

## 9-2. Object 다시 보기

창의적소프트웨어프로그래밍  
2022년도 여름학기  
Racin

# 7일차 내용

- C와 다르게, C++에서는 Data **멤버** 뿐만 아니라 Code **멤버 선언** 또한 가능했어요.
  - 우리는 그 자리에서 중괄호 열고 바로 내용물이 뭔지 **정의**하긴 했어요
- 잘은 모르겠지만 **생성자, 파괴자 정의**도 구경해 보았어요.
  - 이걸 오늘 다시 정리해 볼게요
- 잘은 모르겠지만 operator 키워드도 몇 번 적어 보았어요.
  - 좌항에 `std::ostream`, 우항에 유리수가 있는 << **수식을 계산**하는 방법
    - 함수 정의를 '함수 호출식을 계산하는 방법'이라 볼 수 있으니 말 되는 것 같아요
  - 유리수 **object**를 검은 창 등으로 보내는 Data 흐름을 어떻게 해야 하는지
    - ?

# 7일차 내용

---

- 연산자 '함수'?

# 7일차 내용

---

- 연산자 '함수'?
  - 수업 초반에 **변수 선언**이 필요했던 기본적인 이유는,  
어떤 Data를 최종적으로 검은 창으로 보내기 위한 과정을  
여러 **문장의 실행**을 통해 구성해야 했기 때문이에요
    - 수식 하나로 깔끔하게 표현할 수 있다면 **변수 선언**을 굳이 안 해도 될 지 몰라요

# 7일차 내용

---

- 연산자 '함수'?

- 당장 지난 시간에 만든 유리수 덧셈만 하더라도  
꽤 많은 = 수식을 여러 문장에 걸쳐 적어야만 했어요
  - 사실 곱셈은 `return { boonja * rhs.boonja, boonmo * rhs.boonmo };` 한 번에 되긴 함

# 7일차 내용

- 연산자 '함수'?

- 당장 지난 시간에 만든 유리수 덧셈만 하더라도 꽤 많은 = 수식을 여러 문장에 걸쳐 적어야만 했어요
  - 사실 곱셈은 `return { boonja * rhs.boonja, boonmo * rhs.boonmo };` 한 번에 되긴 함
- 컴파일러는 `int` 덧셈, `double` 덧셈은 다룰 줄 알지만 유리수 덧셈은 모르기 때문에, 누가 그 방법을 '컴파일러가 아는 방법들만의 조합'으로 알려주어야 해요
  - 물론 당연히 유리수 개념을 창시하는 프로그래머가 미리 해 두면 편할 거예요
- 이 때는 '이건 알지? 이 수식들을 이 순서대로 계산해줘'와 같이 알려줘야 하고, 따라서 우리는 문장(들)을 적어 가며 `RN::Add()`, 또는, `RN::operator string()`에 대한 정확한 Code 실행 흐름을 정의하게 돼요
- 이런 측면을 놓고 생각하면 연산자를 Code 관점에서 바라봐도 어색하지 않을 거예요

# 7일차 내용

- 연산자 '함수'?

- 물론 C++에서는 여러분이 직접 유리수용 `operator +()`를 **정의**할 수도 있고, 해 두면 `main()` **정의** 적는 프로그래머가 더 직관적으로 유리수용 코드를 적을 수 있게 될 거예요
  - 이걸 재미있으니 내일로 미룰게요
  - 슬쩍 구경해 보고 싶다면,  
`std::string::operator +()` 등등이 **정의**되어 있으니 한 번 사용해 봐용

# 7일차 내용

- 연산자 '함수'?

- 물론 C++에서는 여러분이 직접 유리수용 operator +()를 **정의**할 수도 있고, 해 두면 main() **정의** 적는 프로그래머가 더 직관적으로 유리수용 코드를 적을 수 있게 될 거예요
  - 이걸 재미있으니 내일로 미룰게요
  - 슬쩍 구경해 보고 싶다면,  
std::string::operator +() 등등이 **정의**되어 있으니 한 번 사용해 봐용
- 아무튼 우리 수업에서는 **멤버 연산자**와 **멤버 함수**를 뭉뚱그려 Code **멤버**라 부를게요
  - 함수 호출식의 **함수 이름**을 적을 때 이 **이름** 자체는 Code 냄새가 잘 안 나듯(**위치 값** 나옴),  
  
