5-3. Object

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

이번 시간에는

- 본격적으로 C++ 동네로 넘어가기 전에 칸에 대한 관념을 살짝 다져 봅니다.
  - 칸 잡는(allocate하는) 방법
    - 칸을 자동으로 잡도록 코드를 구성하는 방법을 복습해 봅니다
    - 칸을 수동으로 잡아 가며 코드를 구성하는 방법을 사용해 봅니다
    - 칸을 '반자동'으로 잡을 수 있게 만들어 봅니다
  - '칸권'
    - '칸의 일생'을 둘러보고, 반자동 방식을 쓰는 C에서의 칸권에 대해 논의해 봅니다
    - 이러한 칸권 위기를 타개하는 방법을 살펴 보고, C++에서의 칸 개념에 대해 구경해 봅니다

이번 시간에는

- 바로 시작해 봅시다.

- 일단 갑자기 자주 등장한 '칸'과 '잡기'에 대해 살짝 짚어 볼께요.

칸

- 칸(**object**)
  - 이제까지 우리 수업에서는 단어 **object**를 '칸'이라 표현해 왔어요

칸

- 칸(**object**)
  - 이제까지 우리 수업에서는 단어 **object**를 '칸'이라 표현해 왔어요

  - Python에서의 칸은 우리 예상보다 좀 더 많이 복잡해요
    - '`int` 칸'에는
      숫자 값 하나 뿐만 아니라 '그 칸이 `int` **형식** 칸임'이라는 정보 또한 동봉되어 있어요
      - 그래서 어떤 칸에 대한 **위치 값**만 들고 있으면
        메모리 위의 그 자리에 있는 숫자 덩어리가 어떤 **형식 값**인지를 알 수 있어요

칸

- 칸(**object**)
  - 이제까지 우리 수업에서는 단어 **object**를 '칸'이라 표현해 왔어요

  - C에서의 칸은 그렇지 않아요
    - 그냥 정직하게 숫자만 담아요

    - 그래서 내 손에 **위치 값**이 하나 있다 하더라도,
      그것만 가지고는 '저기 있는 03 00 00 00이 int **값** 3을 의미한다'라 확신할 수 없어요
      - 어쩌면 03 만 바라보고 char **값** 3이라고 보는 것이 더 정당할 수 있어요

칸

- 칸(**object**)
  - 이제까지 우리 수업에서는 단어 **object**를 '칸'이라 표현해 왔어요

  - C에서의 칸은 그렇지 않아요
    - 그냥 정직하게 숫자만 담아요

  - 그럼에도 불구하고 우리가 이제까지 만든 프로그램들은
    설사 `int` **형식**과 `double` **형식**을 혼용해서 다루었다 하더라도
    중간에 Data가 요상하게 변질된다던가 하는 상황은 전혀 발생하지 않았어요
    - **수식** `number = rate + 1.5` 같은 표현이 얼마든지 가능했고, **계산** 양상 또한 합리적이었어요

칸

- 칸(**object**)
  - 이제까지 우리 수업에서는 단어 **object**를 '칸'이라 표현해 왔어요

  - 답은 여러 번 등장한 적 있지요?
    Python과 달리 C 동네에서는 '**수식**의 **형식**'을 따져요
    - **이름**을 'value of' 용도로 사용할 때, 그 **이름**에 대한 **선언**을 보고
      각각 서로 다른 `mov` 명령어를 사용하도록 컴파일해요

    - 덧셈식이 `int` 덧셈식인지 `double` 덧셈식인지를 구분해서
      각각 서로 다른 `add` 명령어를 사용하도록 컴파일해요

칸

- 칸(**object**)
  - 이제까지 우리 수업에서는 단어 **object**를 '칸'이라 표현해 왔어요

  - 그래서 요약하면, 칸은...
    - **이름** 및 **수식**으로 특정되는 메모리 위의 어떤 영역을 말해요
      - 칸 자체는 '딱 그 **값**(숫자들) 담기 적당한 크기'고,
        어떤 칸에 write할 때는 '딱 그 **값**(숫자들)'만 담아요

      - 그렇긴 하지만 모든 **이름**, **수식**은 그 **형식**이 고정되어 있으므로
        이름을 **선언**해 둔 대로 적법하게 사용하는 한 여간해서는 문제가 되지 않아요

    - **값**을 담거나 가져오는 것을 전제하므로,
      칸 개념은 기본적으로 Data 관점에서 유효해요
      - C에서 **함수 정의**에 대한 컴파일 결과물(명령어 덩어리)은 칸으로 간주하지 않아요

칸

- 칸을 잡다(allocate하다)
  - 칸은 메모리 영역이라고 했으니, 칸을 잡는다는 것은
    쉽게 말하면 그 영역에 대한 (배타적) 점유를 한다는 말이 돼요
    - 메모리 위의 어떤 곳을 static **변수** number용으로 잡아 놨다면,
      다른 static / automatic **변수**에 대한 칸은 절대 그 동일한 곳으로 잡히지 않아요
      - 우리 프로그램의 Code 흐름이 어떻게 흘러가든 절대 이런 일이 발생하지 않도록
        컴파일러가 알아서 적당히 정의를 해 두었어요

자동으로 칸 잡기

- 칸을 잡다(allocate하다)
  - 칸은 메모리 영역이라고 했으니, 칸을 잡는다는 것은
    쉽게 말하면 그 영역에 대한 (배타적) 점유를 한다는 말이 돼요
    - 메모리 위의 어떤 곳을 static **변수** number용으로 잡아 놨다면,
      다른 static / automatic **변수**에 대한 칸은 절대 그 동일한 곳으로 잡히지 않아요
      - 우리 프로그램의 Code 흐름이 어떻게 흘러가든 절대 이런 일이 발생하지 않도록
        컴파일러가 알아서 적당히 **정의**를 해 두었어요
  - 달리 말하면, 우리가 Data **이름**을 잘 **선언**해 두면,
    그 Data **이름**에 대한 칸은 프로그램 실행 도중 자동으로 잡혀요

자동으로 칸 잡기

- 달리 말하면,
  우리는 이제까지 '직접 칸을 잡는' 프로그래밍을 해 본 적은 없어요.
  - 컴파일러 도움만 받아 왔어요

수동으로 칸 잡기

- 달리 말하면,
  우리는 이제까지 '직접 칸을 잡는' 프로그래밍을 해 본 적은 없어요.
  - 컴파일러 도움만 받아 왔어요

- 물론 우리가 원한다면,
  컴파일러의 도움 없이 직접 수동으로 칸을 잡아 사용할 수 있어요.
  - 컴파일러가 잡아 주는 칸들과 겹치지 않도록 미리 적당한 영역을 확보해 두고,
    그 중 일부를 특정하며 잘라 활용할 수 있어요

수동으로 칸 잡기

- 간단한 실습을 해 봅니다.
  - 프로그램 내에서 사용할 칸들을 직접 계획하여 다루어 봐요
    → 의외로 그리 어렵지는 않을 거예요

  - 몇 번 진행해 보며 느낀 점을 이야기해 보고,
    이러한 수동 작업을 라이브러리 함수의 형태로 만들어 둠으로써 '반자동'화해 봐요

  - 일단 CSP_5_3_yeshi_1.c를 VS에 탑재해 봅시다

목표 1: 두 수 입력받아 합 구해 출력하기

- 예시 파일을 열어보면,
  이미 **배열 선언** 하나가 들어 있는 것을 볼 수 있어요.
  - 첫 목표인 만큼 칸 사용 계획도 미리 적어 두었어요

```
// 사용할 Data의 크기에 따라 여기를 살짝 고쳐 주세요.
#define SIZE_DATA 4096

// 나중에 칸을 마련하기 위해 큰 공간을 미리 정의해 두기 위한 선언이에요.
int data[SIZE_DATA];

int main()
{
    /*
        칸 사용 계획
            data[0]: 입력받은 첫 번째 수
            data[1]: 입력받은 두 번째 수
            data[2]: 두 수의 합
    */
```

목표 1: 두 수 입력받아 합 구해 출력하기

- 예시 파일을 열어보면,
  이미 **배열 선언** 하나가 들어 있는 것을 볼 수 있어요.
  - 첫 목표인 만큼 칸 사용 계획도 미리 적어 두었어요

  - 이제 `main()` **정의** 안에 적어 둔 목표에 따라 적절한 **문장**을 채워 보고
    직접 한 번 **실행**해 보세요.
    - 주의: Data **선언**을 더 적으면 안 돼요!
      - 그랬다가는 컴파일러가 자동으로 칸을 더 잡아줄 거예요

목표 1: 두 수 입력받아 합 구해 출력하기

- 그럭저럭 해 볼 만한가요?

목표 1: 두 수 입력받아 합 구해 출력하기

- 그럭저럭 해 볼 만한가요?
  - 사용 가능한 이름이 data 하나밖에 없기 때문에
    여러 칸들 중 하나를 선택하기 위해 [ ] 연산자를 더덕더덕 붙여 썼을 거예요

  - 뭐 그렇긴 하지만 번거로움을 제외하면 다른 부분은 그냥 평소처럼 ㄱㄱ하면 됐어요
    - 그리 어렵지 않은 듯

목표 2: 0을 입력받을 때까지의 합을 출력하기

- 이번에는 지금 적어 둔 코드를 약간 고쳐서
  위의 새 목표를 달성해 보도록 합시다.
  - 칸 사용 계획을 수정하고,
    수정해 둔 주석 내용에 맞게 차근차근 코드를 적어 주면 돼요
  - 다만, 이번에는 숫자가 몇 개 들어오는지 '코드 적는 시점'에는 알 수 없으므로
    필연적으로 **반복** 구조를 도입해 써야 할 거예요
  - 약간의 시간을 들여 직접 ㄱㄱ해 보도록 합시다

목표 2: 0을 입력받을 때까지의 합을 출력하기

- 이번에도 역시나, 큰 어려움 없이 목표를 달성할 수 있었을 거예요.
  - 숫자가 몇 개 들어올 것인지는 모르지만,
    '합 구하기'가 목표인 이상 그리 중요하지는 않았어요

목표 2: 0을 입력받을 때까지의 합을 출력하기

- 이번에도 역시나, 큰 어려움 없이 목표를 달성할 수 있었을 거예요.
  - 숫자가 몇 개 들어올 것인지는 모르지만,
    '합 구하기'가 목표인 이상 그리 중요하지는 않았어요

  - 음... 만약에 '중간값 구하기' 같은 게 목표였다면 어땠을까요?
    - 필연적으로 모든 수를 다 담아 두어야만 하는 경우가 있다면?

