10-1. 소속 관계, 포함 관계, static

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

9일차 내용

- '다시 보기' 시간을 가졌어요.
  - 중간고사 답안 확인 및 시상
  - **Object** 다시 보기
    - 칸권 세 가지를 다시 짚어봤어요
    - **생성자**, **파괴자** 구경해 봤어요
  - **클래스** 다시 보기
    - **클래스** = Data & Code, 별 거 아니었어요
    - 숨어 있던 키워드 this를 구경해 보았어요

9일차 내용

- 진정해요

9일차 내용

- 진정해요
  - 1일차 시간에 갑자기 **문장 실행 수식 계산 값 연산자** 등등 막 튀어나왔지만
    전반부 내내 그럭저럭 일관적으로 다루어 보면서
    지금은 그 때보다는 구경하기 수월하다고 느낄 거예요

  - 지금 보는 내용들도,
    후반부 기간동안 같이 다루다 보면 나름 익숙해지게 될 거예요

오늘 내용

- **이름**'들' 사이의 관계, **object**'들' 사이의 관계를 상상해 보고, nonstatic/static Data/Code **멤버**의 특징을 짚어 봅니다.
  - 컴파일러가 어떤 **이름**에 대한 **선언**을 context에 입각하여 찾는(**lookup**하는) 방법 소개

- **Object**에 대한 **소유**권(ownership) 이야기를 구경해 봅니다.
  - **소유**의 의의, **소유**의 주체(**소유**권자), **object** 공유(또는, **이동**)
  - 그 과정에서 아래 내용을 함께 체크해 봅니다:
    - `new` / `delete` / `delete[]` **연산자**
    - **Reference**, `std::move<>()`
    - `auto` specifier

- 오늘은 최종 목표가 있어요.

이번 시간에는

- **이름**'들' 사이의 관계인 **소속** 관계를 구경해 봅니다.
  - 그 과정에서 **lookup** 이야기를 드디어 소개해요
  - 그 과정에서 나름 중요한 숙어인 ODR(One-Definition Rule)을 드디어 소개해요

- **Object**'들' 사이의 **포함** 관계를 구경해 봅니다.
  - 영속적으로 **포함**될 수도 있고 잠시 들고 있을 수도 있는데 여기서는 전자에 집중해요
  - 이건 워낙 직관적이라 그리 어렵지 않을 거예요

- Nonstatic Data **멤버**,
  static Data **멤버**,
  nonstatic Code **멤버**,
  static Code **멤버**
  …의 특징을 짚어 봅니다.

이번 시간에는

- 이번 시간은 예시 코드 하나를 펼쳐 놓고 구경해본 다음 마지막에 슬라이드로 정리해보려 해요.
  - 일단 CSP_10_1_yeshi.cpp를 탑재해 열어 봅시다
  - 지난 시간보다는 가벼운 편이니 너무 걱정 말아요

복습을 위한 슬라이드

- **소속** 관계
  - **이름**들 사이에 성립하는 관계입니다

  - S가 **구조체**/**class**/namespace **이름**일 때,
    S **정의** 안에서 **선언**한 **이름** data는 'S **소속 이름**'이 됩니다
    - '**선언**을 거기다 적는 것' 이외에 **소속** 관계를 구성하는 다른 방법은 일절 존재하지 않아요!

복습을 위한 슬라이드

- **소속** 관계
  - **이름**들 사이에 성립하는 관계입니다

  - S가 **구조체**/**class**/namespace **이름**일 때,
    S **정의** 안에서 **선언**한 **이름** data는 'S **소속 이름**'이 됩니다
    - '**선언**을 거기다 적는 것' 이외에 **소속** 관계를 구성하는 다른 방법은 일절 존재하지 않아요!

    - (안 중요)**함수 정의** 안에 **선언**을 적을 수도 있긴 하지만,
      이 경우는 '그 **함수** 소속'으로 치지는 않아요
      - 물론 그 **선언**으로 도입하는 **이름**은 local **이름**이며 전반부에서 본 그런 규칙들이 여전히 적용돼요

      - 직접 적어 본 적은 없지만, **함수 정의** 안에도 **구조체**/**class 정의**를 얼마든지 적을 수 있고,
        그 **구조체**/**class 소속 이름** (**멤버 이름**) 또한 얼마든지 **선언**할 수 있어요