<< **수식** 적을 때 적는 << **연산자** 자체는 Code 냄새가 안 나도  
그 **수식**의 **계산** 방법을 정하는 **연산자 정의**는 명백히 Code 덩어리가 맞으니 그럴싸 해요
    - 여기서 슬쩍 나온 << **연산자**처럼, 모든 **연산자**가 **멤버**인 것은 아니에용(재는 그냥 '친구'였어요)



# 오늘 내용

---

- C++의 **object**와 C++의 **클래스**에 대해 슬쩍 복습해 봅니다.
  - **클래스**는 갑자기 튀어나왔는데, 일단 여기서는 개념을 나타내는 명칭으로 쓰려 해요.  
아무튼 복습 맞음
- 새 **클래스 형식**에 대해 **정의** 가능한 특별한 **멤버 함수**들도 가볍게 정리해 봐요
  - **생성자** + 국선 **생성자**, 기본 **생성자**, 복사 **생성자**, 이동 **생성자**(마지막 둘은 오늘 새로 등장)
  - **파괴자** + 국선 **파괴자**
- 숨어 있던 키워드 `this`에 대해 슬쩍 구경만 해 봅니다

# 이번 시간에는

- **Object** 다시 보기
  - 새 구조체 형식에 대해 정의 가능한 특별한 **멤버 함수**들을 가볍게 정리해 봐요
    - 생성자 + 국선 생성자, 기본 생성자, 복사 생성자, 이동 생성자(마지막 둘은 오늘 새로 등장)
    - 파괴자 + 국선 파괴자
  - 각 **object**들의 **생성→파괴** 양상을 살짝 구경해 봐요
    - 전반적인 집계 결과를 구경하고,  
(디스어셈블리 아닌 일반) 디버그 모드를 사용해 볼 예정이에요

# Object 다시 보기

---

- 사람마다, 시기마다 단어 **object**에 대한 해석이 조금씩 달라요.
- 일단 우리는 두 가지 관점을 세워 접근해 보았어요:
  - C 당시의 것
  - C++의 것

# Object 다시 보기

- C 당시
  - 단어 **object**는 (메모리 위의) '칸'을 의미했어요
    - 원론적으로는 '이름'에 의해 특정되는 칸'을 의미하지만  
반자동 방식으로 잡은 칸 또한 (정당하게 사용한다면) **object**라 부르곤 했어요
    - 당시에 '배열 한 칸'이나 '능력치(구조체) 한 칸'과 같이  
'여러 **object**들을 포함하는 큰 **object**' 개념이 존재했어요
      - **Offset**을 다룰 때 종종 'int 기준 몇 칸'과 같은 표현을 들어 보았어요

# Object 다시 보기

- C 당시
  - 프로그램은 당연히 **값**을 얻기 위해 존재하며, **object**는 **값**을 담기 위한 도구로 쓰였어요.
    - '프로그래밍 언어'가 없던 시절에는  
메모리의 특정 위치에 대한 배타성을 프로그래머가 직접 도모해야 했어요  
(디스어셈블리 창에 보이는 Code 숫자들을 직접 적었다 생각해도 될 듯?)
      - 배타성 확보에는 기억력과 꼼꼼함이 필요해요 (까먹고 중복시키면 망함)

# Object 다시 보기

- C 당시

- 프로그램은 당연히 **값**을 얻기 위해 존재하며, **object**는 **값**을 담기 위한 도구로 쓰였어요.
  - 이걸 사람보다 컴퓨터가 더 잘 하는 종목에 속하므로  
'각 **이름**들에 대해 적절한(배타성을 도모 가능한) **위치 값**을 자동으로 정해주는 프로그램'  
...에 대한 수요가 생기는 것이 당연할 거예요
  - 'C 컴파일러' 또한 이러한 수요를 만족시키기 위한 프로그램들 중 하나라 말할 수 있어요.  
단어 'automatic **위치**'를 생각해 보면  
당시의 프로그래머들이 컴파일러에게 얼마나 '적은' 기대를 걸었는지 느낄 수 있을 거예요
    - 여전히 프로그래밍은 사람이 하는 것이며,  
컴파일러는 내가 하기 싫은 단순 노동을 대신 자동으로 해 주는 친구에 불과했어요

# Object 다시 보기

---

- C++ 당시
  - C++가 등장하는 시기 즈음 해서  
'Object = 값 + 칸' 메타를 주장하는 경우가 많아졌어요
    - '값'과 '그 값이 담긴 칸'을 세트로 바라봐요