목표 2: 0을 입력받을 때까지의 합을 출력하기

- 이번에도 역시나, 큰 어려움 없이 목표를 달성할 수 있었을 거예요.
  - 숫자가 몇 개 들어올 것인지는 모르지만,
    '합 구하기'가 목표인 이상 그리 중요하지는 않았어요

  - 음... 만약에 '중간값 구하기' 같은 게 목표였다면 어땠을까요?
    - 필연적으로 모든 수를 다 담아 두어야만 하는 경우가 있다면?

    - 맞아요. 그냥 하면 돼요
      - 실질적으로 **runtime**에 몇 칸을 필요로 하는지 몰라도,
        '코드 적는 시점'에 적당히 큰 배열을 잡아 두기만 하면 어려움 없이 진행할 수 있을 거예요

      - 지금 우리가 진행하고 있는 '수동 칸 잡기' 또한 그거랑 비슷한 뉘앙스라고 느낄 수 있을 거예요

목표 3: Fib()을 재차 호출하여 피보나치 수 구하기

- 이번에는 약간 어려운 목표를 준비해 봤어요.

- 새 함수 void Fib(void)를 **정의**해 둔 다음,
  이를 재차 호출하여 사용자가 입력한 수에 대한 피보나치 수를 구해 봅시다.
  - 주의: int Fib(int)가 아니에요! 이번에도 무조건 Data **이름**은 data만 사용해야 해요

  - 이건 좀 어려울 수 있어요. 한 번 도전해 봅시다

목표 3: Fib()을 재차 호출하여 피보나치 수 구하기

- 강사와 함께 답안들 중 하나를 살짝 구경해 봅시다.

목표 3: Fib()을 재차 호출하여 피보나치 수 구하기

- 강사와 함께 답안들 중 하나를 살짝 구경해 봅시다.
  - 오... '기준 **위치**'에 해당하는 Data를 담을 칸을 하나 마련해 두고
    매 호출 전/후에 해당 **값**을 적절히 증감시키면서
    '이번 호출에서 쓸 칸' 개념을 도입했어요
    - C 컴파일러가 automatic **변수**에 대해 자동으로 칸이 잡히도록 만드는 방법과 유사해요

목표 3: Fib()을 재차 호출하여 피보나치 수 구하기

- 강사와 함께 답안들 중 하나를 살짝 구경해 봅시다.
  - 오... '기준 **위치**'에 해당하는 Data를 담을 칸을 하나 마련해 두고
    매 호출 전/후에 해당 **값**을 적절히 증감시키면서
    '이번 호출에서 쓸 칸' 개념을 도입했어요
    - C 컴파일러가 automatic **변수**에 대해 자동으로 칸이 잡히도록 만드는 방법과 유사해요
  - 아무튼, data의 크기를 벗어나지 않는 이상 이 코드는 잘 동작할 것 같아요.
    - 사실 이것 또한 C 컴파일러가 automatic **변수**를 취급할 때의 단점과 동일해요!

중간 정리

- '칸을 잡음(allocate)'의 본질은,
  '미리 잡아 둔 큰 영역의 일부를 배타적으로 점유'함이에요.
  - C 컴파일러는 이 분야의 달인이라 말할 수 있어요

  - 우리도 스스로 계획해 둔 대로 data의 일부만을 따로 불러 가며 사용할 수 있어요
    - 전체 측면에서는 **배열**의 일부지만, 적어도 우리 스스로에게는 '작은 칸'으로 쓰여요

  - **함수** 호출 과정에도 이러한 뉘앙스가 잘 반영되어 있어요
    - 이전에 살짝 봤던 EBP 레지스터 등이 생각나면 좋을 듯

관점 확장

- 음… 지금까지는 모두 int 형식 Data만 사용해 왔는데,
  만약 여러 **형식** 칸을 수동으로 잡아 사용하려면 어떻게 하면 될까요?

관점 확장

- 음... 지금까지는 모두 int 형식 Data만 사용해 왔는데,
  만약 여러 **형식** 칸을 수동으로 잡아 사용하려면 어떻게 하면 될까요?
  - double **형식** 큰 배열을 선언해 두고 쓰기?
  - data에서의 두 칸을 double 한 칸으로 간주해 쓰기?

관점 확장

- 음... 지금까지는 모두 int 형식 Data만 사용해 왔는데,
  만약 여러 **형식** 칸을 수동으로 잡아 사용하려면 어떻게 하면 될까요?
  - double **형식** 큰 배열을 선언해 두고 쓰기?
  - data에서의 두 칸을 double 한 칸으로 간주해 쓰기?
    - 오늘은 이쪽 관점을 한 번 직접 접근해 보도록 할께요

다양한 형식 칸 수동으로 잡기

- 원래 작성하던 파일은 잠시 빼 두고,
  CSP_5_3_yeshi_2.c를 탑재해 봅시다.
  - 익숙한 **구조체 정의** 하나가 들어 있을 거예요

다양한 형식 칸 수동으로 잡기

- 이전 버전과의 차이점으로,
  여기 있는 data는 int **배열**이 아닌 char **배열**이에요.
  - double 한 칸은 char 여덟 칸과 크기가 동일해요

다양한 형식 칸 수동으로 잡기

- 이전 버전과의 차이점으로,
  여기 있는 data는 int **배열**이 아닌 char **배열**이에요.
  - double 한 칸은 char 여덟 칸과 크기가 동일해요

- 이러한 점에 착안하여,
  main() **정의** 안에 주석으로 적어 둔 목표에 맞는 **문장**들을
  직접 한 번 적어 봅시다.
  - 주의: p_rate와 p_stat에 잡히는 각 칸은 서로 겹치지 않아야 해요!
  - 잘 적었다면 **실행**했을 때 납득할 만한 출력 결과가 나올 거예요

다양한 형식 칸 수동으로 잡기

- 나쁘지 않지요?
  - 예쁘게 칸을 잡으려면 처음에는 [0], 그 다음에는 [8](double 크기만큼)을 쓰면 될 듯!
    - **구조체 멤버 선언**에 대해 컴파일러가 **정의**할 때도 비슷한 느낌을 썼던 것 같아요

다양한 형식 칸 수동으로 잡기

- 나쁘지 않지요?
  - 예쁘게 칸을 잡으려면 처음에는 [0], 그 다음에는 [8](double 크기만큼)을 쓰면 될 듯!
    - **구조체 멤버 선언**에 대해 컴파일러가 **정의**할 때도 비슷한 느낌을 썼던 것 같아요

  - (중요)수동으로 칸을 잡을 때는
    '잘라 쓸 목적으로 원래 **선언**해 둔 **배열**의 형식'은 그리 중요하지 않아요
    - 어차피 그 중 일부를 잘라 '여기까지만 double 한 칸임' 하고 사용할 것이기 때문

반자동

- 나쁘지 않지요?
  - 예쁘게 칸을 잡으려면 처음에는 [0], 그 다음에는 [8](double 크기만큼)을 쓰면 될 듯!
    - **구조체 멤버 선언**에 대해 컴파일러가 **정의**할 때도 비슷한 느낌을 썼던 것 같아요
  - (중요)수동으로 칸을 잡을 때는
    '잘라 쓸 목적으로 원래 **선언**해 둔 **배열**의 형식'은 그리 중요하지 않아요
    - 어차피 그 중 일부를 잘라 '여기까지만 double 한 칸임' 하고 사용할 것이기 때문
  - 그렇게 놓고 보면,
    사실 방금 적었던 두 **문장**이 생김새가 매우 동일했다는 점에 착안해서
    '원하는 크기만큼 칸을 잡아 주는 **함수**'를 **정의**해 두고
    필요할 때마다 그 **함수**를 호출해 칸을 잡아다 쓰는 것을 상상해 볼 수 있을 거예요
    - main() **정의** 적는 프로그래머의 입장에서는 칸을 '반자동'으로 잡아서 쓰는 셈임!

반자동

- 이번에는 CSP_5_3_yeshi_3.c를 탑재해 열어 봅시다.
  - 방금 생각해 본 것을 `void *Alloc(int)`과 같이 적어 두었어요

반자동

- 이번에는 CSP_5_3_yeshi_3.c를 탑재해 열어 봅시다.
  - 방금 생각해 본 것을 `void *Alloc(int)`과 같이 적어 두었어요
  - 추가로, '더 이상 안 쓸 칸을 반납'하기 위한 `void Free(void *)`도 같이 적어 두었어요.

반자동

- 오늘의 마지막 실습은 이 두 함수를 직접 만들어 보는 것이에요.
  - (뒤에 이론 내용이 더 있긴 해요)
  - 적어 둔 주석 설명에 맞게 적절한 **반복** 구조를 채워 주면 될 듯 해요
  - 성공적으로 만들었다면, 실행하고 매 번 엔터 키를 칠 때마다
    '동일한 **위치**'에 각 칸을 잡아 사용한 다음 반납하는 것을 반복하게 될 거예요
  - 차근차근 한 번 시도해 봅시다
    - 어려운 목표긴 해요. **반복**을 어떻게 진행할 것인지 잘 계획해 보아야 할 듯!

반자동

- 답안들 중 하나를 한 번 구경해 봅시다.

반자동

- 답안들 중 하나를 한 번 구경해 봅시다.
  - 지금 시점에는 **반복** 구조 자체에는 집중하지 않아도 돼요.
    '이게 되네'가 중요함!

반자동

- 답안들 중 하나를 한 번 구경해 봅시다.
  - 지금 시점에는 **반복** 구조 자체에는 집중하지 않아도 돼요.
    '이게 되네'가 중요함!

  - (중요)이 방법을 썼을 때의 '칸의 일생'은 **선언**에 의해 결정되는 것이 아닌,
    순수하게 `Alloc()` / `Free()` **함수** 호출식의 **계산**에 의해 결정돼요
    - 원래는 static이면 영생, automatic이면 '이번 호출이 끝나기 전까지'로 고정되어 있었는데
      프로그래머가 좀 더 능동적으로 칸의 목숨을 좌지우지할 수 있게 되었어요

    - 물론, 그만큼 프로그래머의 책임 또한 늘어났다고 볼 수 있겠지요?
      - 지금의 `main()`에서는 매 **반복**마다 마지막에 무조건 `Free()`를 하도록 적혀 있지만,
        이걸 빼먹고 안 적으면 '죽지 못 해 남아 있는 칸'이 점점 쌓이게 될 거예요
        - » 직접 해 보면 재미있을지도!

