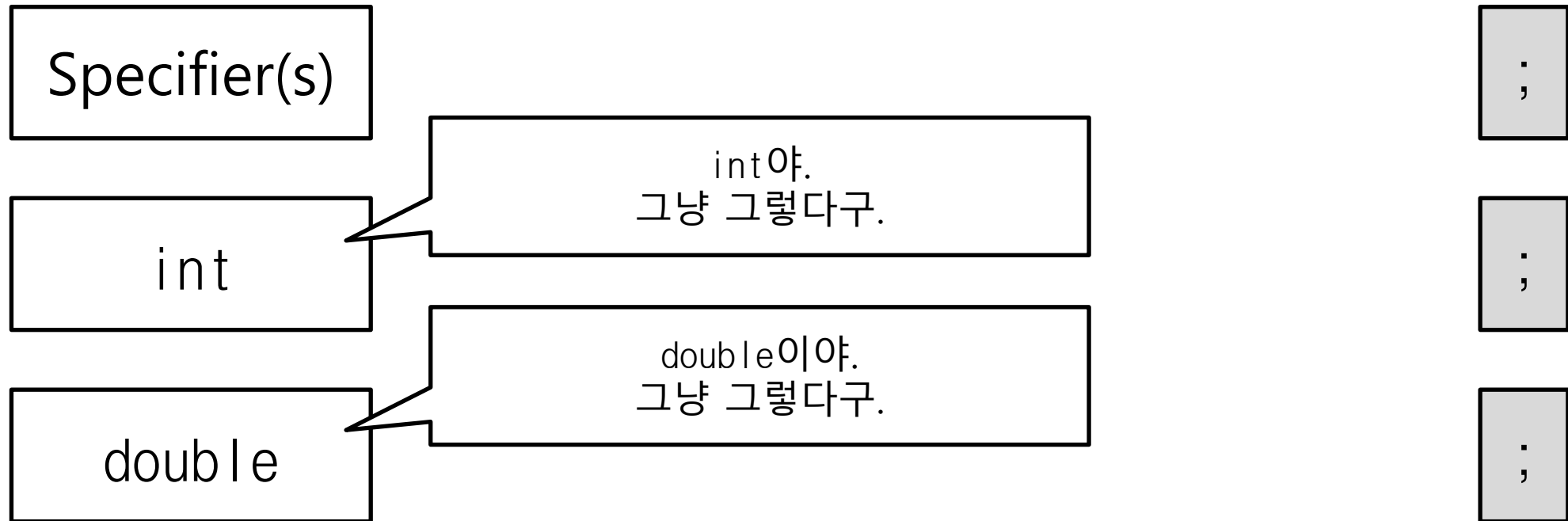
;

double

;

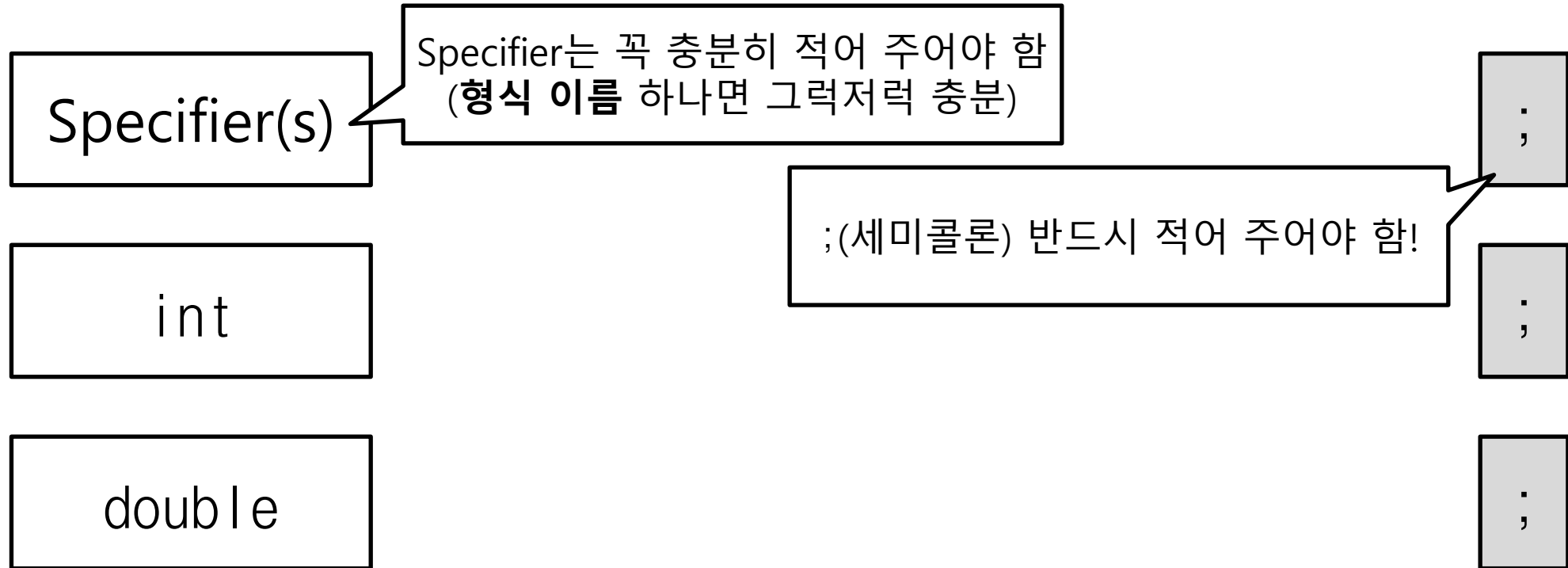
# 선언 둘러보기

- 여기서 약간의 변형이나 생략을 할 수도 있어요.
  - 이건 비밀인데, declarator 부분도 사실 생략해도 좋아요.



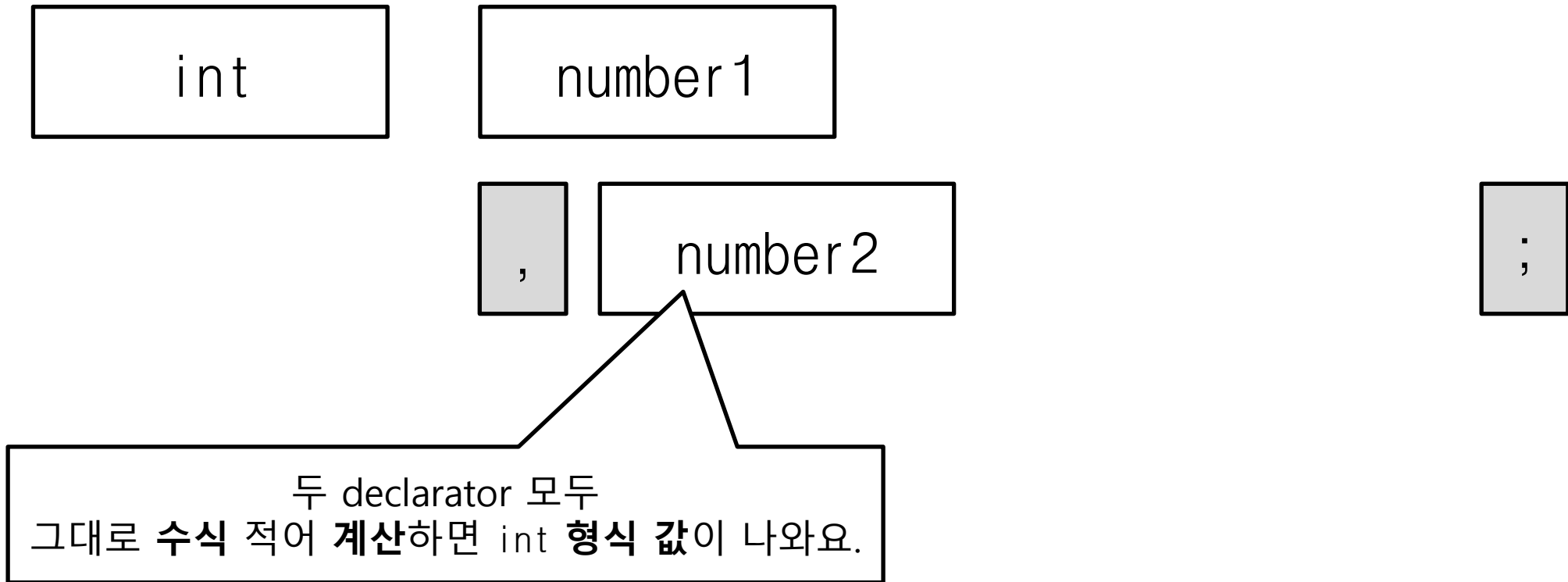
# 선언 둘러보기

- 여기서 약간의 변형이나 생략을 할 수도 있어요.
  - 이게 가장 간단한 버전이에요. 이 둘은 절대로 생략할 수 없어요!