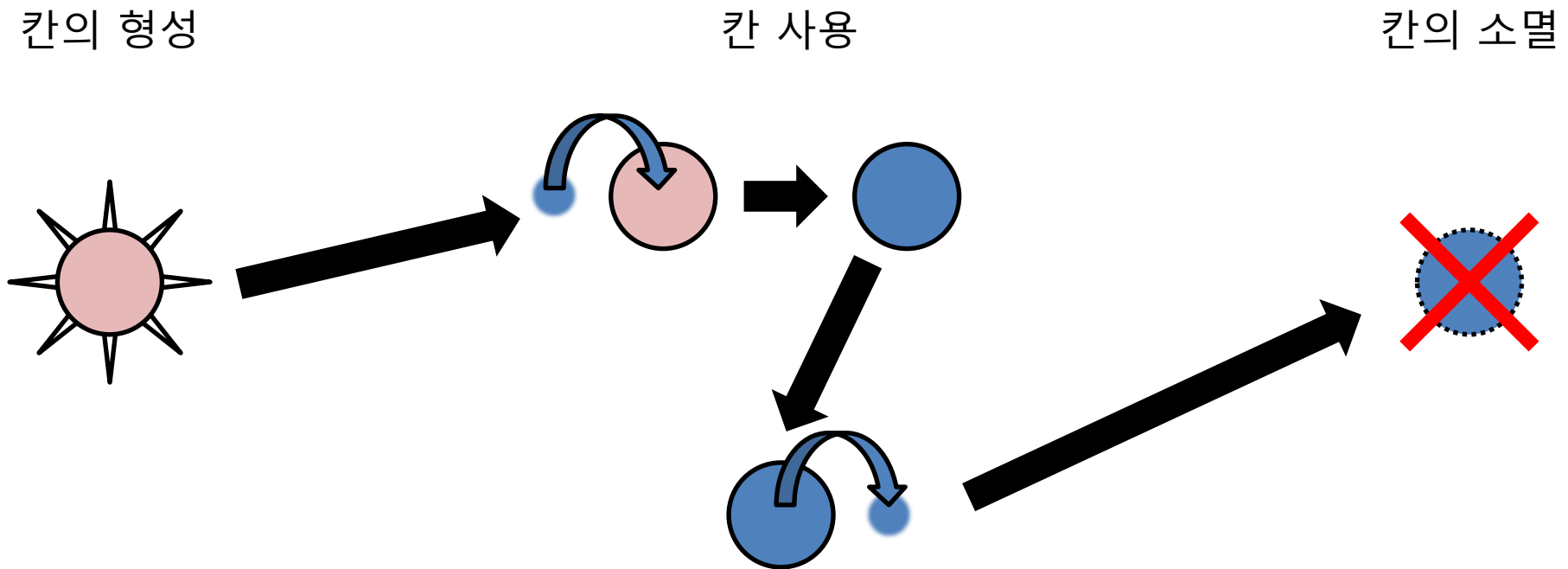
# Object 다시 보기

- C++ 당시
  - C++가 등장하는 시기 즈음 해서  
'Object = 값 + 칸' 메타를 주장하는 경우가 많아졌어요
    - '값'과 '그 값이 담긴 칸'을 세트로 바라봐요
  - ...이것만 놓고 보면 '그래서 뭐?' 하는 게 정상이에요.  
그래서 우리는 약간 속도를 늦추어서  
'칸권'에 대한 짤막한 상상을 해 가며 천천히 접근해 보고 있어요