반자동

- 이제 마지막으로 CSP_5_3_yeshi_4.c를 열어 봅시다.

반자동

- 이제 마지막으로 CSP_5_3_yeshi_4.c를 열어 봅시다.
  - 우리가 방금 만든 바로 그 기능이 사실 C 라이브러리에 이미 포함되어 있었어요 (얘는 우리가 만든 것보다 더 효율적이긴 해요)
  - 사용법도 동일하고, free()를 안 했을 때의 고통 또한 동일하게 받아요

반자동

- 이제 마지막으로 CSP_5_3_yeshi_4.c를 열어 봅시다.
  - 우리가 방금 만든 바로 그 기능이 사실 C 라이브러리에 이미 포함되어 있었어요
    (얘는 우리가 만든 것보다 더 효율적이긴 해요)
  - 사용법도 동일하고, free()를 안 했을 때의 고통 또한 동일하게 받아요
  - 그럼 malloc() / free()는 '어디에 선언/정의해 둔 공간'에서 칸을 떼 오는 것일까요?

반자동

- 이제 마지막으로 CSP_5_3_yeshi_4.c를 열어 봅시다.
  - 우리가 방금 만든 바로 그 기능이 사실 C 라이브러리에 이미 포함되어 있었어요
    (얘는 우리가 만든 것보다 더 효율적이긴 해요)
  - 사용법도 동일하고, `free()`를 안 했을 때의 고통 또한 동일하게 받아요
  - 그럼 `malloc()` / `free()`는 '어디에 선언/정의해 둔 공간'에서 칸을 떼 오는 것일까요?
    - 컴파일러가 우리 몰래 뒤에서 살짝 큰 공간을 하나 마련해 두어요.
      - 이를 옛날에는 '자유 저장 공간(free storage space)'이라 불렀고, 요즘은 heap이라고 불러요!
    - 필요하다면 VS의 프로젝트 설정을 고쳐서
      '(컴파일러가 잡아 줄) 내 프로그램의 heap 크기'를 직접 지정해 줄 수 있어요!

정리

- 칸 잡기(allocate)
  - 자동: 우리가 적은 **선언**에 맞게, 딱 필요한 크기만큼만 잡힘
  - 수동: 미리 큰 공간을 잡아 놓고 나누어 쓰는 형태
  - 반자동: 수동 기법을 Code를 써서 포장해 둠
  - 지금 와서 돌이켜 생각해 보면,
    '**수식** 적는 방법'이 여전히 우리를 괴롭히긴 했지만, 개념 자체는 어렵지 않은 듯 해요

칸의 일생

- 집에 가기 전에, 일반적인 칸의 일생에 대해 살짝 맥락을 짚어 봅시다.
  - 그 과정에서, 방금 만든 반자동 칸 잡기 기능이 '칸권'에 미치는 영향을 생각해 보고, 이를 어느 정도 보완하기 위한 방법을 잠시 살펴 봐요

칸의 일생

- C에서 칸은 단순히 '**값**을 담아 두기 위한 존재'에 불과했어요.
  - int **값** 하나를 담아 두었다 나중에 쓰기 위해 int **변수** 하나를 **선언**

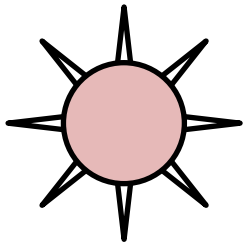
칸의 일생

- C에서 칸은 단순히 '**값**을 담아 두기 위한 존재'에 불과했어요.
  - int **값** 하나를 담아 두었다 나중에 쓰기 위해 int **변수** 하나를 **선언**

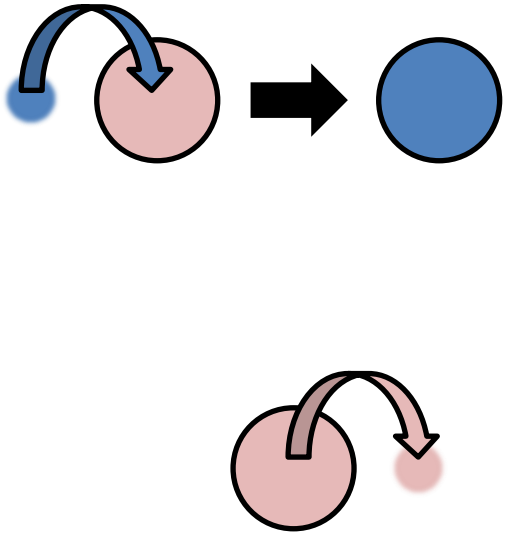
  - 이러한 측면에서, 어떤 칸 하나에 살짝 감정이입을 하면
    '칸의 일생'에 대해 다음 슬라이드와 같은 그림을 그려 볼 수 있을 거예요

칸의 일생

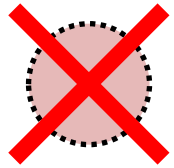
칸의 형성



칸 사용



칸의 소멸

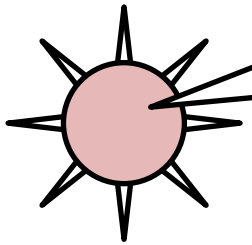


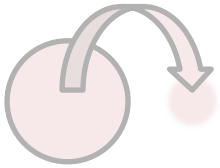
칸의 일생

칸의 형성

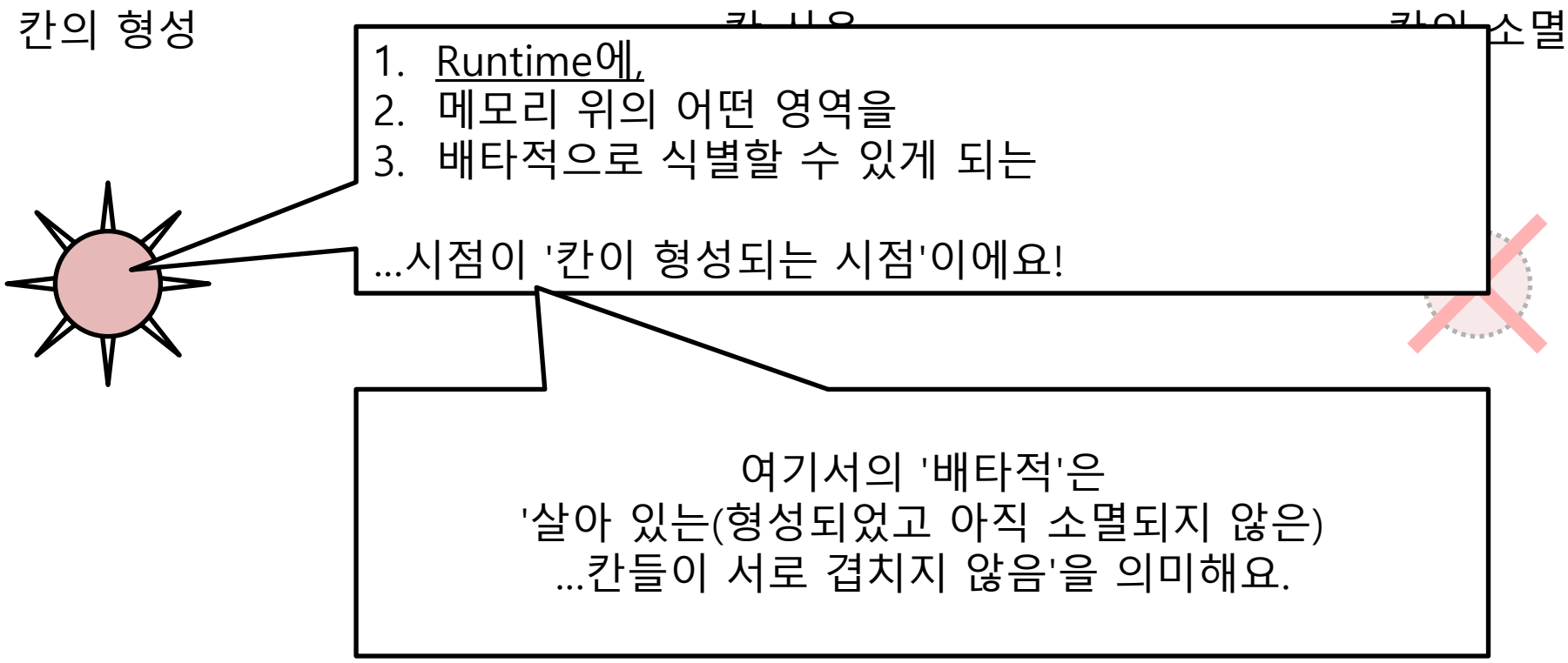
1. Runtime에,
2. 메모리 위의 어떤 영역을
3. 배타적으로 식별할 수 있게 되는

...시점이 '칸이 형성되는 시점'이에요!



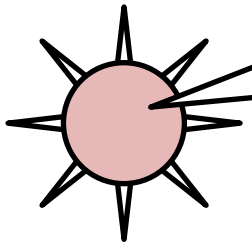


칸의 일생


칸의 형성
칸의 소멸
1. Runtime에,
2. 메모리 위의 어떤 영역을
3. 배타적으로 식별할 수 있게 되는
...시점이 '칸이 형성되는 시점'이에요!
여기서의 '배타적'은
'살아 있는(형성되었고 아직 소멸되지 않은)
...칸들이 서로 겹치지 않음'을 의미해요.

칸의 일생

칸의 형성



1. Runtime에,
2. 메모리 위의 어떤 영역을
3. 배타적으로 식별할 수 있게 되는

...시점이 '칸이 형성되는 시점'이에요!