# 선언 둘러보기

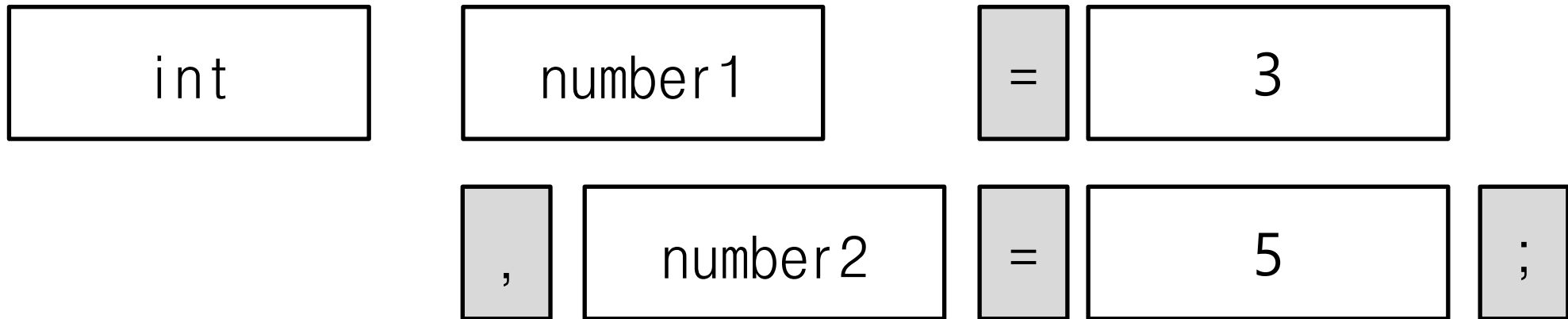
- 여기서 약간의 변형이나 생략을 할 수도 있어요.
  - 반면, 여러 declarator들을 한 **선언**에 몰아 적을 수 있어요. ( ,를 써요 )





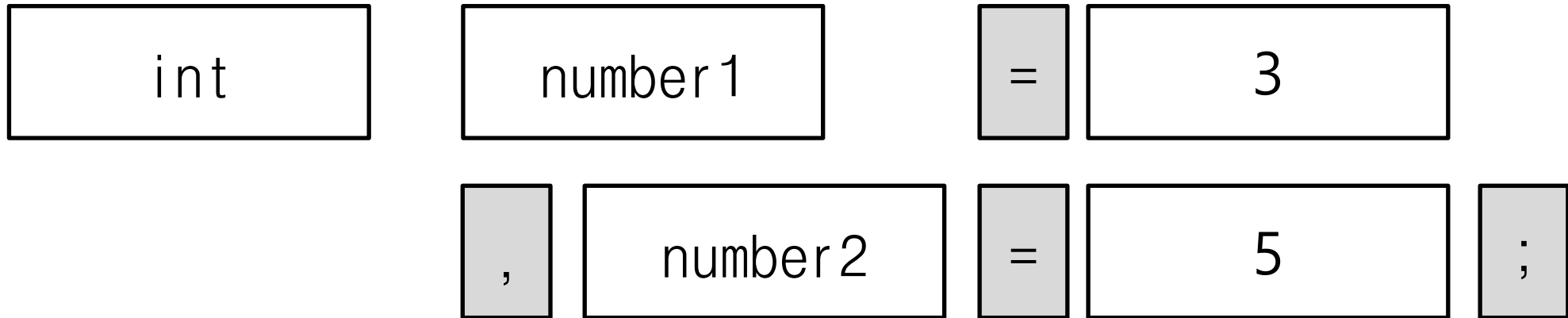
# 선언 둘러보기

- 여기서 약간의 변형이나 생략을 할 수도 있어요.
  - 각 declarator들에 대한 initializer들을 붙일 수도 있어요!



# 선언 둘러보기

- 살짝 주의할 점! 여기서의 =이랑 ,는 연산자가 아니에요.
  - 비슷한 느낌으로, declarator 또한 '이렇게 수식으로 써 적으면 됨'을 의미하지만 그 자체가 수식인 것은 아니에요



# 선언 둘러보기

- 여기까지 등장한 여러 **선언** 용법들을 모아 왔어요:

```
int;
```

```
int number;
```

```
double rate = 3.5;
```

```
int func();
```

```
int choice, five = 5;
```

# 선언 둘러보기

- 여기까지 등장한 여러 **선언** 용법들을 모아 왔어요:

```
int;
```

```
int number;
```

```
double rate = 3.5;
```

```
int func();
```

```
int choice, five = 5;
```

이 친구들은 모두 유효한 C **선언**들이에요!

이 **선언**은 declarator가 func()예요.  
단순히 **이름**만 적지 않고 ()를 추가로 붙어 봤어요.  
아무튼 유효한 C **선언**이 맞아요.

# 선언 둘러보기

- 방금 봤던 세 영어 명칭들, specifier, declarator, initializer는 일단 지금은 이름만 알아 두어도 좋아요!
  - 일단 지금은 이들 중 주로 declarator랑 initializer쪽을 탐험하고, 복습 끝자락 및 그 이후에 specifier쪽을 더 맛볼 예정이에요
- 대신, 아까 정리해 본 두 가지는 지금 꼭 기억해 주세요:
  - '이름을 선언'
  - Declarator에 적어 둔 느낌대로 수식 적으면 specifier로 특정되는 **형식 값**이 나옴!
- 대충 구경해 보니 이 둘은 그럭저럭 납득 가능했지요?

# 선언 둘러보기 마무리

- 이따 실습 할 때,  
지금보다 좀 더 다양한 **선언**을 적는 방법을 구경해 볼게요.
- 지금은 일단 **정의** 이야기로 들어가기 위해  
작은 사잇단계를 넘어가 봅시다
  - 이번에도 VS의 디버그 기능을 사용할 것 같아요

(이전 수업자료 스샷을 그대로 가져와서 VS 생김새가 조금 달라요. 양해해 주세요)

# 수식과 문장

---

- 우리 수업 첫 시간부터 갑자기  
'수식과 문장은 구분해서 봐야 한다'고 말한 적이 있는데...  
그래서 이 둘은 무슨 '차이'를 가지고 있는 것일까요?

# 수식과 문장

---

- 우리가 배운 범위 내에서 이 둘을 구분해 보면...

- 수식은 계산돼요

문장은 실행돼요

- 수식은 연산자를 써서 더 큰 수식을 만들 수 있어요

어떤 문장은 내용물(문장들)을 가질 수 있어요(다른 문장들에 비해 좀 더 커요)



# 수식과 문장

---

- 우리가 배운 범위 내에서 이 둘을 구분해 보면...

- 수식은 계산돼요

문장은 실행돼요

- 수식은 연산자를 써서 더 큰 수식을 만들 수 있어요

어떤 문장은 내용물(문장들)을 가질 수 있어요(다른 문장들에 비해 좀 더 커요)

- ...구분이 잘 안 되는 게 정상이긴 해요

# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다:

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다:

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

이 짧은 코드에서 수식에 해당하는 친구만 표시

이 짧은 코드에서 수식에 해당 안 하는 친구만 표시

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다:

if문의 조건식은 '분기 방법'을 의미하지요?  
**runtime**에 애 **계산**한 결과값에 따라 '다음 **문장**'이 달라져요.  
(**'다음 문장'**은 노란 화살표로 봤던 개 맞아요)

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

갑자기 등장한 **runtime**은  
'프로그램을 실행(run) 하는 시점'을 의미해요.  
(지난 시간에 다루었어야 하는데 깜빡함)

```
else  
    number = +number;
```

= **수식**은 그 자체가 Data 흐름 구성의 기본이에요.  
**runtime**에 노란 화살표가 여기로 오면, 이 **수식**이 **계산**될 거예요.  
아마도 '절대값 담기' 의도를 달성하고 싶은 듯 해요.

# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다:

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

그럼 애네들은 뭘까요?  
그냥 꺾데기만 남아 있는 것 같은데...

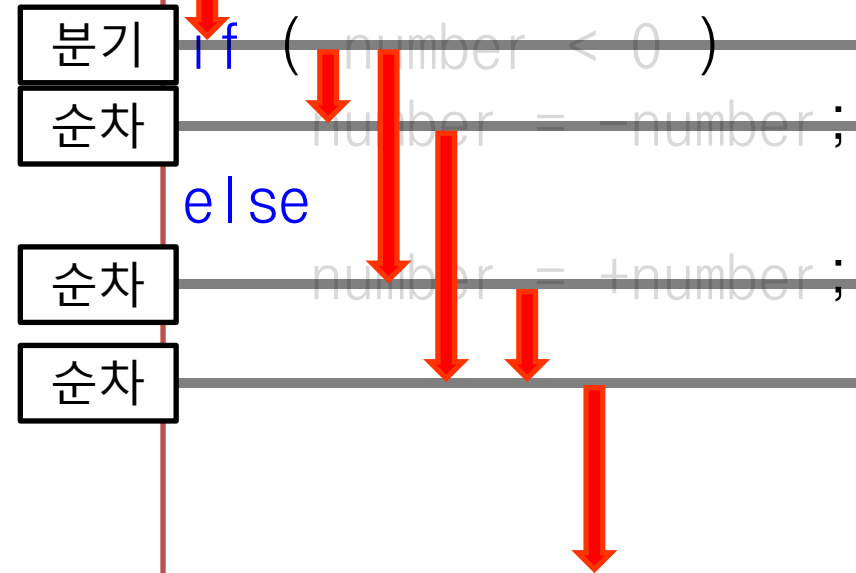
```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

맞아요. 적어도 C의 **문장**들은  
(**'수식을 계산'** 기능을 제외한다면)  
전부 다 '노란 화살표' 그 자체의 갈 길을 정하기 위해 쓰여요!

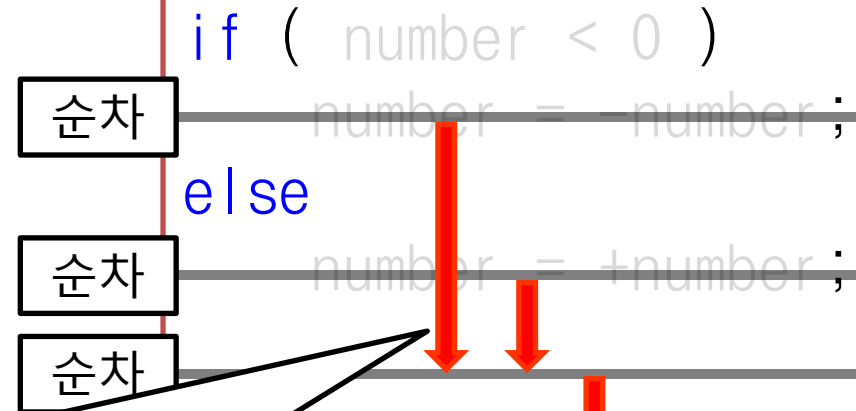


# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다

모든 문장들은 runtime에 실행될 '다음 문장', 즉, '이 문장을 실행한 후의 노란 화살표 위치'가 고정되어 있어요.

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```



물론 '다음 문장'이 정확히 어디인지에 대한 정보는 그 문장 자체에 적혀 있는 것은 아니지요. context(또 나옴)에 의해 유추될 뿐이에요.

# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```

그리고 바로 if문의 if부분이,  
'다음 문장'을 **runtime**에 결정할 수 있는 가능성을 갖는 친구고,  
이게 바로 프로그래밍의 **분기** 개념의 실체예요.  
(여전히 'if문의 다음 문장'은 고정되어 있지만,  
if문의 내용물 **문장**들에 대한 **실행** 흐름은 유동적이에요)

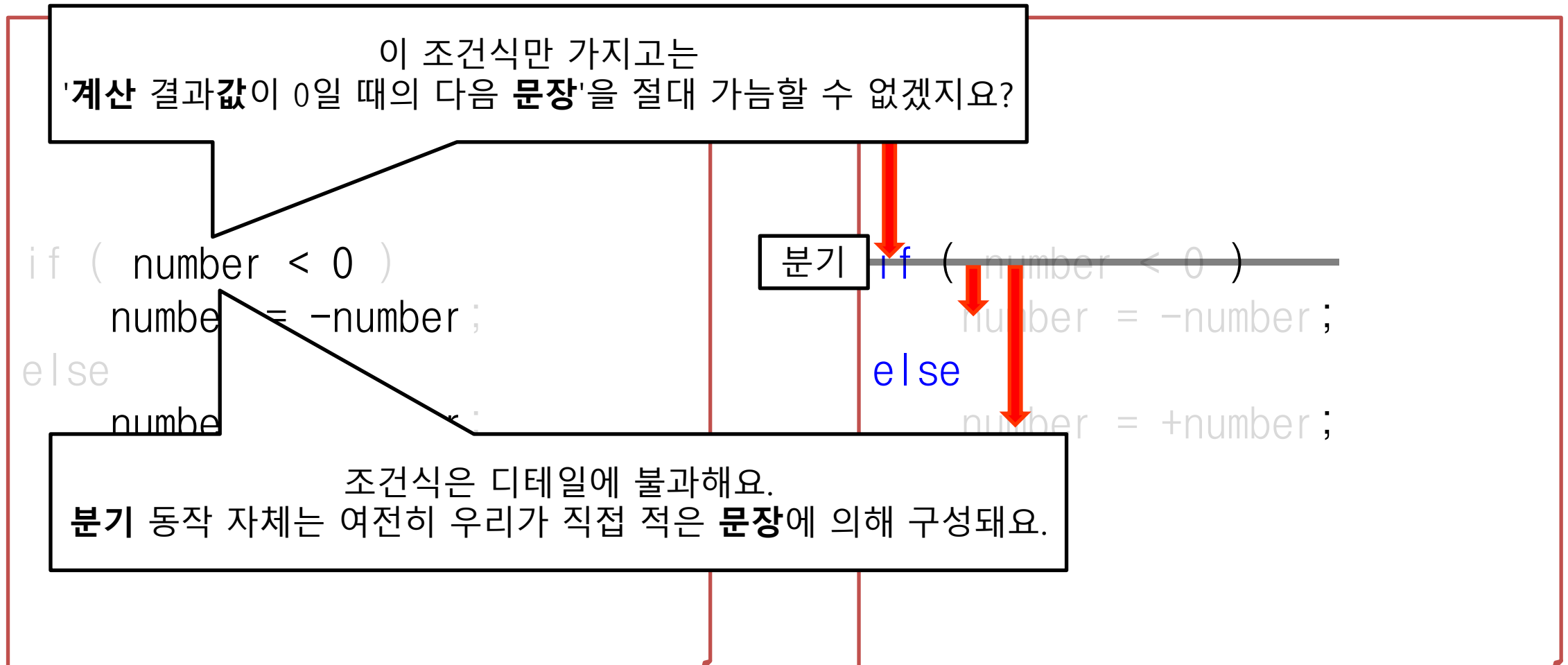
분기