복습을 위한 슬라이드

- **포함** 관계
  - **Object**들 사이에 성립하는 관계입니다
    - 어떤 **구조체/class 소속** nonstatic(**선언**에 static 안 붙인) **멤버** Data **이름**에 대한 **object**는, 해당 **구조체/class 형식 object**에 **포함**됩니다
      - 이렇게 포함되는 object들을 우리 수업에서는 '**멤버 object**'라 부를께요
      - 이 때 성립하는 **포함** 관계는 영속적입니다
        (**구조체/class 형식 object**가 **생성**될 때 **멤버 object**들 또한 **생성**되며,
        해당 **형식 object**가 **파괴**될 때 같이 **파괴**돼요)

복습을 위한 슬라이드

- **포함** 관계
  - **Object**들 사이에 성립하는 관계입니다
    - 어떤 **구조체/class 소속** nonstatic(**선언**에 static 안 붙인) **멤버** Data **이름**에 대한 **object**는, 해당 **구조체/class 형식 object**에 **포함**됩니다
    - **멤버** Code **이름**에 대한 'Code 덩어리'는 다른 **object**에 **포함**되지 않아요
    - (주의)Static **멤버** Data에 대한 **object**는 해당 **구조체/class 형식 object**에 포함되지 않아요!

복습을 위한 슬라이드

- **포함** 관계
  - **Object**들 사이에 성립하는 관계입니다
    - 어떤 **구조체/class 소속** nonstatic(**선언**에 static 안 붙인) **멤버** Data **이름**에 대한 **object**는, 해당 **구조체/class 형식 object**에 **포함**됩니다
    - **멤버** Code **이름**에 대한 'Code 덩어리'는 다른 **object**에 **포함**되지 않아요
    - (주의)Static **멤버** Data에 대한 **object**는 해당 **구조체/class 형식 object**에 포함되지 않아요!
    - (지금은 안 중요) 이러한 '물리적' **포함** 관계 외에도, 관점에 따라 '논리적' **포함** 관계를 인정할 수도 있어요
      - 이거는 소프트웨어공학 수업 등에서 등장할 것 같아요. 지금은 그러려니 합시다

복습을 위한 슬라이드

- Nonstatic / static **멤버** Data
  - C++에서 `static` specifier의 의미가 하나 더 늘었어요.
    멤버 Data **선언**에 붙이면 걔를 'static Data **멤버**'라 불러요
    - 뭐 우리 수업 흐름대로면 'static **멤버** Data'가 적당한 표현이긴 한데,
      자신의 언어 습관이나 주변 문맥에 맞는 것을 고르면 될 것 같아요
    - 아무튼, 안 붙인 쪽은 nonstatic Data **멤버**라 부를 수 있어요

복습을 위한 슬라이드

- Nonstatic / static **멤버** Data
  - C++에서 `static` specifier의 의미가 하나 더 늘었어요.
    멤버 Data **선언**에 붙이면 걔를 'static Data **멤버**'라 불러요
    - 뭐 우리 수업 흐름대로면 'static **멤버** Data'가 적당한 표현이긴 한데,
      자신의 언어 습관이나 주변 문맥에 맞는 것을 고르면 될 것 같아요

    - 아무튼, 안 붙인 쪽은 nonstatic Data **멤버**라 부를 수 있어요
      - (매우 주의)이 경우 해당 **멤버 object**의 **위치**는
        순전히 걔가 **포함**되는 **구조체/class object**의 **위치**에 의해 결정돼요!
        - » **선언** `static RN number;` 가 있을 때 `number.boonja`의 **위치**는 static **위치**가 맞아요

      - (매우 주의)그렇다 보니,
        멤버 Data **정의**를 **구조체/class 정의** 중괄호 밖에 적는 경우
        그 때는 `static` specifier를 붙이면 안 돼요!
        - » 붙이면 internal **이름**을 만들겠다는 뜻이 되는데, C++에서는 막아 놨어요

복습을 위한 슬라이드

- Nonstatic / static **멤버** Data
  - 방금 슬쩍 나왔듯 nonstatic Data **멤버**는 **포함** 관계를 구성해요
    - 따라서 사용할 때 반드시 '기준 **위치**'를 특정하는 **수식**과 함께 '. **수식**'을 적어 사용해야 해요
  - Static Data **멤버**는 **포함** 관계를 구성하지 않으나, 여전히 **소속** 관계는 구성해요
    - 이 경우 **object**를 사용할 때 그 **멤버**가 **소속**된 **구조체/class 형식 object**를 필요로 하지 않아요. 따라서 . **연산자** 대신 : : **연산자**를 사용할 수 있고, 그게 기본이에요