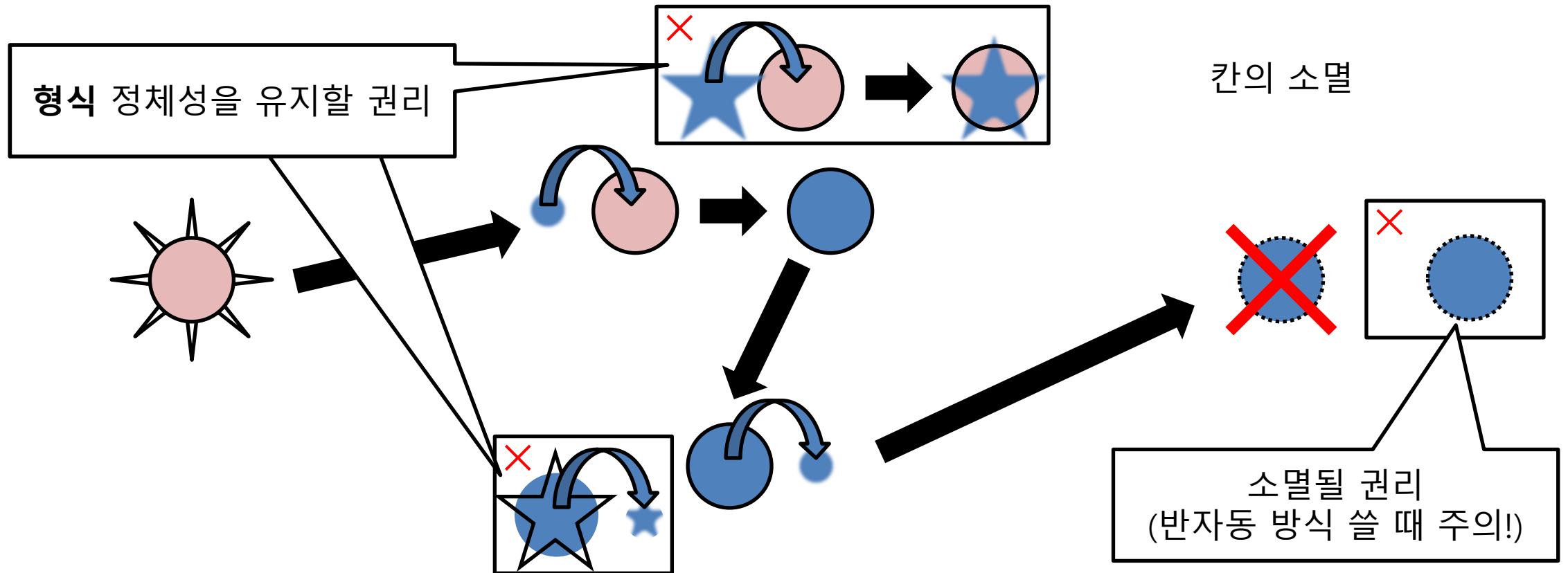
# Object 다시 보기

- C++ 당시
  - 대충 그 때 봤던 그림:



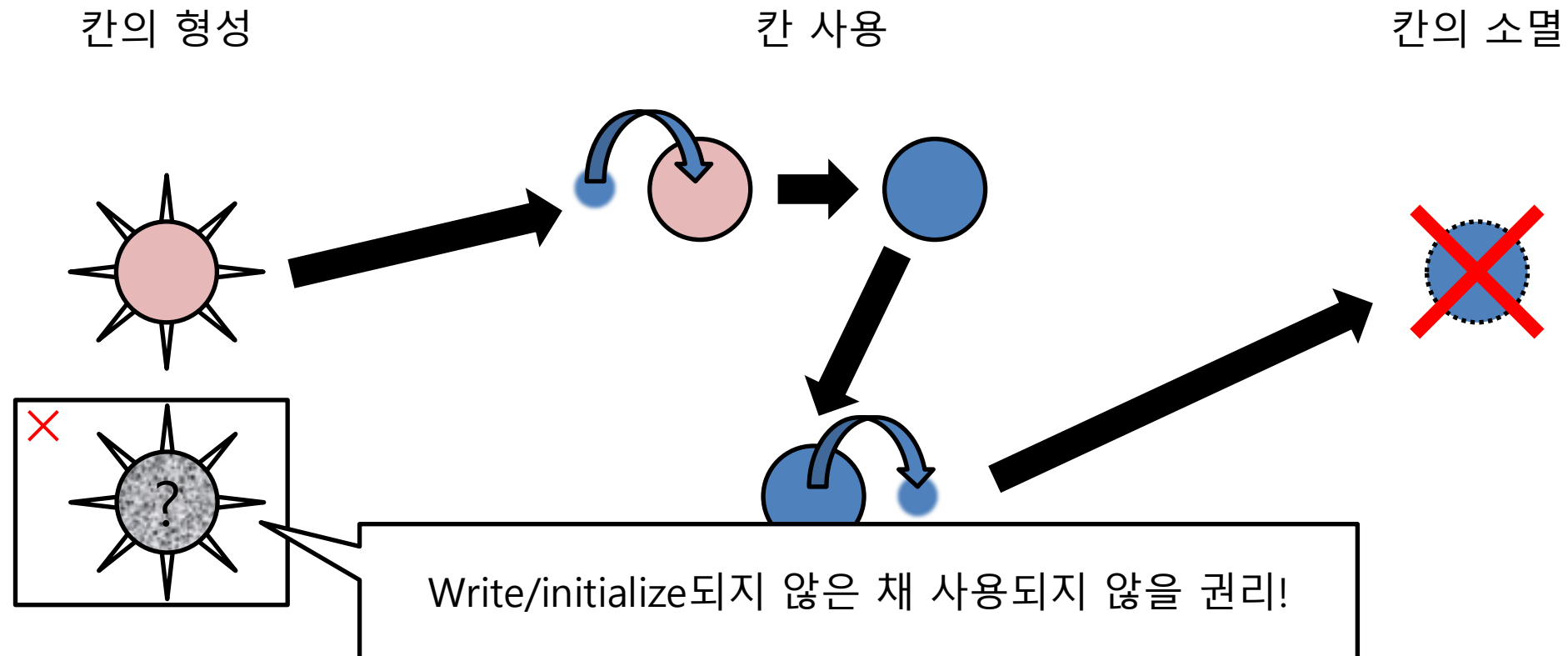
# Object 다시 보기

- C++ 당시
  - 칸권을 표현해 보면...



# Object 다시 보기

- C++ 당시
  - 여기에 '**Object** = 값 + 칸' 메타를 적용했을 때, 한 가지 권리가 더 부각돼요!



# Object 다시 보기

---

- 칸권 세 가지를 정리하면...
  - 소멸될 권리
  - **형식** 정체성을 유지할 권리
  - Write/initialize되지 않은 채 사용되지 않을 권리(추가됨)

# Object 다시 보기

- 칸권 세 가지를 정리하면...
  - 소멸될 권리
  - **형식** 정체성을 유지할 권리
  - Write/initialize되지 않은 채 사용되지 않을 권리(추가됨)
    - C에서는  
New\_Stat()과 같은 별도의 Code를 미리 적어 놓으면  
main() **정의** 적을 사람이 이에 대한 호출식을 적절한 자리에 적음으로써  
이 측면의 권리를 보호할 수 있었어요
    - C++에서는, '미래에 내 후배님이 까먹을 지 모르므로'  
새 **형식**을 **정의**하는 사람이 미리  
'새 **object**를 **생성**할 때마다 적절한 Code의 **실행**을 강제'하도록 지정할 수 있어요!
      - 이러한 목표를 달성하기 위해 우리는 **생성자**(constructor)를 적을 수 있어요!

# Object 다시 보기

---

- 코드를 보면서 가 봅시다.  
CSP\_9\_2\_yeshi\_1.cpp를 VS에 탑재해 주세요.
  - 7일차 최종 목표에 해당하는, 유리수 창시하는 내용이 적혀 있어요
- 강사와 함께 슬쩍 구경해 보고 옵시다

# Object 다시 보기

- 지금 이 코드의 `main()` **정의**에서는,  
lhs **object**에 대한 initializer로 (1/2)을 의도하고 있습니다.
  - 변수는 원래 값을 하나 담을 수 있는 친구였는데,  
이제는 값 + 칸 메타를 타고 있으니  
변수 lhs에 대해 정의된 위치 값으로 특정되는 칸과 거기 담긴 값을 세트로 묶어서  
'lhs **object**'라 불러도 될 것 같아요
    - 일단 유리수 형식 변수일 때는 그럭저럭 납득 가능할듯

# Object 다시 보기

- 지금 이 코드의 main() **정의**에서는,  
lhs **object**에 대한 initializer로 (1/2)을 의도하고 있습니다.
- Initializer를 안 적을 때는 어떻게 될까요?
  - main() **정의** 안 적당한 곳에 아래 코드를 적어 봅시다:

```
RationalNumber number ;
```

```
number .Add(number) ;
```



# Object 다시 보기

- 지난 시간에 살짝 구경했던 그 오류 메시지가 나옵니다.


```
int main()
{
    RationalNumber lhs{ 1, 2 };
    RationalNumber rhs{ 2, 3 };

    std::cout << lhs << " + " << rhs << " == " << lhs.Add(rhs) << std::endl
               << lhs << " - " << rhs << " == " << lhs.Subtract(rhs) << std::endl
               << lhs << " * " << rhs << " == " << lhs.Multiply(rhs) << std::endl
               << lhs << " / " << rhs << " == " << lhs.Divide(rhs) << std::endl;

    RationalNumber number;

    number.Add(number);

    return 0;
}
```

 (지역 변수) RationalNumber number  
온라인 검색  
"RationalNumber" 클래스의 기본 생성자가 없습니다.  
온라인 검색

# Object 다시 보기

- 지난 시간에 살짝 구경했던 그 오류 메시지가 나옵니다.

```
int main()
{
    RationalNumber lhs{ 1, 2 };
    RationalNumber rhs{ 2, 3 };

    std::cout << lhs << " + " << rhs << " == " << lhs.Add(rhs) << std::endl
    << lhs << " - " << rhs << " == " << lhs.Subtract(rhs) << std::endl
    << lhs << " * " << rhs <<
    << lhs << " / " << rhs <<

    RationalNumber number;

    number.Add(number);

    return 0;
}
```

생성자라는 무언가가 존재하고,  
이들 중 '기본 생성자'같은 것을  
개념적으로 구분하고 있는 것 같아요.