```
if ( number < 0 )  
    number = -number;  
else  
    number = +number;
```



# 수식과 문장

- 아래 예시를 잠시 구경해 봅시다:



# 컴파일러가 보는 수식과 문장

- 음... 뭔가 감이 오는 듯 하니 VS를 켜고 간단한 실습을 해 봅시다.
- 아래 내용들이 추가로 등장할 예정이에요:
  - '노란 화살표'의 실체(스포일러: 애도 값임!)
  - Code '숫자'
  - 상수 개념
- 매우 유익한 복습이 될 테니 같이 ㄱㄱ해 보는 것을 권장해요

# 컴파일러가 보는 수식과 문장

- 일단 적당한 프로젝트 / .c 파일을 만들거나 열고 아래 내용을 입력해 봅시다:

```
int Three() { return 3; }

int main() {
    int number, choice;

    number = 5;

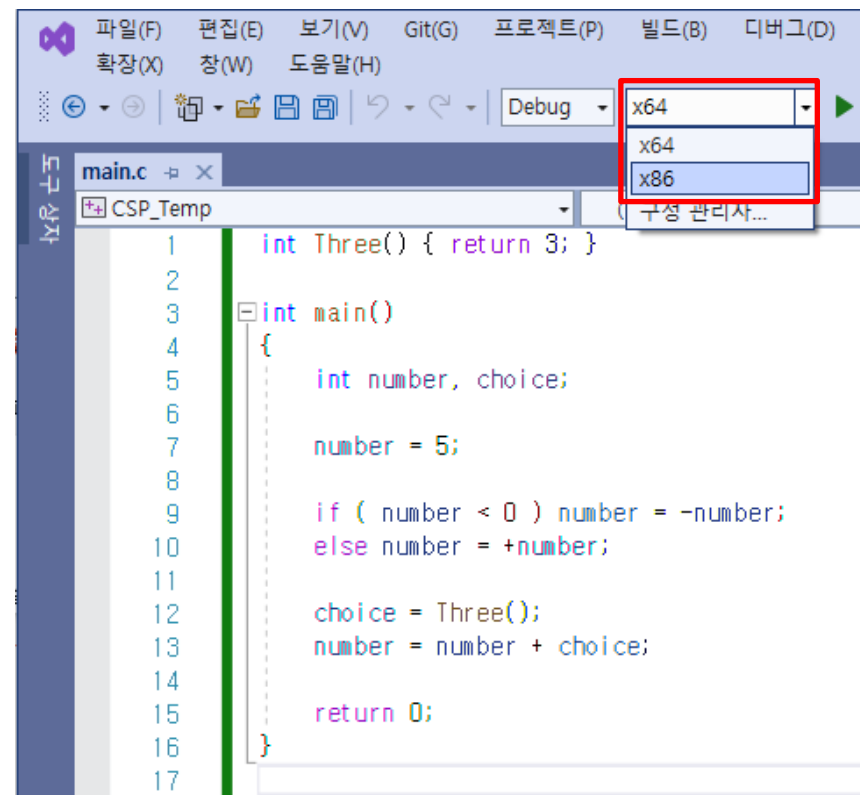
    if ( number < 0 ) number = -number;
    else number = +number;

    choice = Three();
    number = number + choice;

    return 0;
}
```

# 컴파일러가 보는 수식과 문장

- 그 다음, 잠시 편의를 위해  
메뉴 바의 녹색 화살표 옆에 적혀 있는 x64를 눌러  
x86으로 변경해 둡시다.
  - 이제 당분간 우리 프로그램을  
'32비트 프로그램'으로 만들어 줄 거예요



# 컴파일러가 보는 수식과 문장

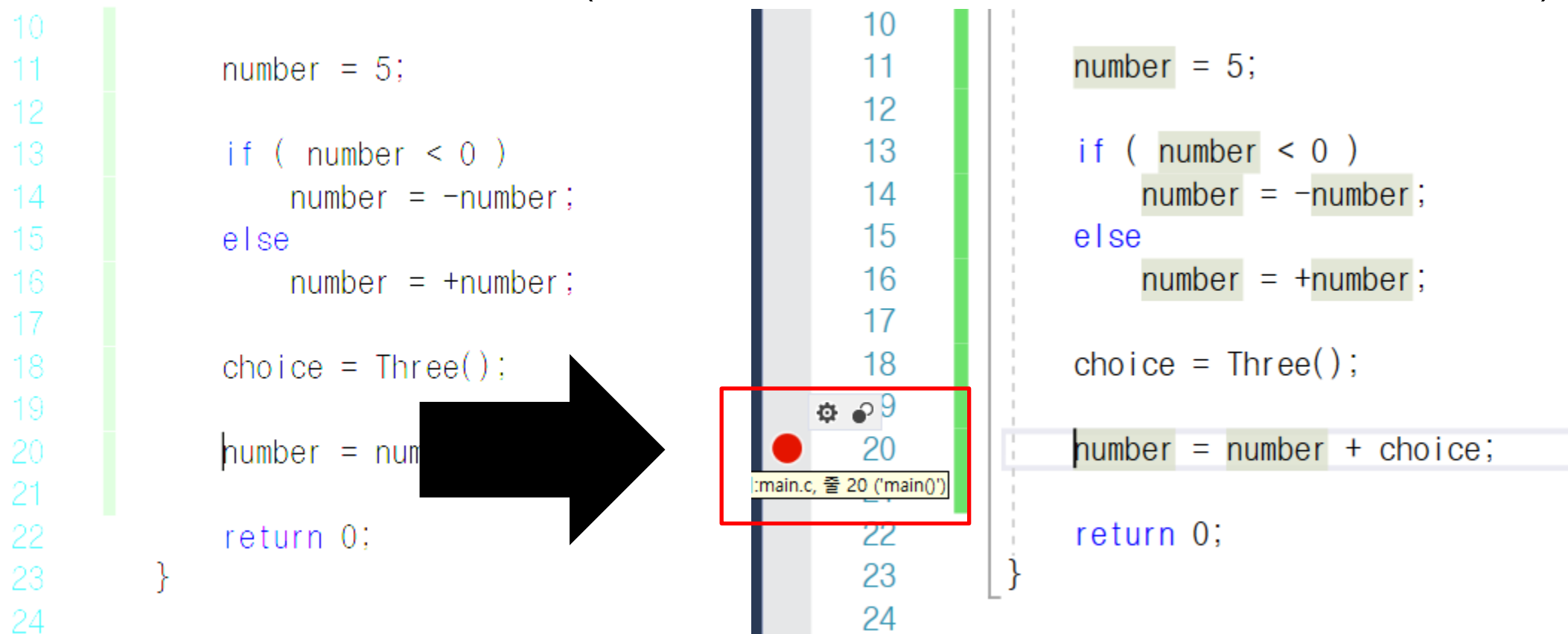
- 변경했다면, 아래와 같이 커서를 위치시킨 다음, F9를 눌러 봅시다.