복습을 위한 슬라이드

- Nonstatic / static **멤버** Code
  - C++에서 `static` specifier의 의미가 또 하나 늘었어요.
    멤버 Code **선언**에 붙이면 걔를 'static Code **멤버**'라 불러요
    - 안 붙인 쪽은 nonstatic Code **멤버**라 부를 수 있어요

복습을 위한 슬라이드

- Nonstatic / static **멤버** Code
  - Nonstatic Code **멤버**는 전반부 마지막 즈음 나온 'Data와 Code의 연계'를 전제해요.
    따라서 호출할 때 . **수식**을 곁들인 '멤버 함수 호출식'을 적어야 해요

  - Static Code **멤버**는
    그 **멤버**가 **소속**된 **구조체/class 형식 object**와의 연계를 전제하지 않아요
    - 따라서 . **연산자** 대신 :: **연산자**를 사용할 수 있고, 그게 기본이에요

복습을 위한 슬라이드

- Nonstatic / static **멤버** Data / Code
  - 기존에 써 오던 specifier를 재탕한 것에 불과하지만…
    이 네 가지 케이스는 완전 따로 구경해 보면 좋을 것 같아요
    - 시험에 나오기 딱 좋아보여요
  - 강사도 꾸준히 구분해 가며 설명할 예정이니
    여러분도 마음 단단히 먹고 복습해 봐요

복습을 위한 슬라이드

- ODR(One Definition Rule)
  - **Lvalue**와 비슷한, 그럭저럭 널리 사용되는 숙어예요

  - '어떤 요소에 대한 **정의**는
    프로그램을 구성하는 전체 코드 내에
    (0개도, 2개 이상도 아닌) 단 한 개만 존재해야 한다'
    ...를 의미해요
    - 방금 예시에서 보았을 때 뭐 납득은 잘 되었을 거예요

복습을 위한 슬라이드

- ODR(One Definition Rule)
  - **Lvalue**와 비슷한, 그럭저럭 널리 사용되는 숙어예요

  - 이를 우회하기 위해 C++에서는 inline specifier를 붙일 수 있어요
    - '**위치 정의**를 수반할 수 있는 / 반드시 수반하는 **선언**'에 붙여야 의미를 가져요. 붙여 두면, 해당 **이름**에 대한 **정의**가 프로그램용 코드 내에 여럿 있어도 컴파일러가 그냥 그러려니 해요
      - ➢ 물론 요즘 C++ 컴파일러는 똑똑하므로 알아서 '**정의**를 한 곳에만 적은 것처럼' 컴파일해 줘요

    - 의미상 nonstatic Data **멤버 선언**에는 붙일 수 없어요(얘네에 대해서는 **offset 값**이 **정의**됨)
      - ➢ 옛날 C++에서는 static Data **멤버 선언**에도 못 붙였어요

    - 의미상 **구조체/class 선언/정의**에는 붙일 수 없어요

복습을 위한 슬라이드

- 컴파일러의 **lookup**
  - 컴파일러는 기본적으로,
    우리가 적은 **이름**을 보고 그 **이름**에 대한 **선언**을 찾을 때 context를 감안해요
  - 이러한 컴파일러의 선언 찾기 과정을 **lookup**이라 불러요
    - 오늘은 소개만 했어요. 나중에 다시 묶어볼께요
  - 아무튼, context상 컴파일러가 '내가 의도한 **선언**'을 찾기 어려운 경우,
    직접 : : **연산자**를 붙여 가며 **소속** 관계를 명시해줄 수 있어요
    - global 소속 **이름**을 의도하고 싶을 때는 좌항 없이 : : **연산자**를 적을 수도 있어요
    - 뭐 여기까지는 납득하기 어렵지 않을 거예요

마무리

- 여기까지, 이번 시간에는
  클래스를 좀 더 구체적으로 바라보기 위한 필수 요소들을 구경해 보았어요.
  - 뭐 절반 정도는 예전에 본 것들이고,
    나머지 대부분도 이전 수업내용의 조합으로 구성되어 있으니
    그럭저럭 납득하긴 어렵지 않을 거예요

  - 15일차에 기말고사 볼 예정이지만
    1일차의 나 자신이 그러했듯 미래의 걔에게 희망을 걸면 될 것 같아요

  - 잠시 쉬었다가, 이번에는 다시 한 번 프로그래머들 사이의 갈등에 주목해 봅시다