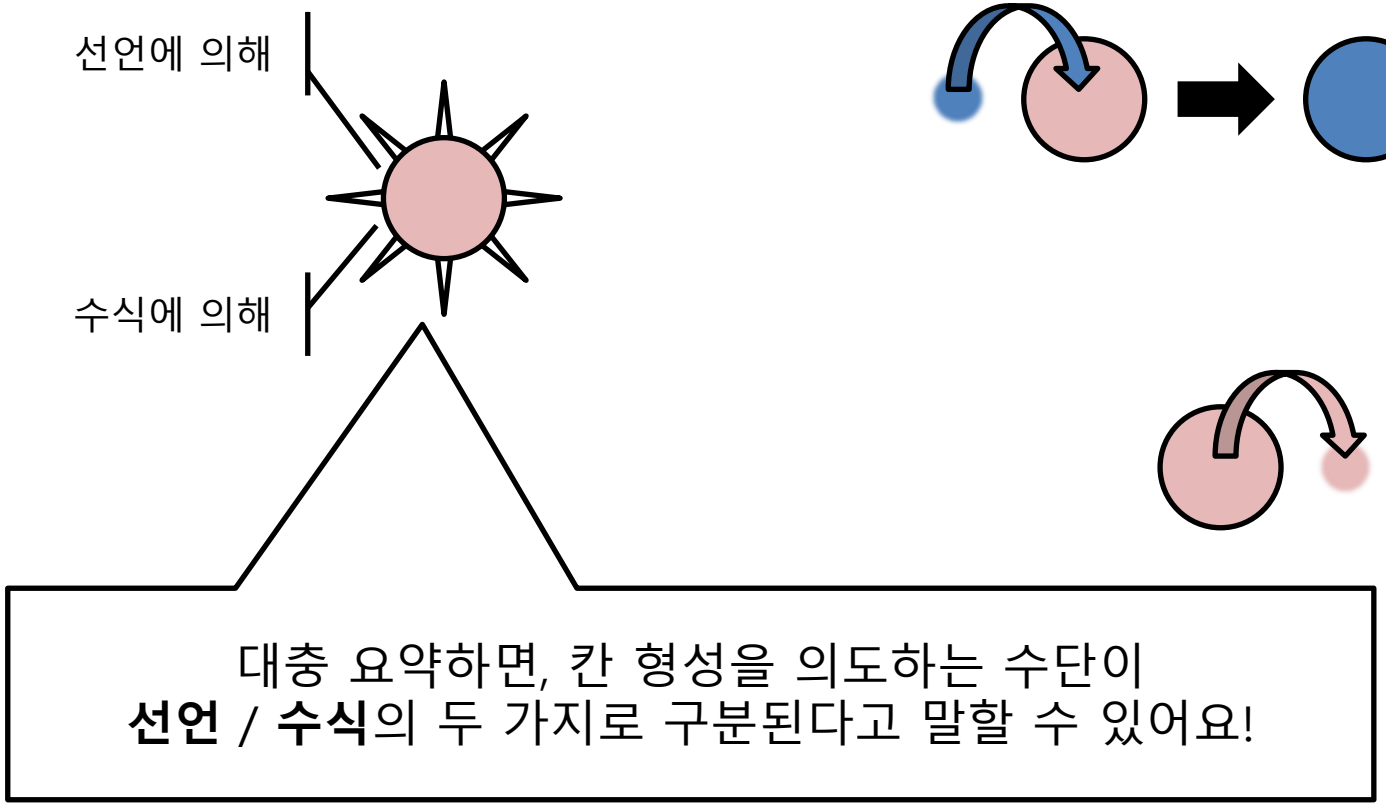
여기서의 '식별'은
'해당 칸에 대한 시작(기준) **위치 값**이 결정됨'을 의미해요.

칸의 일생

- 앞에서 본 세 가지 칸 잡는 방법들을 칸의 형성 관점에서 나열하면…
  - 자동
    - 프로그래머가 적어 둔 **선언**에 따라 적절한 시점에 형성되도록 컴파일돼요
      - Static 친구들은 아예 컴파일 도중에 정확한 **위치**가 결정돼요
      - Automatic 친구들은 **runtime**에, 해당 **선언**이 있는 { }로 노란 화살표가 진입하는 시점에 결정돼요
        - » 그렇긴 하지만 **offset 값**은 컴파일 도중에 결정되긴 해요
  - 수동 및 반자동
    - **선언**에 의해 칸이 형성되지 않아요. 오직 **수식**에 의해서만 칸이 형성돼요.
      - **Runtime**에 `malloc()` 호출식을 계산하는 시점에 정확한 **위치**가 결정돼요
        - » Automatic 친구들과 달리 컴파일 도중에는 **위치**는 커녕 **offset 값**조차 결정되지 않아요!

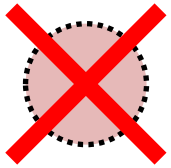
칸의 일생

칸의 형성



칸 사용

칸의 소멸

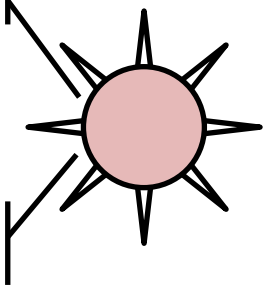


대충 요약하면, 칸 형성을 의도하는 수단이
선언 / **수식**의 두 가지로 구분된다고 말할 수 있어요!

칸의 일생

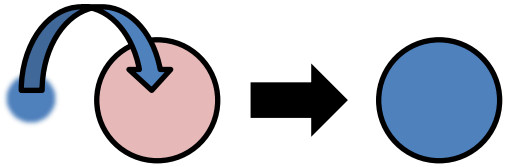
칸의 형성

선언에 의해

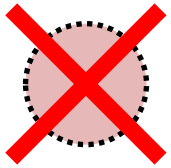


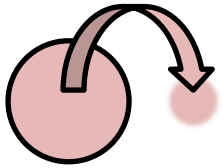
수식에 의해

칸 사용



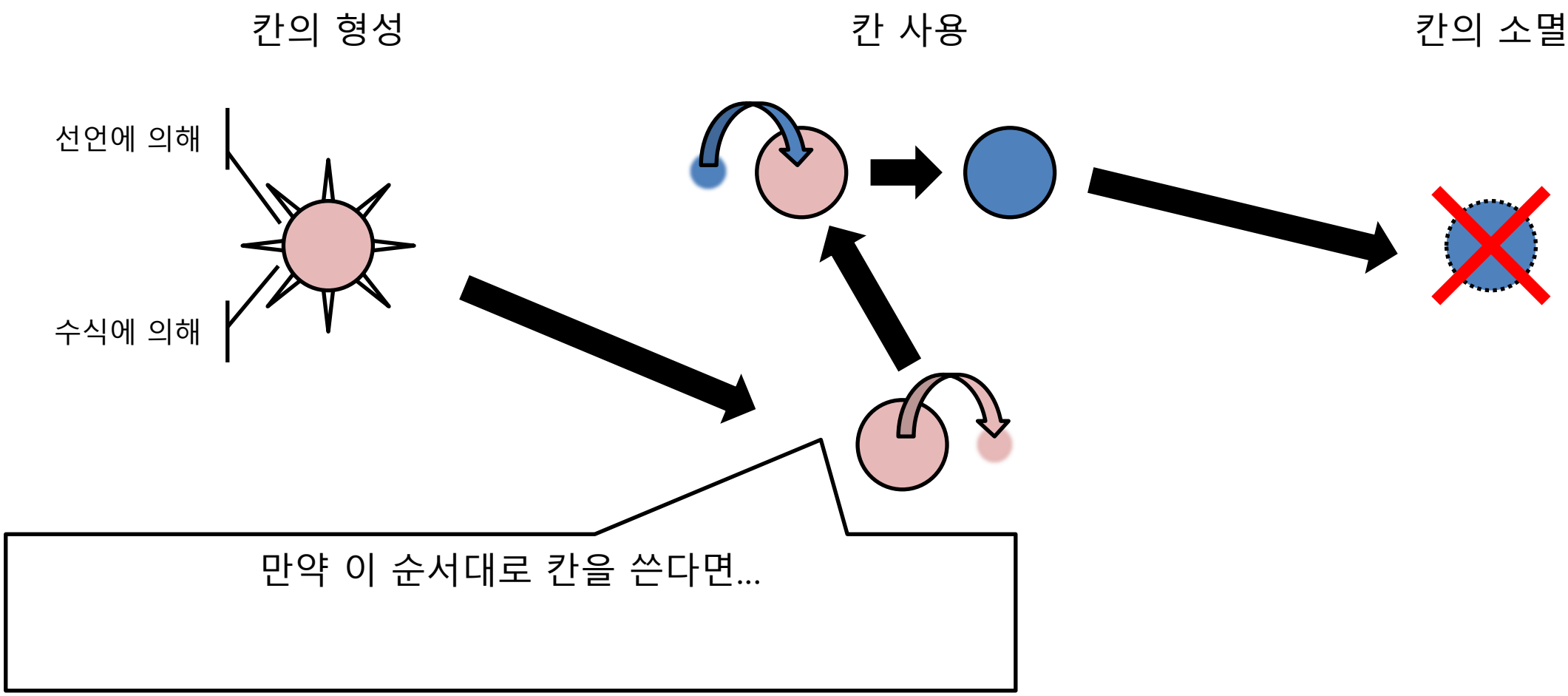
칸의 소멸



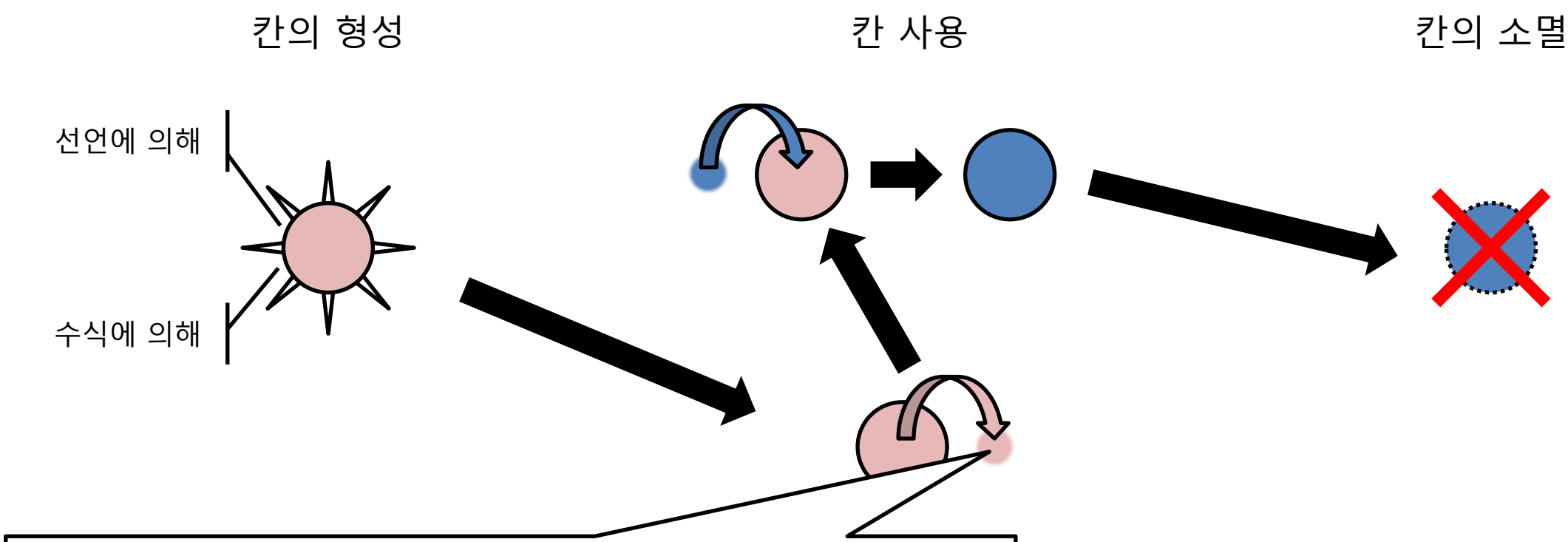


이제 칸이 사용당하는 두 가지 핵심 작업을 생각해 봅시다.
(칸에 **값** 담기, 칸에 담겨 있는 **값** 읽기)