```
10
11  number = 5;
12
13  if ( number < 0 )
14      number = -number;
15  else
16      number = +number;
17
18  choice = Three();
19
20  number = number + choice;
21
22  return 0;
23  }
24
```

클릭하든 해서 커서를 이 줄에 두고 F9!

# 컴파일러가 보는 수식과 문장

- 변경했다면, 아래와 같이 커서를 위치시킨 다음, F9를 눌러 봅시다.
  - 오오... 빨간 점이 콕 박혔습니다! (F9가 안 먹는 친구들은 스샷의 저 부분을 클릭해도 됨)



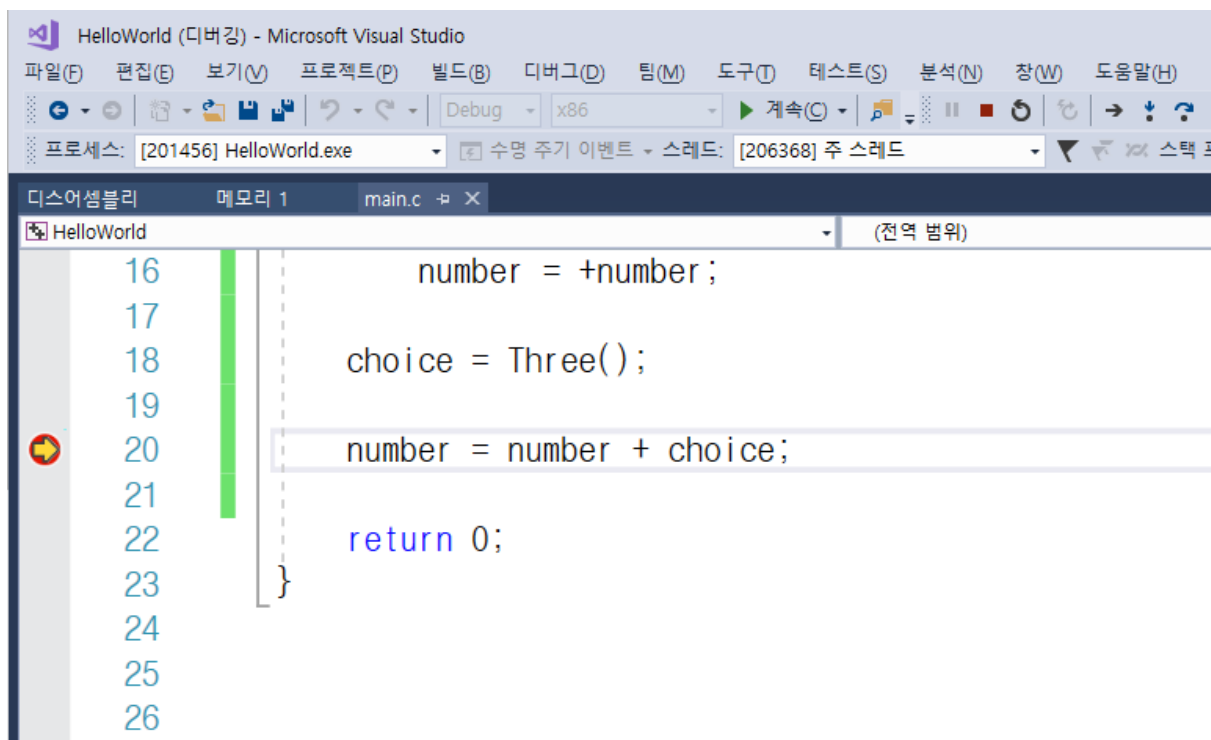
# 컴파일러가 보는 수식과 문장

---

- 빨간 점을 찍어 둔 상태에서 F5를 눌러 봅시다.
  - Ctrl + F5나 F10이 아님!

# 컴파일러가 보는 수식과 문장

- 오오... '디버깅'이 시작되더니, 노란 화살표가 빨간 점 자리에 멈춰 있습니다.



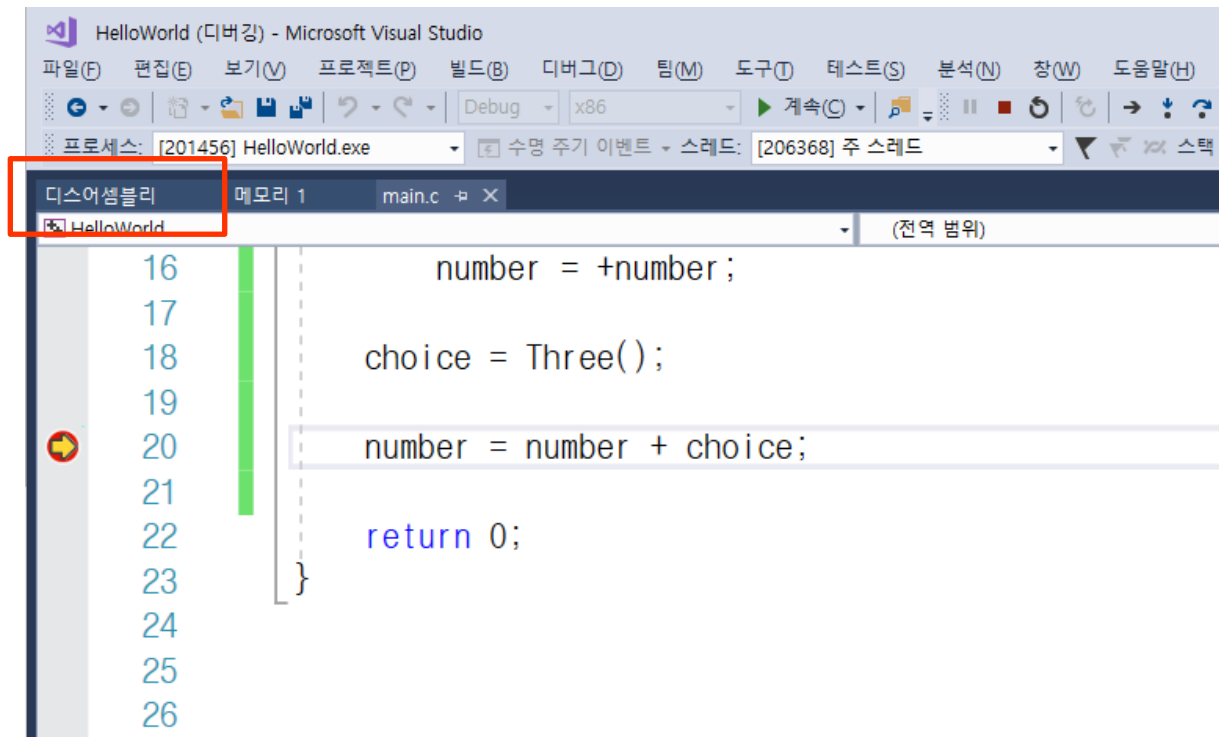


# 컴파일러가 보는 수식과 문장

- 오오... '디버깅'이 시작되더니, 노란 화살표가 빨간 점 자리에 멈춰 있습니다.
  - 이렇게 찍는 빨간 점을 breakpoint(박살점? 중단점)라 불러요
- F10이 '첫 문장부터 천천히 디버깅'이었다면 F5는 '일단 짚욱 디버깅'을 의미하고, 디버깅 중에는 노란 화살표가 빨간 점(으로 표시해 둔 Code 앞)으로 올 때마다 자동으로 일시정지를 해 줘요
  - 큰 프로그램을 만들 때는 main() 첫 줄부터 따라 들어가려면 매우 번거롭기 때문에 이런 식으로 미리 빨간 점을 적절히 찍어 둔 다음 F5를 눌러 ㄱㄱ하게 돼요
  - 일단 노란 화살표가 떠 있는 시점부터는 평소처럼 F10 눌러서 한 문장씩 ㄱㄱ 가능해요!

# 컴파일러가 보는 수식과 문장

- 이제 여기서 '디스어셈블리' 탭을 눌러 봅시다.
  - 안 보이는 친구들은 디버그(D) → 창(W) → 디스어셈블리(D)를 눌러요.



# 컴파일러가 보는 수식과 문장

- 뭔가 불길한 예감이 듭니다.
  - 이 수업 듣다 16진수 극혐 하게 될 것 같아요

```
디스어셈블리  X 메모리 1  main.c
주소(A): main(...)
보기 옵션
choice = Three( );
00F5173B E8 F6 F9 FF FF call _Three (0F51136h)
00F51740 89 45 EC mov dword ptr [choice],eax

number = number + choice;
00F51743 8B 45 F8 mov eax,dword ptr [number]
00F51746 03 45 EC add eax,dword ptr [choice]
00F51749 89 45 F8 mov dword ptr [number],eax

return 0;
00F5174C 33 C0 xor eax,eax
}
00F5174E 5F pop edi
00F5174F 5E pop esi
00F51750 5B pop ebx
00F51751 81 C4 D8 00 00 00 add esp,008h
```

# 컴파일러가 보는 수식과 문장

여기 있는 익숙한 4B짜리 친구들은 모두 위치 값이 맞아요.

```
choice = Three( );
00F5173B E8 F6 F9 FF FF      call    _Three (0F51136h)
00F51740 89 45 EC          mov     dword ptr [choice],eax

number = number + choice;
00F51743 8B 45 F8          mov     eax,dword ptr [number]
00F51746 03 45 EC          add     eax,dword ptr [choice]
00F51749 89 45 F8          mov     dword ptr [number],eax

return
00F5174C 33 00              xor     eax,eax
}
00F5174E 5F                pop     edi
00F5174F 5E                pop     esi
00F51750 5B                pop     ebx
00F51751 81                int3
```

실제로 메모리 탭 가서 검색해 보면 이 숫자들이 들어 있어요!

숫자가 안 보이는 경우 여기를 우클릭한 다음  
코드 바이트 표시(Y)를 눌러 주세요.

# 컴파일러가 보는 수식과 문장

- 뭔가 불길한 예감이 듭니다.
  - 이 수업 듣다 16진수 극혐 하게 될 것 같아요

디스어셈블리 X 메모리 1 main.c

주소(A): main(...)

보기 옵션