(지역 변수) RationalNumber number;  
온라인 검색  
"RationalNumber" 클래스의 기본 생성자가 없습니다.  
온라인 검색

# 생성자

- 지난 시간에 그 **정의**를 적어 본, **함수**같이 생겼는데 그 **이름**이 **구조체 이름**과 동일한 친구를 **생성자**라 불러요.
  - 우리 코드에는 20번째 줄 정도에 이렇게 적혀 있어요

```
/*
    생성자, 파괴자 정의 부분
*/

// 지정된 분자, 분모 값을 갖는 새로운 유리수 object를 initialize합니다.
RationalNumber(int boonja, int boonmo) : boonja{ boonja }, boonmo{ boonmo }
{
    // 기본적으로 유리수 object에는 '기약분수'를 담으려 하고 있어요
    Yakboon();
}
```

# 생성자

- 여기예다 아래와 같이 **생성자 정의**를 하나 더 추가해 봅시다:
  - 인수 없이 무조건  $(0/1) == 0$ 으로 initialize하도록 의도하고 있어요
    - 다 적었으면 `main()` **정의**에서 lhs, rhs **선언**의 initializer들을 지워버리고 한 번 실행해 보세요

```
/*  
    생성자, 파괴자 정의 부분  
*/
```

```
RationalNumber() : boonja{ 0 }, boonmo{ 1 } { }
```

```
// 지정된 분자, 분모 값을 갖는 새로운 유리수 object를 initialize합니다.  
RationalNumber(int boonja, int boonmo) : boonja{ boonja }, boonmo{ boonmo }  
{  
    ...  
}
```

# 생성자

- 오오... '인수 안 받는 생성자'를 추가하니 오류가 사라졌고, 실행해 보니  $0 + 0 = 0$ ,  $0 * 0 = 0$  해서 0이 잘 나오는 것 같습니다!
  - 인수 안 받는 생성자를 특별히 '기본 생성자'라고 부르나 봐요 (사실임)
    - 반면 원래부터 있던 생성자는 '(그냥) 생성자'예요

# 생성자

- 오오... '인수 안 받는 **생성자**'를 추가하니 오류가 사라졌고, 실행해 보니  $0 + 0 = 0$ ,  $0 * 0 = 0$  해서 0이 잘 나오는 것 같습니다!
  - 인수 안 받는 **생성자**를 특별히 '기본 **생성자**'라고 부르나 봐요 (사실임)
- (중요)아까는 '유리수 **형식** 만든 프로그래머'가 기본 **생성자 정의**를 적어 두지 않았기 때문에 `main()` **정의** 적는 사람이 initializer 없는 새 유리수 **변수 선언**을 적는 것이 차단되었어요!
  - 엄밀히는 '새 유리수 **object 생성**'을 의도했다가 차단되었어요
    - 처음에 뭐 담을지 유리수 창시자도 안 정해 놓았고 `main()` **정의** 적는 사람도 정하지 않는 경우 칸권이 보장되지 않을 수 있어요
  - '내가 만든 유리수 **형식 object**를 사용하려면 처음에 담을 분자/분모 **값**을 직접 정해오세요' ...와 같은 느낌으로 받아들일 수 있을 거예요
    - 칸권 보장을 위해 `main()` **정의** 적는 사람의 자유도를 깎았어요

# 생성자

---

- 이처럼 C++에서는, 어떤 **object**를 생성할 때 해당 **object**의 **형식**을 **정의**한 프로그래머가 **정의**해 둔 **생성자들** 중 하나를 호출하도록 구성되어 있어요.