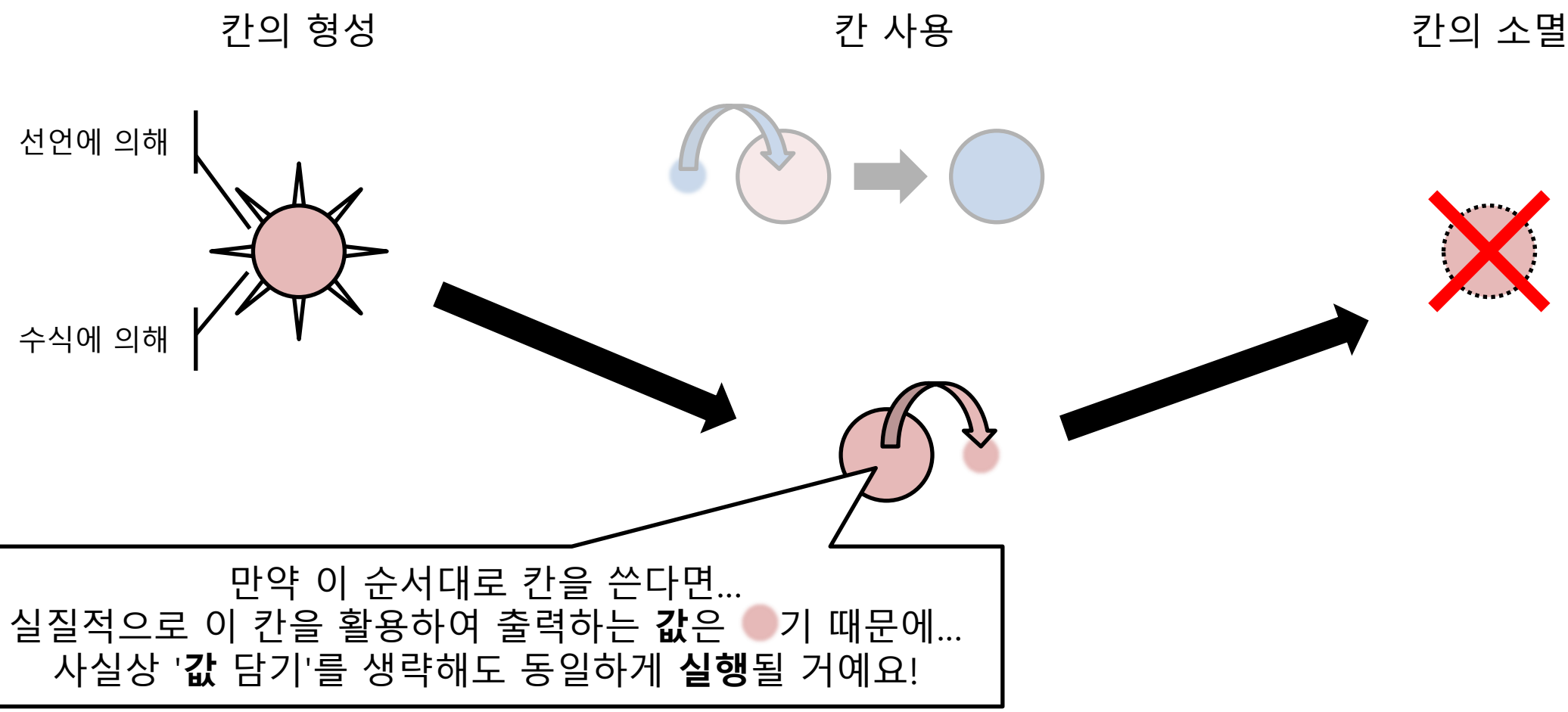
칸의 일생



칸의 일생


칸의 형성
칸 사용
칸의 소멸
선언에 의해
수식에 의해
만약 이 순서대로 칸을 쓴다면...
실질적으로 이 칸을 활용하여 출력하는 값은 기 때문에...

칸의 일생


칸의 형성
칸 사용
칸의 소멸
선언에 의해
수식에 의해
만약 이 순서대로 칸을 쓴다면...
실질적으로 이 칸을 활용하여 출력하는 **값**은 기 때문에...
사실상 '**값** 담기'를 생략해도 동일하게 **실행**될 거예요!

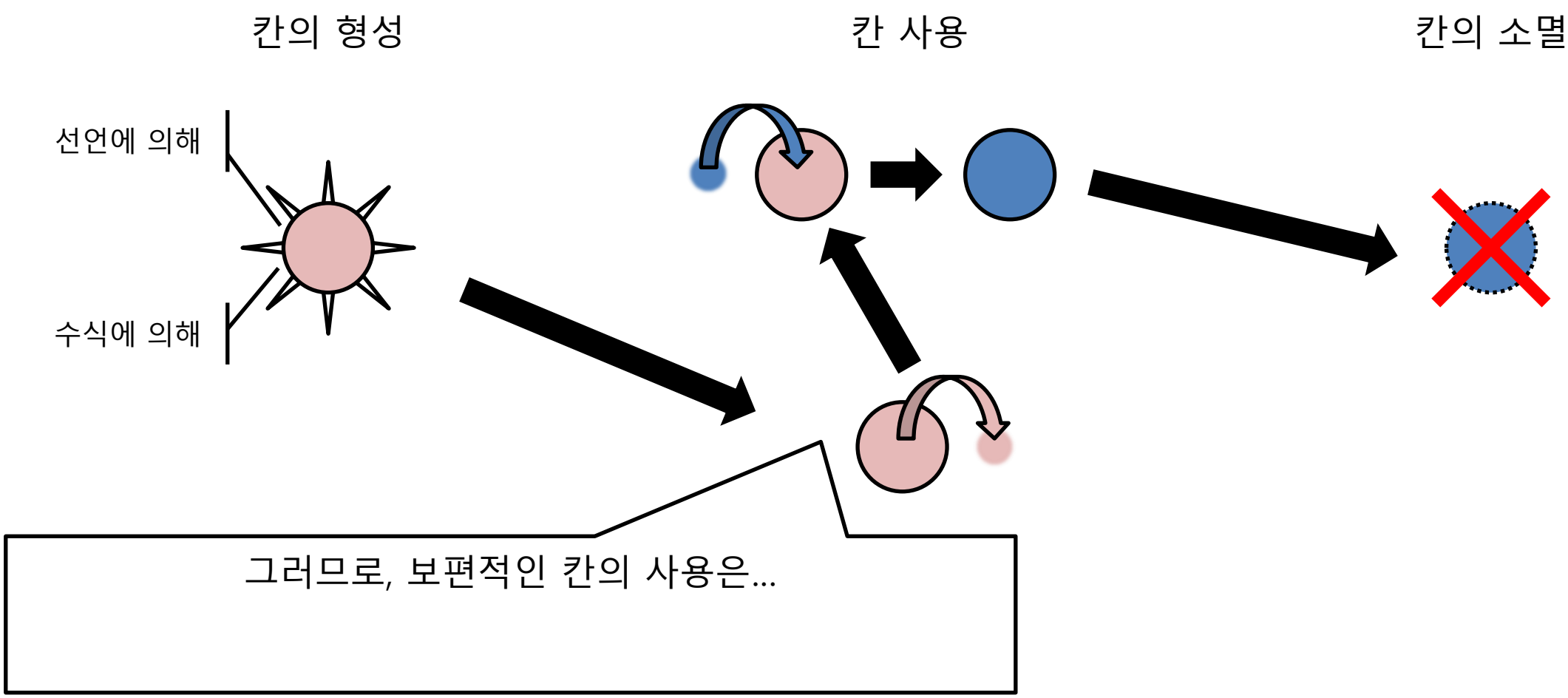
칸의 일생

- 넓은 관점에서는 Read → Write 순서로 읽는 게 일반적이에요.
  - 입/출력, CD-RW 등등 여러 표현에서 발견 가능

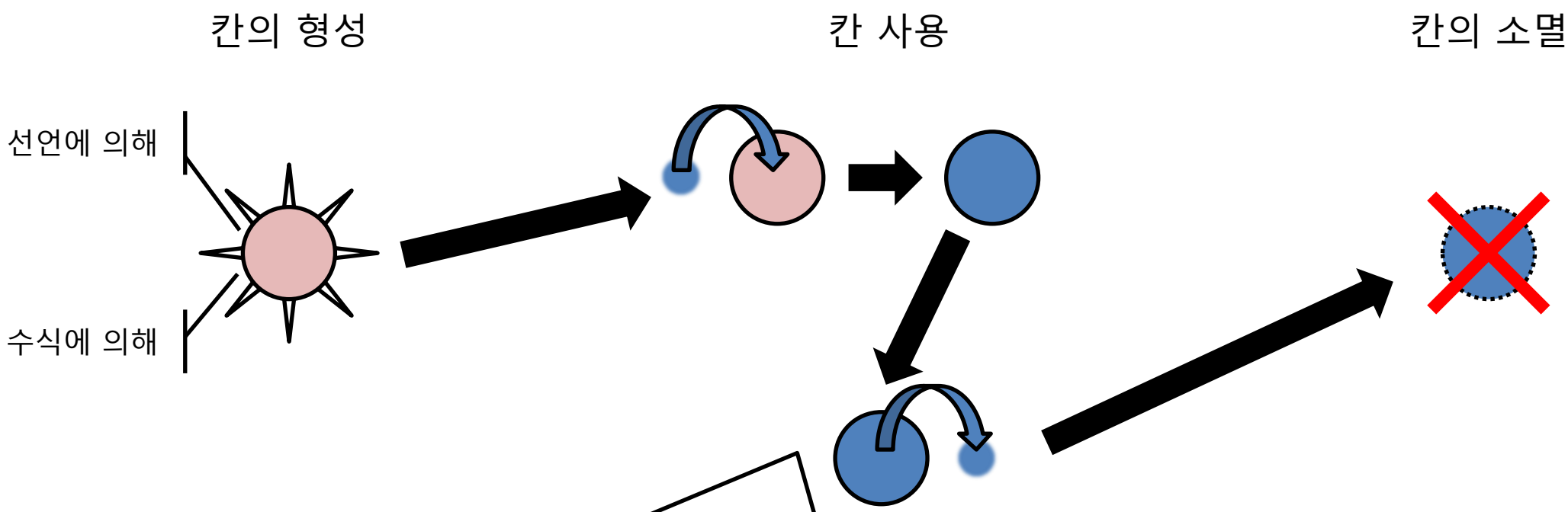
- (나름 중요)칸 관점에서는 이를 반대로 해석하게 돼요.
  - scanf()는 키보드에서 '읽어서' 내가 의도한 칸에 '쓰는' 작업을 하고 싶을 때 호출해요
  - printf()는 칸에 담긴 값을 '읽어서' 검은 창에 '쓰는' 작업을 하고 싶을 때 호출해요

  - 어떤 칸을 중심에 놓고 봤을 때는 WR이 일반적임!