```
choice = Three();
00F5173B E8 F6 F9 FF FF    call    _Three (0F51136h)
00F51740 89 45 EC              mov     dword ptr [choice],eax

number = number + choice;
00F51743 8B 45 F8              mov     eax, dword ptr [number]
00F51746 03 45 EC              add     eax, dword ptr [choice]
00F51749 89 45 F8              mov     dword ptr [number],eax

return 0;
00F5174C 33 C0                xor     eax, eax
}
00F5174E 5F                  pop     edi
00F5174F 5E                  pop     esi
00F51750 5B                  pop     ebx
00F51751 81 C4 D8 00 00 00    add     esp, 000000D8h
```

숫자 옆에 있는 뭔가 암호같은 친구들이,  
'이 숫자를 사람이 좀 더 편하게 읽을 수 있는 형태로 적은 것',  
소위 말하는 '어셈블리 코드'에 해당하는 친구예요.

# 컴파일러가 보는 수식과 문장

- 잠시 시간을 내어,  
지금 main() 정의 안에 적어 둔 **문장**들의 의미를 토대로,  
여기 있는 어셈블리 코드 친구들을 잠시 구경해 봅시다.
  - 비슷한 **문장** / **수식**들은 비슷한 코드로 구성되어 있어요
  - 몇몇 고유명사급 단어들을 빼면 그럭저럭 읽을 만한 영어 단어로 구성되어 있어요
    - DWORD PTR [number]는 '**변수** number에 대한 **위치 값**'이라 생각하면 정확해요
  - 조금 있다가 강사랑 함께 천천히 리뷰해 볼게요

# 복습을 위한 슬라이드

- 우리가 적는 수식, 문장들도 결국 **runtime**에는 숫자가 되어 존재합니다.
  - CPU는 이 숫자들의 의미를 정확히 알고 있으니, 우리는 지금은 몰라도 무방해요
  - (중요)이 숫자들의 나열이 곧 **순차** 개념의 본질에 해당해요!
- = **수식**은 주로 mov 명령어로 구성됩니다.
  - mov 명령어 자체는 '수식으로써 적은 **이름**'들 하나하나에 다 붙어 있는 듯 해요
- < 연산자는 일단 cmp 명령어(뺄셈을 해요)로,  
'다음 명령어'를 지정하는 것은 jge, jmp 같은 '점프' 친구들로 구성됩니다.
  - jge: 앞 뺄셈의 결과가 0보다 크거나 같다면 '노란 화살표 값'을 변경
  - jmp: 그딴거 없이 무조건 변경

# 복습을 위한 슬라이드

- 'eax'가 꽤 자주 등장했습니다.
  - 정식 명칭은 register(레지스터)라 불러요
- CPU가 자신이 사용할 값을 담아 둘 수 있는 친구예요(CPU에 내장되어 있음)
  - 당연히겠지만 메모리 자체는 CPU '밖'에 있어요!(노트북 기준으로는 아예 다른 부품임)
- 우리가 본 Intel CPU용 명령어는, 보통은,  
한 손에는 레지스터, 다른 손에는 레지스터 / **위치 값** / **상수 값**을 줘도록 구성돼요.
  - 양 손에 **위치 값**을 주는 것은 보통은 잘 안 해요
  - 그래서 수식 `number + choice`는...
    - `eax`에 `number`에 담긴 값을 담고,  
`eax`와 `choice`에 담긴 값을 더해서 다시 `eax`에 담고(+= 연산자같은 느낌?)  
`number` 자리에 `eax` 값을 담도록 컴파일되었어요



# 복습을 위한 슬라이드

- (중요)수식 `number = 5`의 경우,  
아예 명령어 안에 `05 00 00 00`이 그대로 박혀 있었습니다.
  - 계산에 필요한 숫자를 명령어에 내장시켜 두면  
굳이 메모리를 방문하지 않아도 원하는 값 5를 얻을 수 있어요
- 그래서 이런 '명령어에 내장된 값'을 보통 `immediate value`(즉시값)라 불러요

# 복습을 위한 슬라이드

- (중요)이렇게 해도 괜찮은 이유는,  
**runtime**에 프로그램이 어떤 실행 흐름을 타는지와 전혀 무관하게  
**수식 5를 계산한 결과값은 언제나 동일할 것이기 때문이에요!**
  - 이게 바로 **변수**와 대비되는 개념인 **상수**의 의미예요
    - **변수 이름**으로 수식을 적는다면,  
그걸 **계산한 결과값은 계산** 시점에 메모리의 그 **변수** 자리에 뭐가 담겨 있냐에 따라 달라져요
    - **상수 수식**을 적는다면,  
그걸 **계산한 결과값은 runtime**의 동작과 무관하게 언제나 동일해요
      - 그래서 심지어 '실행 전', 다시 말하면 compile time에도 그 **값**을 확정할 수 있어요!

# 컴파일러가 보는 수식과 문장

- 간단한 실험을 해 봅시다.  
코드에서 `number = 5;`에 해당하는 부분을 아래와 같이 고쳐 적어 주세요:

```
//number = 5;  
number = 2 + 3;
```

# 컴파일러가 보는 수식과 문장

---

- 간단한 실험을 해 봅시다.  
코드에서 `number = 5;`에 해당하는 부분을 아래와 같이 고쳐 적어 주세요:
  - 다 고쳤다면 F10이나 F5를 누르고 디스어셈블리 창의 해당 부분 내용을 확인해 봅시다

# 컴파일러가 보는 수식과 문장

- 간단한 실험을 해 봅시다.  
코드에서 `number = 5;`에 해당하는 부분을 아래와 같이 고쳐 적어 주세요:
  - 다 고쳤다면 F10이나 F5를 누르고 디스어셈블리 창의 해당 부분 내용을 확인해 봅시다
- 세상에...  
똑똑한 컴파일러가 미리 **상수 수식**  $2 + 3$ 을 **계산**해서 **컴파일**을 해 두었습니다

# 컴파일러가 보는 수식과 문장

- 간단한 실험을 해 봅시다.  
코드에서 `number = 5;`에 해당하는 부분을 아래와 같이 고쳐 적어 주세요:
  - 다 고쳤다면 F10이나 F5를 누르고 디스어셈블리 창의 해당 부분 내용을 확인해 봅시다
- 세상에...  
똑똑한 컴파일러가 미리 **상수 수식**  $2 + 3$ 을 **계산**해서 **컴파일**을 해 두었습니다.
- 사실,  
지금 명령어에 `number`나 `choice`, `Three`라는 '**이름**(글자) 자체'는 전혀 안 들어 있지요?

# 컴파일러가 보는 수식과 문장

- 간단한 실험을 해 봅시다.  
코드에서 `number = 5;`에 해당하는 부분을 아래와 같이 고쳐 적어 주세요:
  - 다 고쳤다면 F10이나 F5를 누르고 디스어셈블리 창의 해당 부분 내용을 확인해 봅시다
- 세상에...  
똑똑한 컴파일러가 미리 **상수** 수식 `2 + 3`을 계산해서 **컴파일**을 해 두었습니다.
- 사실,  
지금 명령어에 `number`나 `choice`, `Three`라는 '이름(글자) 자체'는 전혀 안 들어 있지요?
  - 이 **이름**들 또한 컴파일 과정에서 그 **이름**에 대한 '**위치 상수 값**'으로 바뀌어 명령어에 탑재되어 있어요!
    - 진짜 **위치 값**은 길기 때문에 CPU 본인만 알 수 있을 형태로 가공해 담긴 했음 (이건 그러려니 해도 돼요)

# 컴파일러가 보는 수식과 문장

- 여기까지 나온 내용을 요약하면...
  - 컴파일러는 우리가 적은 **수식 / 문장**을 가지고 CPU가 읽을 만한 숫자를 만든다
  - 그 과정에서 **상수 수식**에 대한 **계산**을 미리 수행해 둔다
    - 수식  $2 + 3$ 은 명백한 **상수 수식**이므로 미리 **계산**함
    - 수식 `number`도 '`number`'의 **위치 값**'이 고정되어 있다 생각할 수 있으므로 해당 **값**을 미리 **계산**함
      - 이게 고정되어 있어야 '아까 담은 **값**을 나중에 찾기'가 가능해요!
  - **계산**된 일부 **상수**들은 별도의 메모리 어딘가에 담겨 있지 않고 명령어 안에 내장되어 있다
    - Immediate value(즉시값)이라 불러요



# 컴파일러가 보는 수식과 문장

---

- 음... 그런데 다시 생각해 보면,

# 컴파일러가 보는 수식과 문장

- 음... 그런데 다시 생각해 보면,  
수식  $2 + 3$ 은 미리 **계산**해 두면서,  
수식 `Three()`는 왜 미리 **계산**해 두지 않고 있을까요?
  - 어차피 무조건 3을 `return`하기 때문에, 그냥 수식 3 이라고 고쳐 적어도 될 듯 해요

# 컴파일러가 보는 수식과 문장

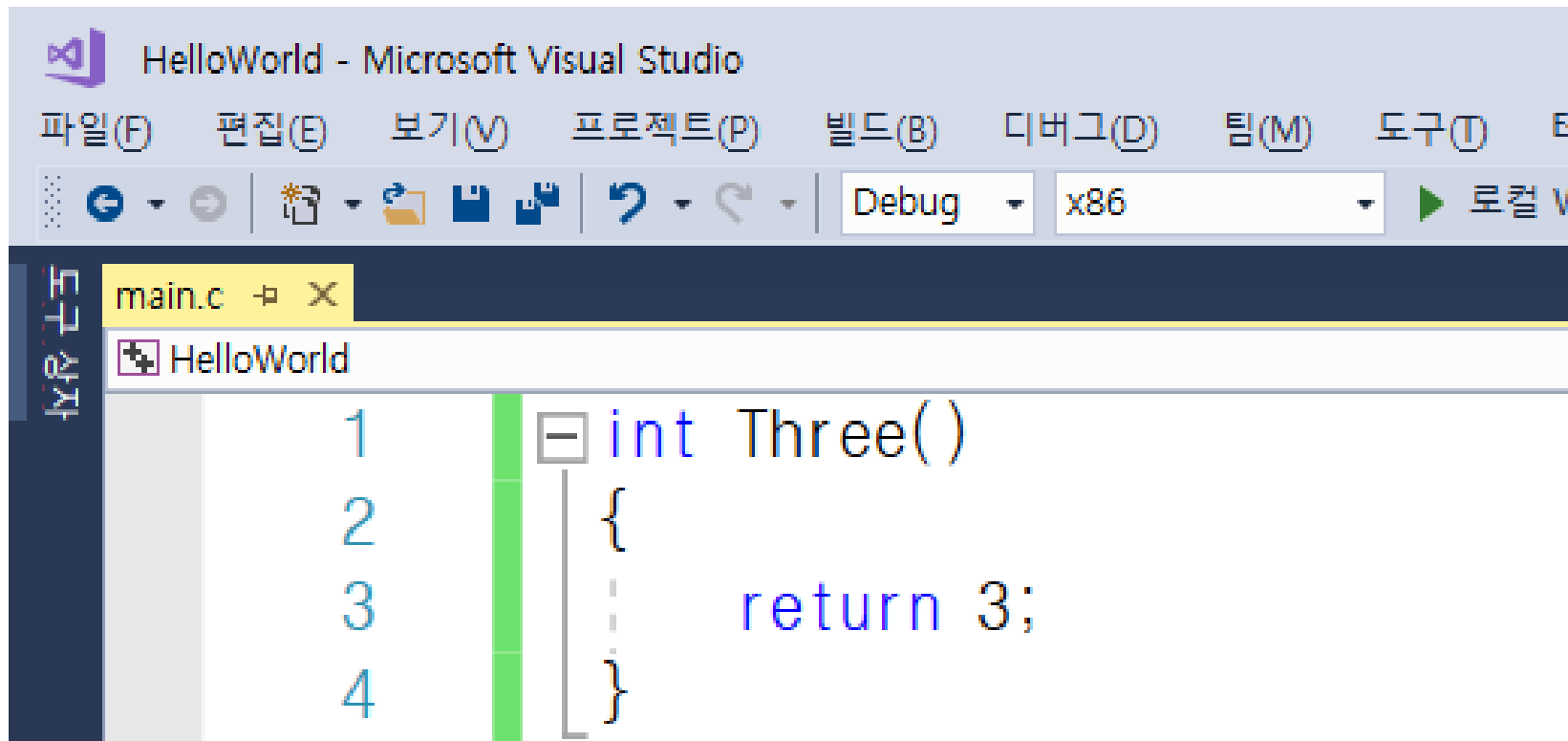
- 음... 그런데 다시 생각해 보면,  
수식  $2 + 3$ 은 미리 **계산**해 두면서,  
수식 `Three()`는 왜 미리 **계산**해 두지 않고 있을까요?
  - 어차피 무조건 3을 `return`하기 때문에, 그냥 수식 3 이라고 고쳐 적어도 될 듯 해요
- 맞아요. 여기서의 정답이 곧  
컴파일러 입장에서 **수식**과 **문장**을 다르게 보고 있음을 의미해요.
  - 프로그래머가 굳이 이렇게 `return`문 섞어서 동일한 **값**을 얻도록 구성한 것에 뭔가 다른 의도가 있을 것이라고 생각하기 때문이에요

# 컴파일러가 보는 수식과 문장

- 음... 그런데 다시 생각해 보면,  
수식  $2 + 3$ 은 미리 **계산**해 두면서,  
수식 `Three()`는 왜 미리 **계산**해 두지 않고 있을까요?
  - 어차피 무조건 3을 return하기 때문에, 그냥 수식 3 이라고 고쳐 적어도 될 듯 해요
- 맞아요. 여기서의 정답이 곧  
컴파일러 입장에서 **수식**과 **문장**을 다르게 보고 있음을 의미해요.
  - 프로그래머가 굳이 이렇게 return문 섞어서 동일한 **값**을 얻도록 구성한 것에 뭔가 다른 의도가 있을 것이라고 생각하기 때문이에요
  - 그러면 이제 컴파일러에게 '나 그런 의도 없어'라고 말하는 방법을 살펴 봅시다!

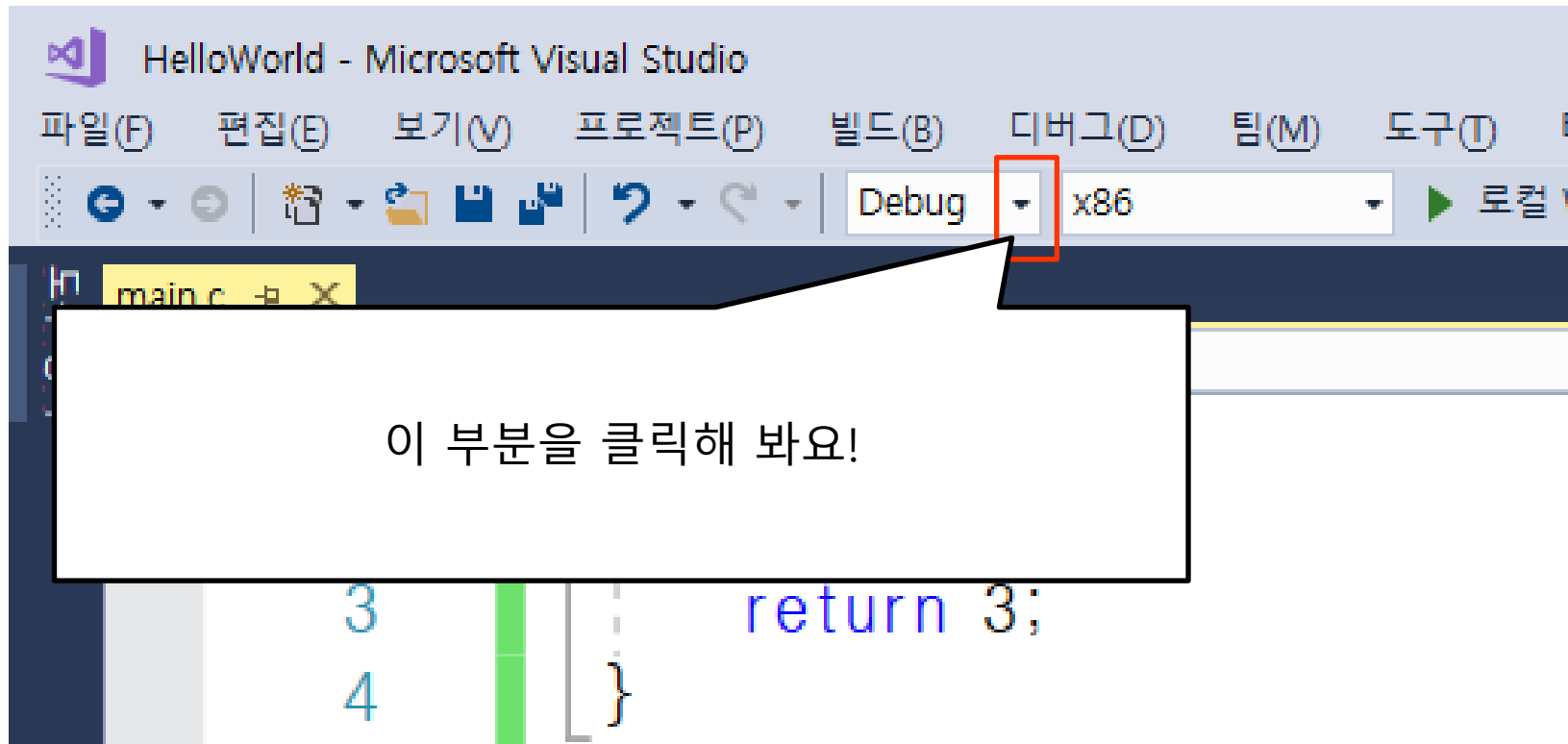
# 컴파일러가 보는 수식과 문장

- 디버깅중이라면 Shift + F5 눌러서 중단하고, 코드 창 윗 부분을 잠시 확인해 봅시다.



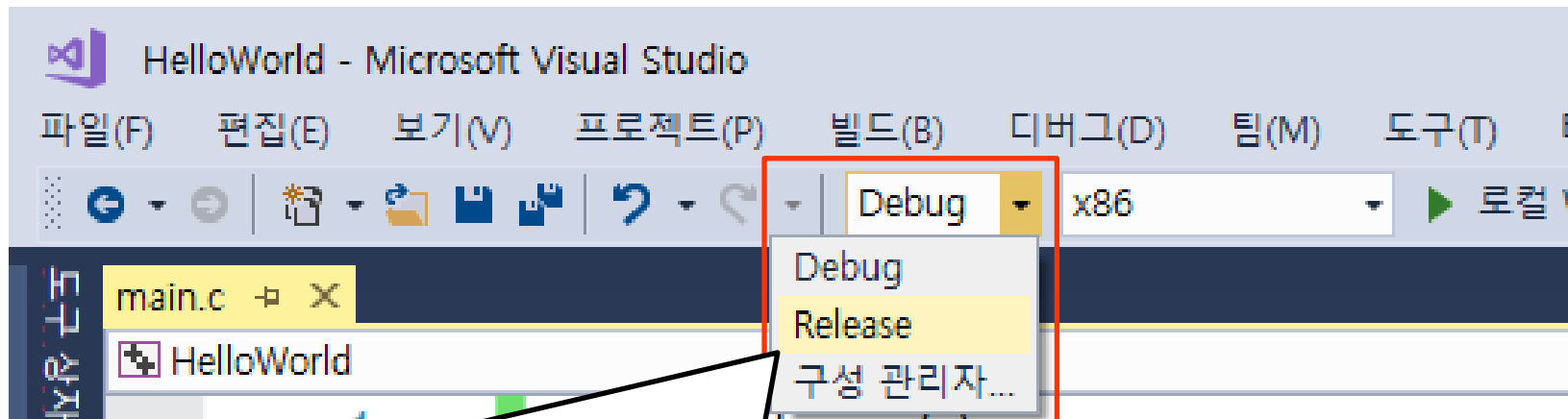
# 컴파일러가 보는 수식과 문장

- 디버깅중이라면 Shift + F5 눌러서 중단하고, 코드 창 윗 부분을 잠시 확인해 봅시다.



# 컴파일러가 보는 수식과 문장

- 디버깅중이라면 Shift + F5 눌러서 중단하고, 코드 창 윗 부분을 잠시 확인해 봅시다.



이렇게 뜨는 메뉴에서 Release를 골라 봅시다.

3;

# 컴파일러가 보는 수식과 문장

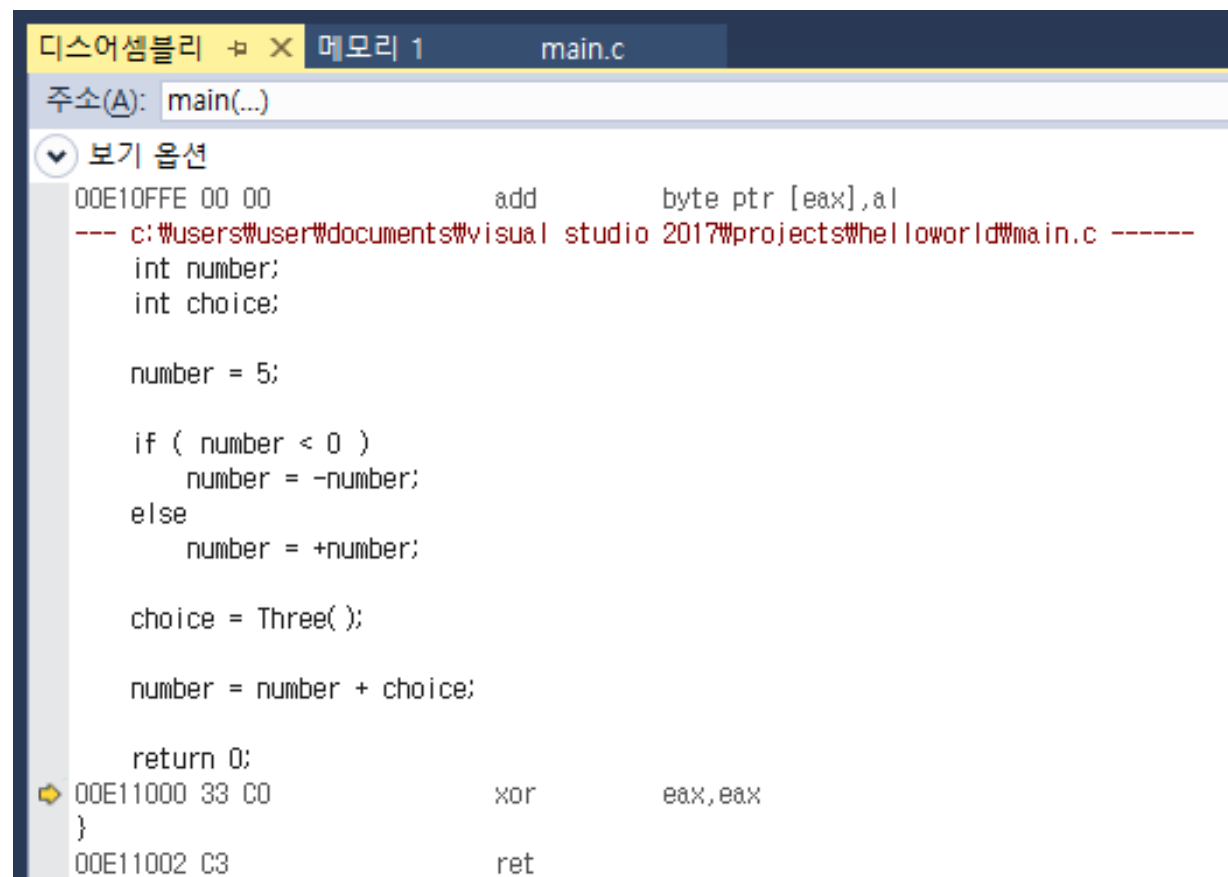
---

- 잘 골랐다면, 다시 한 번 F10을 누르고 디스어셈블리를 구경해 봅시다.
  - 이번에는 F5 말고 F10 눌러 주세요



# 컴파일러가 보는 수식과 문장

- 세상에...