# 생성자

- 이처럼 C++에서는, 어떤 **object**를 생성할 때 해당 **object**의 **형식**을 정의한 프로그래머가 정의해 둔 생성자들 중 하나를 호출하도록 구성되어 있어요.
  - 새 구조체 정의를 적는 프로그래머가 아무 생성자도 정의하지 않았다면 컴파일러가 자동으로 '국선' 기본 생성자를 제공하도록 설계되어 있어요
    - 생성자 정의를 하나라도 적어 두었다면 그 형식에 대한 국선 기본 생성자는 자동 제공되지 않아요
    - 국선 기본 생성자는 호출하면 (덜 중요한 그 예외를 제외한 형식) 멤버 **object**들에 대한 기본 생성자를 호출하며, 그 외에 다른 일은 안 해요
    - 기본 생성자 정의를 `RationalNumber() = default;` 와 같이 적을 수도 있어요. 이 경우 '기본 생성자는 국선 버전을 쓰겠어'와 같이 수동 지정한 셈이 돼요



# 파괴자

- 비슷한 느낌으로  
**파괴자**(destructor)도 여기서 가볍게 짚고 갈 수 있을 것 같아요.
  - 사용이 끝난 **object**에 대한 마지막 **값** 정리 작업을 할 수 있도록  
해당 **object**에 대한 칸이 소멸되기 전 시점에 호출돼요
  - **파괴자 정의**를 적을 때는 앞 부분을 ~RationalNumber()와 같이 적어요
    - 평소에는 이게 (**멤버 함수**의 일종인) **파괴자**의 **이름**이라고 생각해도 무방해요

# 파괴자

- 비슷한 느낌으로  
**파괴자**(destructor)도 여기서 가볍게 짚고 갈 수 있을 것 같아요.
  - **생성자 정의**와 다르게 **파괴자 정의**는 유효한 버전이 최대 하나만 존재할 수 있어요
- 역시나 국선 버전이 존재하고,  
우리가 직접 **정의**하지 않는다면 컴파일러가 국선 버전을 제공해 주고,  
원한다면 직접 '국선 버전 쓸게여' 하고 **정의**할 수도 있어요
  - `~RationalNumber() = default;` 하면 돼요

# 국선 버전 거부하기

- 지난 시간에 슬쩍 나왔듯,  
**파괴자 정의**를 `~RationalNumber() = delete;` 와 같이 적는 것도 가능해요.
  - 내가 **정의**하지 않으면 국선 버전이 적용될 상황에서 컴파일러에게 그 적용을 하지 말아달라는 의미가 돼요
    - 국선 버전을 제공하는 다른 요소(**생성자** 등)에 대해서도 동일한 느낌으로 적을 수 있어요
- 이 부분은 지금은 일단 그러려니 하고 넘어갑시다

# Object 다시 보기

---

- 대충 구경해 보았으니,  
이번에는 CSP\_9\_2\_yeshi\_2.cpp를 탑재해 열어 봅시다.
  - 방금 전 버전과 덧셈/뺄셈/곱셈/나눗셈 양상이 조금 달라요
  - 강사와 같이 확인해 봅시다

# Object 다시 보기

---

- 음... 방금 전과 동등한 프로그래밍 목표를 달성해 놓긴 했는데, `main()` **정의** 내용물이 아까보다 꽤 지저분합니다.

# Object 다시 보기

- 음... 방금 전과 동등한 프로그래밍 목표를 달성해 놓긴 했는데, `main()` **정의** 내용물이 아까보다 꽤 지저분합니다.
  - 이전 버전은 아무리 덧셈을 해도 좌항 **object**가 변경되지 않았지만, 이번 버전은 덧셈 결과가 좌항 **object**에 반영되므로 그 내용물이 변경될 수 있어요
  - 따라서, lhs **object**를 원본 그대로 살려 가며 활용하기 위해 매 번 새 **object**를 **생성**해 가며 덧셈 등의 좌항으로써 사용하고 있어요
  - 그렇다 보니 결과적으로 방금 전과 출력 결과는 동일해요

# Object 다시 보기

- 음... 방금 전과 동등한 프로그래밍 목표를 달성해 놓긴 했는데, `main()` **정의** 내용물이 아까보다 꽤 지저분합니다.
- 두 버전의 '성능'을  
'동일한 목표를 달성하던 도중 **생성-파괴**된 **object** 수'로 잰다고 한다면  
두 버전 중 어떤 것이 더 고성능이라 말할 수 있을까요?

# Object 다시 보기

- 음... 방금 전과 동등한 프로그래밍 목표를 달성해 놓긴 했는데, `main()` **정의** 내용물이 아까보다 꽤 지저분합니다.
- 두 버전의 '성능'을  
'동일한 목표를 달성하던 도중 **생성-파괴된 object** 수'로 잰다고 한다면  
두 버전 중 어떤 것이 더 고성능이라 말할 수 있을까요?
  - 확인해 보기 위해 `CSP_9_2_yeshi_3.cpp`를 탑재한 다음 구경→실행해 봅시다!