칸의 일생

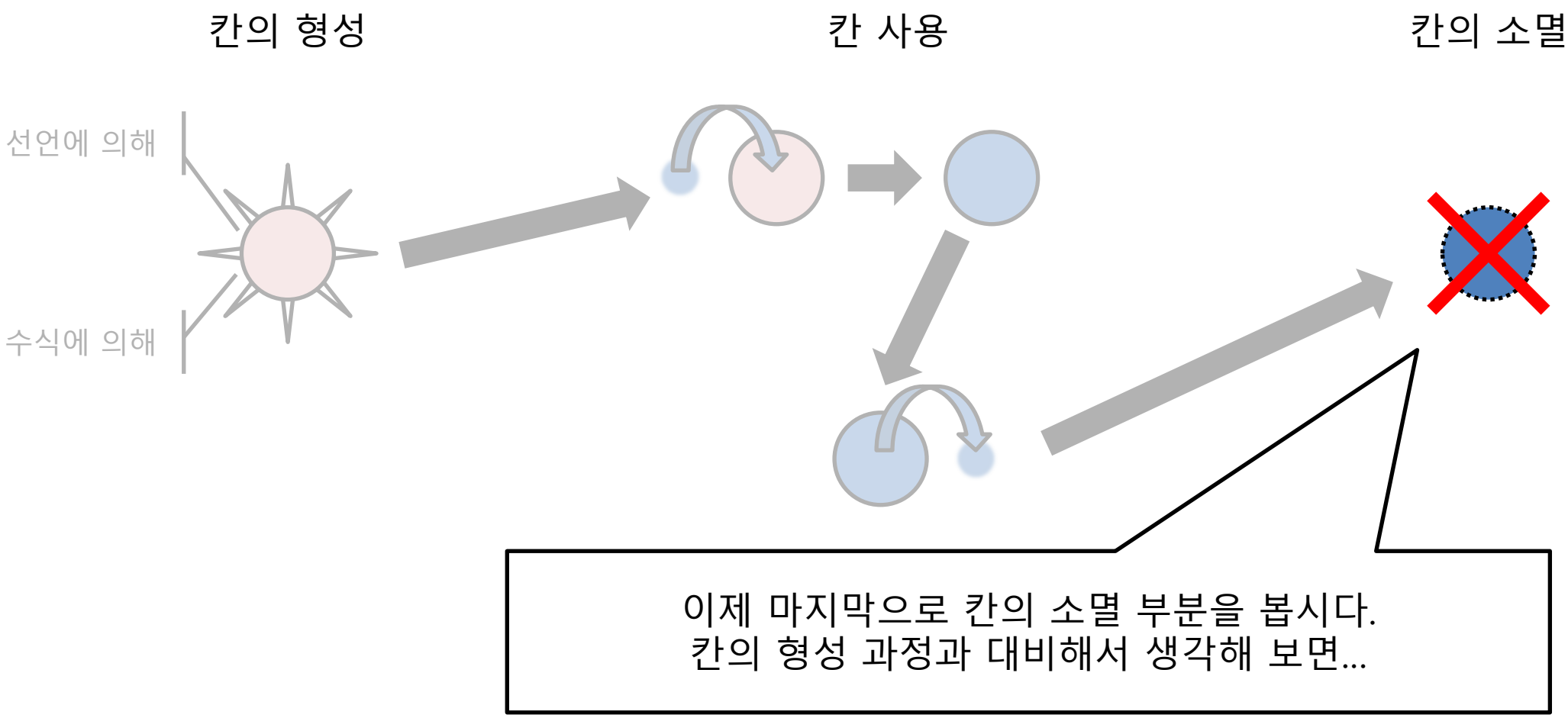


칸의 일생



그러므로, 보편적인 칸의 사용은...
...이런 흐름을 띄게 된다고 말할 수 있어요!
(나중에 읽지 않을 값을 지금 담아 둘 필요는 전혀 없음!)

칸의 일생

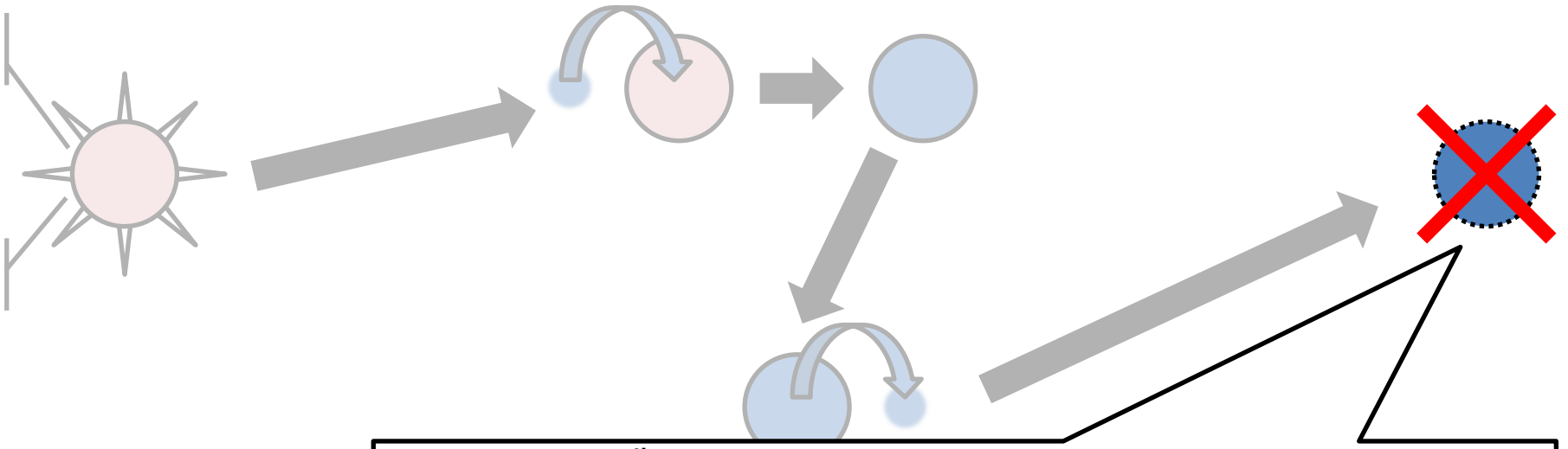

칸의 형성
칸 사용
칸의 소멸
선언에 의해
수식에 의해
이제 마지막으로 칸의 소멸 부분을 봅시다.
칸의 형성 과정과 대비해서 생각해 보면...

칸의 일생

칸의 형성 칸 사용 칸의 소멸

선언에 의해

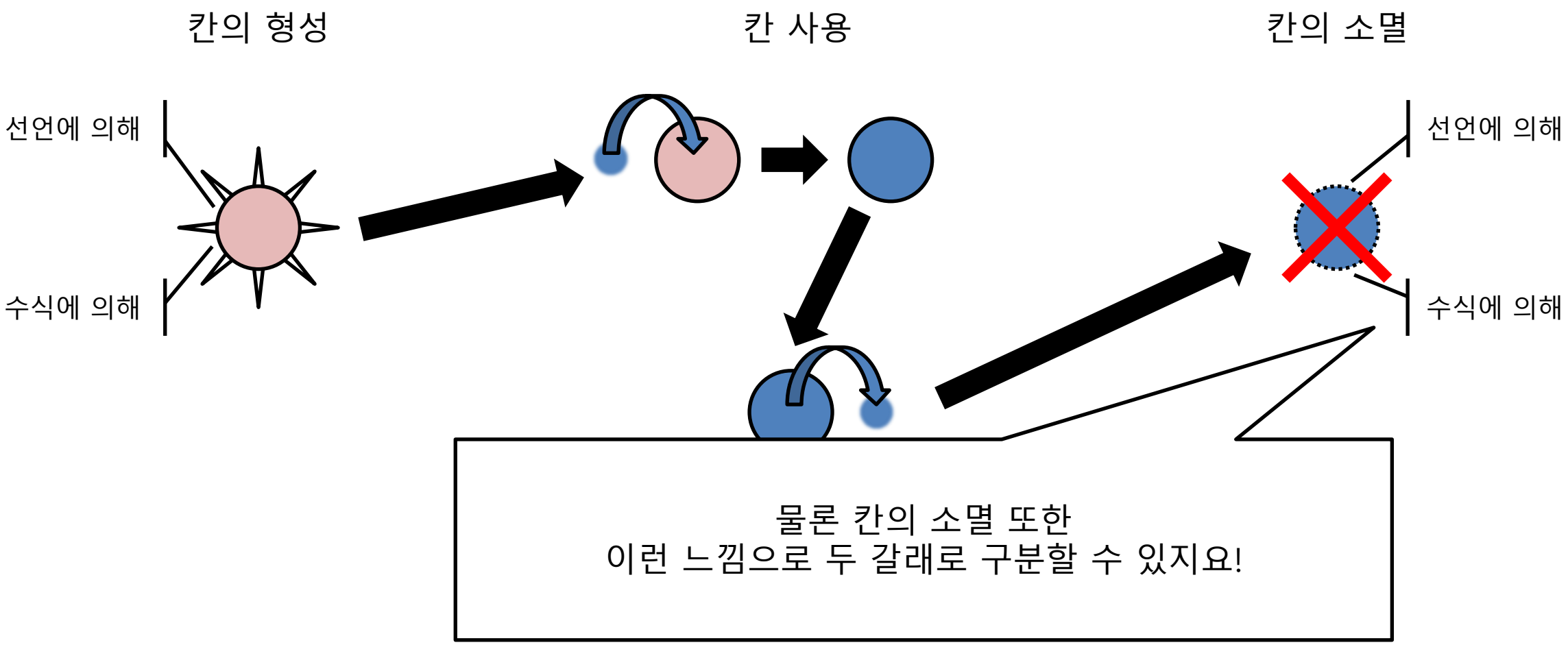
수식에 의해



1. Runtime에,
2. 메모리 위의 특정 영역에 대한
3. 원래 가지고 있던 배타적 권한을 상실하는

...시점을 '칸이 소멸되는 시점'이라 말할 수 있어요!