```
디스어셈블리  X 메모리 1  main.c
주소(A):  main(...)
보기 옵션
00E10FFE 00 00          add     byte ptr [eax],al
---- c:\users\user\documents\visual studio 2017\projects\helloworld\main.c -----
    int number;
    int choice;

    number = 5;

    if ( number < 0 )
        number = -number;
    else
        number = +number;

    choice = Three( );

    number = number + choice;

    return 0;
00E11000 33 C0          xor     eax, eax
}
00E11002 C3          ret
```

# 컴파일러가 보는 수식과 문장

사실 애네는 main() 바깥 세상에 아무 영향도 주지 않는 친구들이었어요.  
무자비한 컴파일러의 눈에는 다 '뿔짓'으로 보이는 게 정상이었고,  
그래서 마지막 return문을 제외한 나머들이 싹 다 증발했어요.

main()을 호출하는 쪽 입장에서 본다면, 애를 호출한다 해서  
메모리 어디가 달라지거나  
프로그램 밖(검은 창, 키보드 등)과 Data 흐름을 구성하거나  
...하지 않기 때문에 뿔짓으로 보이는 게 맞고,

return문은 '나한테 보낼 숫자 값'을 특정하는 문장이니  
의미가 있는 게 맞아요.

00E10FFE 00 00

---- c:\users\user\documents\visual studio 2017\projects\helloworld\main.c -----

int number;

int choice;

add byte ptr [eax],al

xor eax, eax

}  
00E11002 C3

ret

# 컴파일러가 보는 수식과 문장

- 세상에...

디스어셈블리 X 메모리 1 main.c

주소(A): main(...)

보기 옵션

00E10FFE 00 00 add byte ptr [eax], al

elloworld\main.c -----

여기 있는 `xor eax, eax`는  
이 동네에서 'eax에 0 담기'를 의미하는 보편적 숙어예요.  
(Debug 모드일 때도 이렇게 썼고,  
얘가 `mov eax, 0`보다 명령어 길이가 더 짧아요)

Intel CPU용 컴파일러들은 보편적으로  
'int **형식** return값'을 `eax`에 담아 전달해요.

number = number

return 0;

00E11000 33 C0 `xor eax, eax`

}

00E11002 C3 ret

# 컴파일러가 보는 수식과 문장

- 방금 구경한 것은 컴파일러의 '최적화' 기능입니다.
  - 수식 단위로만 제한하지 않고,  
문장 단위, 또는 **파일** 단위까지 넘보면서 더 멋진 코드를 만들기 위해 노력해 줘요
    - 프로그램 크기를 줄이거나 실행 속도를 향상시키거나 할 수 있어요
  - 가끔 '내 의도와 다른 칼질'을 해서 속상할 가능성이 있긴 하지만,  
뭐 수식들 문장들을 건전한 마음가짐으로 적어 놓았다면 크게 걱정 안 해도 돼요
  - 최적화와 관련한 옵션들이 꽤 많지만,  
지금은 그냥 '두 가지 모드가 있다' 정도만 봐 두면 적당할 듯?
    - 일단 지금은 다시 Debug, x64 모드로 돌려놓도록 합시다

# 사잇단계 마무리

- 꽤 많은 복선들이 등장하긴 했지만 그럭저럭 납득하기 어렵지 않았을 거예요.
  - 지금은 그냥 납득만 해 두어도 충분해요. 완벽히 이해하려면 3학년쯤 되어야 함
- 수식과 문장의 의도 측면에서의 차이점,  
상수 개념(상수 값, 상수 수식),  
방금 디스어셈블리 탭을 구경해 본 기억 정도만 가져가면 돼요
- 쉬고 싶은 마음은 이해하지만... 조금만 더 진행한 다음 쉬는 시간을 가집시다

# 정의

---

- 이제 본격적으로 **정의**(definition)에 대해 소개합니다.
- 납득하기 위해 필요한 요소들은 이미 자주 경험해 보았으니 빠르게 진행할 수 있을 듯 해요.

# 정의

---

- C의 **정의**(definition)는 두 가지 의미를 가집니다:

# 정의

---

- C의 **정의**(definition)는 두 가지 의미를 가집니다:
  - 함수 정의에서처럼, 무언가의 내용물을 정하는 것



# 정의

---

- C의 **정의**(definition)는 두 가지 의미를 가집니다:
  - 함수 정의에서처럼, 무언가의 내용물을 정하는 것
  - 그리고, 무언가의 위치 '**상수**' 값을 정하는 것

# 정의

---

- C의 **정의**(definition)는 두 가지 의미를 가집니다:
  - 함수 정의에서처럼, 무언가의 내용물을 정하는 것
  - 그리고, 무언가의 위치 '**상수**' 값을 정하는 것

'누가' 정하냐고 묻는다면...

# 정의

---

- C의 **정의**(definition)는 두 가지 의미를 가집니다:
  - **함수 정의**에서처럼, 무언가의 내용물을 정하는 것
    - 프로그래머가 직접 정합니다
  - 그리고, 무언가의 **위치 '상수' 값**을 정하는 것
    - 컴파일 과정에서 컴파일러가 정해 줍니다

# 정의

- 선언되지 않은 **이름**을 적었을 때 이런 오류를 본 적이 있을 거예요:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      number = 3;
6
7      return 0;
8  }
9
```

식별자 "number"이(가) 정의되어 있지 않습니다.

# 정의

---

- **수식** `number = 3`은 'number 자리에 3 담기'를 의미하지요?
  - 디스어셈블리 창에서 `DWORD PTR [number]` 같은 느낌으로 본 것 같군요

# 정의

- **수식** `number = 3`은 'number 자리에 3 담기'를 의미하지요?
  - 디스어셈블리 창에서 `DWORD PTR [number]` 같은 느낌으로 본 것 같군요
- Python에서는, '**이름에 값을 담는 것**'이 일단 그 **값**을 메모리 어딘가에 담아 두거나 이미 그 **값**이 담긴 곳을 찾는 다음 이름 사전의 해당 **이름** 자리를 변경하는 것을 의미했어요.

# 정의

- **수식** `number = 3`은 'number 자리에 3 담기'를 의미하나요?
  - 디스어셈블리 창에서 `DWORD PTR [number]` 같은 느낌으로 본 것 같군요
- Python에서는, '**이름에 값을 담는 것**'이 일단 그 **값**을 메모리 어딘가에 담아 두거나 이미 그 **값**이 담긴 곳을 찾는 다음 이름 사전의 해당 **이름** 자리를 변경하는 것을 의미했어요.
- C의 경우, **변수에 값을 담는 것**은 컴파일러가 미리 정해 둔 자리에 **값**을 담는 것을 의미해요.
  - 디스어셈블리 창에서 본 내용들은 전부 **상수**라 말할 수 있어요
  - **변수의 위치**가 고정되어 있고 거기에 그 **값**을 반드시 담으므로 **runtime**에 신경쓸 게 줄어들게 돼요 → 그래서 더 빠름!

# 정의

---

- 다시 말하면, **이름** number에 대한 위치 '상수' 값은,  
우리가 프로그램을 실행하기 전에 컴파일을 통해 미리 정의됩니다.



# 정의

---

- 다시 말하면, **이름** number에 대한 **위치 '상수' 값**은,  
우리가 프로그램을 실행하기 전에 컴파일을 통해 미리 정의됩니다.
- '...하기 전에 먼저'라는 말을 어딘가에서 들어 본 적 있지 않나요?