# Object 다시 보기

---

- 음? 이상합니다.
  - 의외로 두 번째 버전이 더 **object**친화적이라고 나오고 있어요

# Object 다시 보기

---

- 음? 이상합니다.
  - 의외로 두 번째 버전이 더 **object**친화적이라고 나오고 있어요
  - 다행히도 우리에게 VS의 최적화 기능이 있으니,  
Release 모드로 변경한 다음 다시 한 번 실행해 봅시다

# Object 다시 보기

---

- 음? 이상합니다.
  - 의외로 두 번째 버전이 더 **object**친화적이라고 나오고 있어요
  - 다행히도 우리에게 VS의 최적화 기능이 있으니,  
Release 모드로 변경한 다음 다시 한 번 실행해 봅시다
  - 최적화 결과 첫 버전의 살상 빈도가 약간 줄어들긴 했지만  
여전히 두 번째 버전이 더 친**object**적입니다

# Object 다시 보기

- 음? 이상합니다.
  - 사실 그것보다 더 이상한 점은,  
**생성** 횟수와 **파괴** 횟수가 매우 많이 차이난다는 점입니다
    - 칸권이 침해될 만한 부분은 없었던 것 같고,  
**생성자 정의** 안에는 **분기** 흐름이 전혀 없는데도  
**생성** 횟수가 제대로 합산되지 않고 있습니다
    - 이유가 뭘까요?

# Object 다시 보기

- 음? 이상합니다.
  - 사실 그것보다 더 이상한 점은,  
**생성** 횟수와 **파괴** 횟수가 매우 많이 차이난다는 점입니다
    - 칸권이 침해될 만한 부분은 없었던 것 같고,  
**생성자 정의** 안에는 **분기** 흐름이 전혀 없는데도  
**생성** 횟수가 제대로 합산되지 않고 있습니다
    - 맞아요. 툴팁에서 슬쩍 얼굴을 들이밀었던 또 다른 두 국선 **생성자**들이 있는데,  
context에 따라 우리가 정의해 둔 그냥 **생성자** 대신 그 **생성자**들 중 하나를 골랐을 수 있어요
      - 국선 버전을 썼든 아니든 **파괴자**는 하나뿐이므로 **파괴** 횟수는 제대로 카운트됨!

# Object 다시 보기

---

- 강사와 함께 CSP\_9\_2\_yeshi\_4.cpp를 구경해 보고 옵시다.

# 복사 생성자와 이동 생성자

- 방금 본 새로운 두 생성자들을 각각 **복사 생성자**, **이동 생성자**라 불러요.
  - **복사 생성자**(copy constructor)는 원본과 동일한 **값**을 갖도록 새 **object**를 initialize해요
    - 그 과정에서 원본 **object**는 변경되지 않아요
  - **이동 생성자**(move constructor)는 원본이 **소유**하는 **object**를 자신의 **object**로 삼아요

# 복사 생성자와 이동 생성자

- 방금 본 새로운 두 생성자들을 각각 **복사 생성자**, **이동 생성자**라 불러요.
  - **복사 생성자**(copy constructor)는 원본과 동일한 **값**을 갖도록 새 **object**를 initialize해요
    - 그 과정에서 원본 **object**는 변경되지 않아요
  - **이동 생성자**(move constructor)는 원본이 **소유**하는 **object**를 자신의 **object**로 삼아요
    - ?????????
    - **소유** 이야기는 내일 모아서 진행해 볼게요. 일단은 그러려니 합시다
    - 유리수 **형식**은 매우 간단하므로,  
예시에서는 bool **형식** Data **멤버**를 몰래 **선언**해 둔 다음 '**이동**당함' 표시용으로 사용했어요
      - 파괴자 정의에서 파괴 횟수를 카운트할 때  
**이동**당했는지(이미 빈터리인지) 여부에 따라 서로 다르게 카운트했고  
이를 통해 '뜯어온' 횟수와 '뜯긴' 횟수가 동일함을 관찰할 수 있었어요



# 마무리

- 좋아요.  
이 정도 보았으면 일단 **object**에 대한 둘러보기는 어느 정도 끝난 듯 해요.
  - 칸권 세 가지를 짚어 보았어요
    - 소멸될 권리
    - **형식** 정체성을 유지할 권리
    - Write/initialize되지 않은 채 사용되지 않을 권리
  - 각종 **생성자** / **파괴자** 친구들을 구경해 보았어요
  - 잠시 쉬었다가, 이 다음에는 **클래스**에 대한 이야기를 이어서 진행해 봅시다