칸의 일생


칸의 형성
칸 사용
칸의 소멸
선언에 의해
수식에 의해
선언에 의해
수식에 의해
물론 칸의 소멸 또한
이런 느낌으로 두 갈래로 구분할 수 있지요!

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));

    free(ptr);


}
```

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));

    free(ptr);


}
```

이 친구는 static 친구예요.
main()을 호출하기도 전에 칸이 형성되고,
main()이 끝난 다음에야 칸이 소멸해요.
→ '내 코드' 관점에서는 얘는 영생을 누린다 볼 수 있음

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));

    free(ptr);


}
```

이 친구는 func() **정의** 안에 **선언**된 automatic 친구예요.
func()를 호출하는 시점에 칸이 형성되고,
func() **정의** 내용물 **실행**이 끝난 시점에 칸이 소멸해요.

칸의 일생

- 잠시 C 코드로 예시를 들어 보면…

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));
    printf("%d", number);
    free(ptr);

    printf("%d", number);
}
```

이 친구는 func() **정의** 안에 **선언**된 automatic 친구예요.
func()를 호출하는 시점에 칸이 형성되고,
func() **정의** 내용물 **실행**이 끝난 시점에 칸이 소멸해요.
→ 적어도 local **이름** number가 유효한 동안에는 살아 있음!

5가 출력됨!

5가 출력됨!

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));

    free(ptr);

}
```

이 친구는 func() 정의 안에 선언된 automatic 친구예요.
func()를 호출하는 시점에 칸이 형성되고,
func()의 실행이 끝난 시점에 칸이 소멸해요.
→ 그러므로 {에서 형성, }에서 소멸한다 여겨도 좋아요.

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));

    free(ptr);


}
```

여기서 malloc()을 호출함으로써
반자동으로 int 한 칸을 형성하고 있어요.
해당 칸에 대한 시작 **위치**를 ptr에 담아 놓고 있지요.

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));

    free(ptr);


}
```

그리고 이후에 free()를 호출함으로써
방금 형성한 칸을 소멸시키고 있어요.

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;

    int *ptr = malloc(sizeof(int));

    free(ptr);


}
```

그리고 이후에 free()를 호출함으로써
방금 형성한 칸을 소멸시키고 있어요.
→ 따라서 이 칸은 이 두 **수식**이 **계산**되는 시점 '사이'에만
살아 있다고 말할 수 있어요!

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;
    int *ptr = malloc(sizeof(int));
    *ptr = 7;
    printf("%d", *ptr);
    free(ptr);
    *ptr = 11;
    printf("%d", *ptr);
}
```

그리고 이후에 free()를 호출함으로써
방금 형성한 칸을 소멸시키고 있어요.
→ 따라서 이 칸은 이 두 수식이 계산되는 시점 '사이'에만
살아 있다고 말할 수 있어요!

적법한 배타적 공간 사용!

적법하지 않은 사용!

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```c
int number = 3;

void func()
{
    int number = 5;
    int *ptr = malloc(sizeof(int));
    *ptr = 7;
    printf("%d", *ptr);

    *ptr = 11;
    printf("%d", *ptr);
}
```

만약 free() 호출식을 적지 않는다면...

칸의 일생

- 잠시 C 코드로 예시를 들어 보면...

```
int number = 3;

void func()
{
    int number = 5;
    int *ptr = malloc(sizeof(int));
    *ptr = 7;
    printf("%d", *ptr);

    *ptr = 11;
    printf("%d", *ptr);
}
```

만약 free() 호출식을 적지 않는다면...
형성된 칸의 소멸이 발생하지 않았으므로
해당 칸 입장에서는 불사의 몸이 된다고 볼 수 있어요!

이 코드에서는, func()의 **실행**이 끝나며 **이름** ptr에 대한 칸이 소멸되면
더 이상 malloc()으로 형성한 칸의 존재를 식별할 수 있는 수단이 없게 돼요.
소멸당하지 않는다고 좋은 게 아님!

'칸권'

- 반자동 방식을 써서 칸을 다룰 때는 '칸권'에 대한 위험성이 생겨요.
  - **선언**에 의존하는 자동 방식은 컴파일러가 알아서 칸의 형성 및 소멸 시점을 챙겨 줘요
    - **선언**함 → **정의**해 줌
      → 적당한 때 형성되고 (그 **이름**을 사용할 수 있는 동안은 유효하다) 적당한 때 소멸됨(정상)
    - **선언** 안 함 → **정의**도 안 해 줌 → 아무 일도 일어나지 않음(정상)
  - 반자동 방식은 컴파일러가 제공하는 그런 서비스를 전혀 누리지 못해요.
    따라서, '죽어야 하는 칸'은 반드시 프로그래머가 **수식**을 잘 적어서 죽여 주어야 해요
    - '소멸당할 권리'는 칸권의 중요한 요소들 중 하나예요.
      (물론 반대로 '형성되지도 않았는데 소멸을 요구받는' 상황도 문제가 되기는 해요)

'칸권'

- 반자동 방식을 써서 칸을 다룰 때는 '칸권'에 대한 위험성이 생겨요.
  - **선언**에 의존하는 자동 방식은 컴파일러가 알아서 칸의 형성 및 소멸 시점을 챙겨 줘요
    - **선언**함 → **정의**해 줌
      → 적당한 때 형성되고 (그 **이름**을 사용할 수 있는 동안은 유효하다) 적당한 때 소멸됨(정상)
    - **선언** 안 함 → **정의**도 안 해 줌 → 아무 일도 일어나지 않음(정상)
  - 반자동 방식은 컴파일러가 제공하는 그런 서비스를 전혀 누리지 못해요.
    따라서, '죽어야 하는 칸'은 반드시 프로그래머가 **수식**을 잘 적어서 죽여 주어야 해요
    - '소멸당할 권리'는 칸권의 중요한 요소들 중 하나예요.
      (물론 반대로 '형성되지도 않았는데 소멸을 요구받는' 상황도 문제가 되기는 해요)
  - 이외에도 위험 요인이 하나 더 있는데...

'칸권'

- 아래 코드를 구경해 봅시다:

```
void func()
{
    int number;

    number = 3;

    printf("%d", number);


}
```

```
void func()
{
    int *ptr = malloc(sizeof(char));

    *ptr = 3;

    printf("%d", *ptr);

    free(ptr);
}
```

'칸권'

- 아래 코드를 구경해 봅시다:

```
void fun
{
    int number;

    number = 3;

    printf("%d", number);


}
```

'3 담을 칸'을 의도하는 **선언**이에요.

```
void fun
{
    int *ptr = malloc(sizeof(char));

    *ptr = 3;

    printf("%d", *ptr);

    free(ptr);
}
```

'3 담을 칸'을 의도하는 **수식**이에요.

'칸권'

- 아래 코드를 구경해 봅시다:

```
void func()
{
    int number = 3;

    printf("%d", number);

}
```

그 칸에 3을 담는 **문장**이에요.

```
void func()
{
    int *ptr = 3;

    printf("%d", *ptr);

    free(ptr);
}
```

그 칸에 3을 담는 **문장**이에요.

'칸권'

- 아래 코드를 구경해 봅시다:

```
void func()
{
    int number;

    number = 3;

    printf("%d", number);

}
```

그 칸에 담긴 3을 출력하는 **문장**이에요.

```
void func()
{
    int *ptr = (int*)malloc(sizeof(int));

    *ptr = 3;

    printf("%d", *ptr);

    free(ptr);
}
```

그 칸에 담긴 3을 출력하는 **문장**이에요.

'칸권'

- 아래 코드를 구경해 봅시다:

```c
void func()
{
    int number;

    number = 3;

    printf("%d", number);

}
```

출력이 끝나면 그 칸은 소멸돼요.

```c
void func()
{
    int *ptr = malloc(sizeof(char));

    *ptr = 3;

    printf("%d", *ptr);

    free(ptr);
}
```

출력이 끝나면 그 칸은 소멸돼요.

'칸권'

- 아래 코드를 구경해 봅시다:

```c
void func()
{
    int number;

    number = 3;

    printf("%d", number);


}
```

```c
void func()
{
    int *ptr = malloc(sizeof(char));

    *ptr = 3;

    printf("%d", *ptr);

    free(ptr);
}
```

두 코드의 차이점이 눈에 보이나요?

'칸권'

- 아래 코드를 구경해 봅시다:

```
void func()
{
    int number;

    number = 3;

    printf("%d", number);


}
```

```
void func()
{
    int *ptr = malloc(sizeof(char));

    *ptr = 3;

    printf(      tr);
```

맞아요. 의도상 'int 한 칸'을 형성했어야 함에도 불구하고
그에 못 미치는 크기만큼을 잡아 사용하고 있어요.
(소위 말하는 'segmentation fault'의 원인이 됨!)