# 정의

---

- 다시 말하면, **이름** number에 대한 **위치 '상수' 값**은,  
우리가 프로그램을 실행하기 전에 컴파일을 통해 미리 정의됩니다.
- '...하기 전에 먼저'라는 말을 어딘가에서 들어 본 적 있지 않나요?
- 네, C에서 **정의**는 항상 **선언**을 통해 이루어집니다.
  - 컴파일러는 C 선언을 읽을 때  
그 **이름**을 사용하기 위한 **위치 '상수' 값**을 알아서 정해 줍니다.

# 정의

- 다시 말하면, **이름** number에 대한 **위치 '상수' 값**은,  
우리가 프로그램을 실행하기 전에 컴파일을 통해 미리 정의됩니다.
- '...하기 전에 먼저'라는 말을 어딘가에서 들어 본 적 있지 않나요?
- 네, C에서 **정의**는 항상 **선언**을 통해 이루어집니다.
  - 컴파일러는 C 선언을 읽을 때  
그 **이름**을 사용하기 위한 **위치 '상수' 값**을 알아서 정해 줍니다.
- 뭐 여기엔 상당히 복잡한 규칙들이 관여하긴 하지만,  
일단 이 부분만 납득해 두면 꽤 많은 영역을 명확히 관찰할 수 있게 될 거예요

# 선언과 정의

---

- 아직은 **정의**에 대해 감이 잘 안 오는 게 정상이에요.
  - 한 학기 동안 꾸준히 나올 예정이니 너무 걱정 말아요
- 일단 지금은 '**이름을 선언, 내용과 위치를 정의**' 정도로만 외워 두고 다음 주제로 넘어가 보도록 합시다.

# 어떤 이름에 대해 얻을 수 있는 값들

---

- 이번에는 조금 관점을 틀어서,  
어떤 **이름**이 있을 때, 이 **이름**이랑 연관되는 **값**들이 뭐가 있을지  
잠시 상상해 보는 시간을 가져 봅시다.
- 지금은 쉬운거로 가 볼게요.

# 어떤 이름에 대해 얻을 수 있는 값들

- 아래 선언을 봅시다:

```
int number = 3;
```

# 어떤 이름에 대해 얻을 수 있는 값들

- 아래 선언을 봅시다:

```
int number = 3;
```

이 선언이 위에 적혀 있을 때,  
우리는 어떤 '종류'의 값들을 **runtime**에 활용할 수 있을까요?

# 어떤 이름에 대해 얻을 수 있는 값들

- 아래 선언을 봅시다:

```
int number = 3;
```

뭐 다른 = 수식을 계산하거나 했다면 바뀔 수 있겠지만,  
아무튼 '변수 number에 담긴 값'을 먼저 곱을 수 있습니다.



# 어떤 이름에 대해 얻을 수 있는 값들

- 아래 선언을 봅시다:

```
int number = 3;
```

뭐 다른 = 수식을 계산하거나 했다면 바뀔 수 있겠지만,  
아무튼 '변수 number에 담긴 값'을 먼저 쫓을 수 있습니다.

근데 이 3은 '어디'에 담겨 있지요?

# 어떤 이름에 대해 얻을 수 있는 값들

- 아래 선언을 봅시다:

```
int number = 3;
```

맞아요. 이 선언을 읽고 컴파일러가 정의해 둔,  
int 값 하나 담을 자리에 대한 위치 값도 얻을 수 있어요.

# 어떤 이름에 대해 얻을 수 있는 값들

- 아래 선언을 봅시다:

```
int number = 3;
```

맞아요. 이 선언을 읽고 컴파일러가 정의해 둔,  
int 값 하나 값을 자리에 대한 위치 값도 얻을 수 있어요.

그리고, 마지막으로...

# 어떤 이름에 대해 얻을 수 있는 값들

- 아래 선언을 봅시다:

```
int number = 3;
```

사실 Python에도 있었던, 'int 값 하나의 크기' 또한  
이름 number와 연계되는 값이라 할 수 있습니다.

이렇게 총 세 가지를 들 수 있어요!

# 어떤 이름에 대해 얻을 수 있는 값들

---

- int 변수 이름 number로 얻을 수 있는 값들을 나열하면...

# 어떤 이름에 대해 얻을 수 있는 값들

---

- int **변수 이름** number로 얻을 수 있는 **값들**을 나열하면...
  - 변수에 든 값
  - 정의된 위치 값
  - 크기 값

# 어떤 이름에 대해 얻을 수 있는 값들

---

- int **변수 이름** number로 얻을 수 있는 **값들**을 나열하면...
  - 변수에 든 값 → value of number
  - 정의된 위치 값 → address of number
  - 크기 값 → size of number

# 어떤 이름에 대해 얻을 수 있는 값들

- int **변수 이름** number로 얻을 수 있는 **값들**을 나열하면...
  - 변수에 든 값 → value of number
  - 정의된 위치 값 → address of number
  - 크기 값 → size of number
- 이 세 가지 값들의 'C 수식에서의 사용 빈도(직접 적는 빈도)'를 감안한다면, 지금 나열된 순서대로 사용된다고 볼 수 있을 거예요.
  - 새 **변수 이름**을 도입하는 이유 자체가 **값** 하나 담기 위해서일 테니 value가 1등
  - scanf() 등을 종종 쓰게 될 테니 address가 2등
  - 사실 우리 수업 환경에서 int 크기가 4B라는 건 이미 다 알고 있으니 size가 3등



# 어떤 이름에 대해 얻을 수 있는 값들

---

- 따라서 C에서는 이들이 **계산** 결과로 나올 C 수식을...
  - 변수에 든 값 → value of number
  - 정의된 위치 값 → address of number
  - 크기 값 → size of number

# 어떤 이름에 대해 얻을 수 있는 값들

- 따라서 C에서는 이들이 **계산** 결과로 나올 C 수식을...
  - 변수에 든 값 → value of number → number (그냥 **변수 이름** 적으면 됨)
  - 정의된 위치 값 → address of number → &number (남는 기호가 별로 없었음)
  - 크기 값 → size of number → sizeof number (기호 매진이라 키워드화함)

...와 같이 & 연산자와 sizeof 연산자를 써서 구분해 적도록 구성해 두었어요!

# 어떤 이름에 대해 얻을 수 있는 값들

- 따라서 C에서는 이들이 **계산** 결과로 나올 C 수식을...
  - 변수에 든 값 → value of number → number (그냥 **변수 이름** 적으면 됨)
  - 정의된 위치 값 → address of number → &number (남는 기호가 별로 없었음)
  - 크기 값 → size of number → sizeof number (기호 매진이라 키워드화함)

...와 같이 & **연산자**와 sizeof **연산자**를 써서 구분해 적도록 구성해 두었어요!

- scanf()는 '내가 지정한 **위치**'에 **값**을 담아 주는 친구기 때문에,  
그 친구를 호출할 때는 '여기에 담아주세요' 하면서 **위치 값**을 담아 보내야 해요
  - 그래서 &를 붙여왔어요!
  - scanf()한테는 '내 number에 지금 3 있다?' 같은 정보는 전혀 쓸 데 없어요!  
그러니 그냥 **위치 값**만 보내면 되고, 아무튼 그래서 &를 붙여왔어요!

# 어떤 이름에 대해 얻을 수 있는 값들

- 오... 마지막으로 아래 **문장**을 구경해 봅시다:

```
number = 5;
```

# 어떤 이름에 대해 얻을 수 있는 값들

- 오... 마지막으로 아래 **문장**을 구경해 봅시다:

**number** = 5;

이 수식을 계산하면 뭐가 나올까요?  
= 수식임을 감안한다면, 뭐가 나와야 할까요?

# 어떤 이름에 대해 얻을 수 있는 값들

- 오... 마지막으로 아래 **문장**을 구경해 봅시다:

**number** = 5;

이 수식을 계산하면 뭐가 나올까요?  
= 수식임을 감안한다면, 뭐가 나와야 할까요?

맞아요. 위치 값이 나와야 해요(원래 있던 3?은 노쓸모)!

# 어떤 이름에 대해 얻을 수 있는 값들

- 오... 마지막으로 아래 **문장**을 구경해 봅시다:

**number** = 5;

그래서 C에서는, = 수식의 좌항에 해당하는 수식을 계산할 때는  
마지막에 & 연산자를 붙인 것처럼 계산하도록 컴파일돼요.

사용 빈도 1등은 여기서 고려 대상이 아니니 2등을 골라준다 보면 됨!

# 어떤 이름에 대해 얻을 수 있는 값들

---

- 정리하면...
  - C에서 메모리를, **정의**를, **위치 값**을 다루기 시작하는 시점부터  
여러분의 = **수식**은,  
여러분이 만드는 프로그램의 Data 흐름은 이전보다 훨씬 더 많은 자유도를 갖게 돼요



# 어떤 이름에 대해 얻을 수 있는 값들

- 정리하면...
  - C에서 메모리를, **정의**를, **위치 값**을 다루기 시작하는 시점부터  
여러분의 = **수식**은,  
여러분이 만드는 프로그램의 Data 흐름은 이전보다 훨씬 더 많은 자유도를 갖게 돼요
  - 하지만 뭐 사용 빈도를 고려해 보면  
'**변수**에 뭐 담겨 있는지'가 '그에 대한 **위치**가 어디로 **정의**되어 있는지'보다 자주 쓰여요
    - 그래서 기본적으로는 그냥 **변수 이름** 적으면 빈도 수 1등인 '안에 든 **값**'이 나와요
      - 그게 싫다, 2등 좋다 하면 & **연산자**를 붙여서 적으면 돼요
    - 예외가 좀 있는데, 매우 자주 적게 될 = **수식**의 좌항 자리에 **변수 이름**을 적을 때는  
'그 **변수** 자리에 원래 무슨 **값**이 있었는지'는 전혀 고려 대상이 아니기 때문에  
그냥 **변수 이름** 적어도 자동으로 & **연산자**를 붙인 것처럼 컴파일해 줘요
      - 오히려 여러분이 붙이면 오류남!

# 마무리

---

- 좋아요. 여기까지 살펴 보면  
이제 C 선언 적는 방법을 본격적으로 확장해 볼 준비가 된 셈이에요.
  - 3-3 할 시간이 남아 있을 지 조금 걱정되기는 하지만  
잠깐 쉬었다가 슬쩍 구경해 봅시다