'칸권'

- 아래 코드를 구경해 봅시다:

```
void func()
{
    int number;

    number = 3;

    printf("%d", number);

}
```

```
void func()
{
    int *ptr = malloc(sizeof(char));

    *ptr = 3;

    printf("%d", *ptr);
```

이런 일은 자동 방법을 쓸 때는 절대 발생하지 않을 것 같아요.
내가 적은 int **변수 선언**으로 인해 형성되는 칸은
반드시 딱 int **값** 하나 들어갈 만한 크기를 가질 테니까요.
(이게 나중에 발목을 잡긴 할 텐데, 아무튼 지금은 편하고 좋아요)

'칸권'

- 반자동 방식을 써서 칸을 다룰 때는 '칸권'에 대한 위험성이 생겨요.
  - malloc()의 가장 큰 단점은,
    '만들려는 칸에 담을 **값**의 **형식**'에 대한 정보를 전혀 감안하지 않는다는 것이에요
    - 단순히 '몇B짜리 새 칸을 만들 건지'만 인수로 정해요

  - 그렇다 보니,
    자동 방식은 똑 소리 나는 컴파일러가 알아서 잘 **정의**해 주므로 걱정 없었지만,
    반자동 방식을 쓰기 시작하면서 'int 한 칸' 같은 칸 정체성이 모호해지기 시작해요
    - 물론 프로그래머가 마음먹으면 char 여덟 칸을 double 한 칸으로 써도 되는게 C의 세계지만,
      아예 칸이 형성되는 시점부터 정체성이 위협받는 것은 칸권에 있어 그리 좋지 않아요

'칸권'

- 반자동 방식을 써서 칸을 다룰 때는 '칸권'에 대한 위험성이 생겨요.
  - malloc()의 가장 큰 단점은,
    '만들려는 칸에 담을 **값**의 **형식**'에 대한 정보를 전혀 감안하지 않는다는 것이에요
    - 단순히 '몇B짜리 새 칸을 만들 건지'만 인수로 정해요

  - 그렇다 보니,
    자동 방식은 똑 소리 나는 컴파일러가 알아서 잘 **정의**해 주므로 걱정 없었지만,
    반자동 방식을 쓰기 시작하면서 'int 한 칸' 같은 칸 정체성이 모호해지기 시작해요
    - 물론 프로그래머가 마음먹으면 char 여덟 칸을 double 한 칸으로 써도 되는게 C의 세계지만,
      아예 칸이 형성되는 시점부터 정체성이 위협받는 것은 칸권에 있어 그리 좋지 않아요

  - 이에 대한 약간의 대안을 생각해 본다면...

'칸권'

- 아래 코드를 구경해 봅시다:

```
struct Stat { int str, dex, con; };

struct Stat *New_Stat(
    int str, int dex, int con) {
    struct Stat *result
        = malloc(sizeof(struct Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}

void Delete_Stat(struct Stat *stat) {
    free(stat);
}
```

```
void func1()
{
    struct Stat *stat
        = malloc(sizeof(struct Stat));
    stat->str = 30;
    stat->dex = 50;
    stat->con = 20;
    free(stat);
}

void func2()
{
    struct Stat *stat = New_Stat(30, 50, 20);
    Delete_Stat(stat);
}
```

'칸권'

- 아래 코드를 구경해 봅시다:

```c
struct Stat { int str, dex, con; };

struct Stat *New_Stat(
    int str, int dex, int con) {
    struct Stat *result
        = malloc(sizeof(struct Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}

void Delete_Stat(struct Stat *stat) {
    free(stat);
}
```

새로운 능력치 칸 하나를 반자동 방식으로 얻으려 할 때
아마도 필연적으로 적게 될 코드들을

'**구조체 정의**를 적는 시점'에 미리 딱 **함수**로 만들어 놨어요.

뭐 만드는 김에 대칭성의 아름다움을 뽐내기 위해
능력치 칸 하나를 소멸시키는 **함수**도 같이 만들어 놨어요.

'칸권'

- 아래 코드를 구경해 봅시다:

```
struct Stat { int str, dex, con; };

Stat* New_Stat(int str, int dex, int con)
{
    Stat* result = (Stat*)malloc(sizeof(Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}
```

그래서,
원래대로라면 새 능력치 칸 하나를 형성할 때마다
이런 느낌으로 각 내용을 오타 없이 잘 적어야 했지만...

미리 정의해 둔 '능력치 한 칸용 **함수**'가 있다면
새 능력치 칸 하나를 형성할 때마다
이런 식으로 좀 더 간편하고 오타 위험이 적은 **수식**을
대신 적을 수 있게 될 거예요!

```
void func1()
{
    struct Stat *stat
        = malloc(sizeof(struct Stat));
    stat->str = 30;
    stat->dex = 50;
    stat->con = 20;
    free(stat);
}

void func2()
{
    struct Stat *stat = New_Stat(30, 50, 20);
    Delete_Stat(stat);
}
```

'칸권'

- (중요)malloc()의 맹점이었던 '**형식** 정보의 부재'는,
  '이 **형식** 칸만을 위한 **함수**'를 별도로 미리 만들어 놓음으로써
  어느 정도 완화할 수 있어요.
  - 프로그래머들끼리, 또는 미래의 자기 자신과
    '이 **구조체 형식** 칸 만들 때는 꼭 이 **함수**를 호출해요'와 같이 약속을 해 둔다면
    적어도 '4B 필요한데 오타 내서 1B만 잡아 쓰는' 상황은 막을 수 있어요

'칸권'

- (중요)malloc()의 맹점이었던 '**형식** 정보의 부재'는,
  '이 **형식** 칸만을 위한 **함수**'를 별도로 미리 만들어 놓음으로써
  어느 정도 완화할 수 있어요.
  - 프로그래머들끼리, 또는 미래의 자기 자신과
    '이 **구조체 형식** 칸 만들 때는 꼭 이 **함수**를 호출해요'와 같이 약속을 해 둔다면
    적어도 '4B 필요한데 오타 내서 1B만 잡아 쓰는' 상황은 막을 수 있어요

- (다음 시간 예고)그리고, C++는 한 단계 더 나아가서,
  C++ 컴파일러가 프로그래머에게 위의 약속을 반드시 지킬 것을 요구하도록
  설계되어 있어요!
  - 칸 입장에서 보면, '12B짜리 모호한 공간'으로서 존재하는 시점이 사라지고
    처음 initialize될 때부터 '제대로 세트된 능력치 한 칸'으로서의 정체성을 갖게 되는,
    매우 살기 좋은 환경이 되었다 말할 수 있을 거예요. → 코드로 다시 가서 보면...

'칸권'

- 아래 코드를 구경해 봅시다:

```
struct Stat { int str, dex, con; };

Stat* New_Stat(int str, int dex, int con)
{
    Stat* result = (Stat*)malloc(sizeof(Stat));
    result->str = str;
    result->dex = dex;
    result->con = con;
    return result;
}
```

이 **문장**들이 다 **실행**된 이후 시점에서야,
세 능력치가 다 담긴
'온전한 능력치 한 칸'으로 존재하게 돼요.

이 **선언**의 initializer 자리 **수식**만 **계산**해도,
세 능력치가 다 담긴
'온전한 능력치 한 칸'으로서 존재할 수 있어요!

```
void func1()
{
    struct Stat *stat
        = malloc(sizeof(struct Stat));
    stat->str = 30;
    stat->dex = 50;
    stat->con = 20;
    free(stat);
}

void func2()
{
    struct Stat *stat = New_Stat(30, 50, 20);
    Delete_Stat(stat);
}
```

Object의 의미 확장

- 음... 아직은 감이 잘 안 오는 게 정상이긴 해요.
  - func1()에서 직접 적은 **문장**들과 같은 내용이 New_Stat()에 적혀 있기 때문에, New_Stat()을 호출하도록 설계된 func2()가 더 간단해보이는 게 당연하지요.

Object의 의미 확장

- 음... 아직은 감이 잘 안 오는 게 정상이긴 해요.
  - func1()에서 직접 적은 **문장**들과 같은 내용이 New_Stat()에 적혀 있기 때문에, New_Stat()을 호출하도록 설계된 func2()가 더 간단해보이는 게 당연하지요.

- 지금 느끼는 약간의 당혹감이,
  C 관점에서 C++ 관점으로 옮겨가기 위한 좋은 시작점이라 여겨도 좋아요.
  - C에서 단어 **object**는 단순히 '칸'을 의미했지만,
    C++에서는 **object**를 '**값** + 칸'으로 바라보는 메타를 채용해 두었어요

Object의 의미 확장

- 음... 아직은 감이 잘 안 오는 게 정상이긴 해요.
  - func1()에서 직접 적은 **문장**들과 같은 내용이 New_Stat()에 적혀 있기 때문에, New_Stat()을 호출하도록 설계된 func2()가 더 간단해보이는 게 당연하지요.

- 지금 느끼는 약간의 당혹감이,
  C 관점에서 C++ 관점으로 옮겨가기 위한 좋은 시작점이라 여겨도 좋아요.
  - C에서 단어 **object**는 단순히 '칸'을 의미했지만,
    C++에서는 **object**를 '**값** + 칸'으로 바라보는 메타를 채용해 두었어요

  - 방금 본 New_Stat()같은 **함수**가, '새 칸에 지정된 새 **값**을 넣어 발송'함으로써 '능력치 칸임에도 능력치 **값**이 들어 있지 않은 상황을 배제'시키려는,
    C++에서의 **object** 관점을 추구하는 **함수**라 생각해 보면 좋을 듯 해요!
    - 일단은 그런가보다 하고 다음 시간 내용으로 돌입해 보면 좋겠어요

정리

- '칸을 잡는(allocate하는)' 방법으로 자동, 수동, 반자동이 있어요.
  - 자동: **선언**해 둔 대로 적당히 칸이 살아 있도록 컴파일러가 **정의**해 줘요
  - 수동: 큰 칸(아마도 **배열**?)의 일부를 직접 '여긴 int 한 칸임' 하고 정해 쓸 수 있어요
  - 반자동: 수동 기능을 **함수**로 포장해 둔 버전이에요. `malloc()` / `free()`도 마찬가지임
- C의 반자동 기능은 칸권 침해의 위험성을 가져요.
  - 자칫하면 칸의 '소멸할 권리'가 잘 지켜지지 않을 수 있어요
  - 자칫하면 칸이 의도한 크기만큼 잡히지 않을 수 있어요
- C++에서는 단어 object를 '값 + 칸'으로서 다룬다는 듯 해요.
  - 반자동임에도 `New_Stat()`를 호출해 쓰면 칸권 침해를 어느 정도 방지할 수 있어요
  - 더불어, '능력치 칸에 능력치 값이 담겨 있지 않는' 시점을 배제하는 효과도 있는 듯?

마무리

- 이제 우리는 C++ 동네로 넘어가기 위한 첫 관문을 지났어요.

- 일단 오늘은 느낀 점 적고 퇴근합시다.
  다음 시간에는 또 다른 관점에서의 도약이 기다리고 있을 거예요.