7-1. C++의 형식

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

5, 6일차 내용

- 두 가지 측면에서 의미 확장을 시도해 보았어요:
  - C에서의 칸 → **object**: **값** + 칸
    - 칸의 권리
      - 사용되기 전에 유효한 **값**을 지닐 권리
      - **형식** 정체성을 유지할 권리
      - 적절한 시점에 정당하게 소멸당할 권리
  - C의 **구조체**는 Data들을 모아 담기 위해 쓰임 → Data & Code
    - Data와 그걸 다루기 위한 Code를 연계
    - Code와 그 **실행**에 필요한 Data를 연계
  - 그 과정에서
    offset이니 deduction이니 반자동이니 다양한 것들이 튀어나왔지만
    뭐 일단 시험 공부 하는 셈 치고 다시 보면 대충 가져갈 수 있을 것 같아요

오늘 내용

- C++에서 새 **형식 이름**을 정하고 그 내용을 **정의**해 봅니다.
  - (C++ 스타일) **구조체 정의**부터 시작할 예정이에요. 여기에...
    - 해당 **구조체 형식 object**와 연계해 사용하기 위한 Code를 **정의**해 봐요
    - '칸권'을 도모하기 위한 Code 요소들을 도입해 봐요
    - Context에 따라 컴파일러가 고를 수 있는, 간단한 새 deduction 방안을 도입해 봐요
  - 중간에 새 키워드 class를 사용해 봐요
  - 오늘은 일단 가볍게 둘러보고, 중간고사 끝난 다음 다시 풀어볼께요

이번 시간에는

- 유리수를 다루기 위한 새로운 **형식**인 `struct RationalNumber`를 **정의**해 봅니다.
  - (일단 만들어 둔 다음, 앞으로 새로 등장하는 개념들을 조금씩 덧대 볼 예정이에요)
  - 현실의 유리수 값 하나를 다루기 위한 Data들을 담을 **멤버 선언**
  - 유리수 **object**를 출력하기 위한 Code **정의**

이번 시간에는

- 시작해 봅시다.
  - 오늘은 평소와 다르게 .cpp 파일을 준비해 주세요
    - 오늘 최종 목표, 며칠 후 수업에서도 계속 이 파일을 다룰 예정이니
      아예 새 프로젝트 만들고 그 안에서 .cpp 파일을 준비해 두면 좋을 것 같아요
      - 동봉된 main.cpp를 그대로 사용해도 좋아요

복습 겸 실습

- 아래 목표에 맞는 코드를 적어 봅시다:
  - std::cout 등을 사용하기 위한 헤더 파일 iostream include하기
  - 새 **구조체 형식** struct RationalNumber에 대한 **구조체 정의** 적기
    - **멤버 선언**으로는 int boonja; 와 int boonmo; 를 적어 주세요
  - main() **정의** 적기
    - 유리수 **형식 변수** n **선언**
    - n **object**에 '3 / 5' 담기
    - n **object**의 내용을 std::cout으로 출력

복습 겸 실습

- 출력 부분을 제외하면 아직은 C랑 다를 바 없는 것 같아요.
  - 사실 C++에서도 여전히 printf()를 사용할 수 있기는 해요

복습 겸 실습

- 출력 부분을 제외하면 아직은 C랑 다를 바 없는 것 같아요.
  - 사실 C++에서도 여전히 printf()를 사용할 수 있기는 해요

- 여기서 목표를 조금 고쳐서,
  n에 대한 칸을 형성한 '다음' 분자, 분모 **값**을 담는 것이 아닌,
  n에 대한 칸을 형성하는 그 시점에 **값**을 담도록(initialize하도록)
  만들어 봅시다.
  - 이전에 C 다룰 때 능력치 **형식 변수 선언**의 initializer를 적어 본 적이 있으니,
    그 때의 기억을 토대로 한 번 시도해 봐요

  - 이미 그렇게 해 놨다면 잠시 쉬어도 좋아요

복습 겸 실습

- 답을 확인해 봅시다:

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

여기 있는 C++ **선언**들은
(주석으로 막은 부분을 제외하면)
모두 3/5로 예쁘게 initialize하는 동등한 **선언**이에요!

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    struct RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

C++에서는 이렇게 '**구조체 이름**'을 적을 수도 있지만,
struct 키워드를 안 붙인 채
그냥 **형식 이름**으로써 적을 수도 있고,
꼭 필요하지 않다면 안 붙이는 편이에요.
(자주 등장할 예정이니 이제부터는 **typename**이라 지칭)

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

얘가 유효하지 않은 이유는...

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

(안 중요)얘가 유효하지 않은 이유는...
이 (3, 5)가 int **형식 수식**으로 간주되기 때문이에요.
, **연산자**로 묶은 셈이고,
계산하면 맨 뒤 **수식**의 **계산** 결과**값**인 5가 나와요.

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

반면 이 경우는 인정이에요.

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

반면 이 경우는 인정이에요.
컴파일러는 코드를 (우리처럼) 좌→우 방향으로 읽는데,
typename을 먼저 적어 둠으로써
뒤이어 등장한 ( ) 표현이
새 **object**의 내용을 정한 셈이라 인식할 수 있어요.

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNu
    RationalNumber n6 = RationalNu

    // ...
}
```

반면 이 경우는 인정이에요.
컴파일러는 코드를 (우리처럼) 좌→우 방향으로 읽는데,
typename을 먼저 적어 둠으로써
뒤이어 등장한 ( ) 표현이
새 **object**의 내용을 정한 셈이라 인식할 수 있어요.

우리가 직접 내용을 정하는 것은 **정의**라 부를 수 있고,
함수 정의나 **구조체 정의**를 생각하면,
새 **object**의 내용을 정하는 것 또한
중괄호를 써서 표현해도 그리 나쁘지 않을 것 같아요.

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

다시 이 표현을 구경해 보면,
얘는 마치 **함수** 호출식 같이 생긴 것 같아요.
함수 호출식은
미리 정해 둔 Code 덩어리를 **실행**하고 싶을 때
적을 수 있는 **수식**이지요.

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

궁극적으로
여기 적힌 내용에 입각해서 = **수식**을 **계산**한다는 점
...을 생각하면, 이 표현도 꽤 괜찮은 것 같아요.

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2{ 3, 5 };
    //RationalNumber n3 = (3, 5);
    RationalNumber n4 = { 3, 5 };
    RationalNumber n5 = RationalNumber(3, 5);
    RationalNumber n6 = RationalNumber{ 3, 5 };

    // ...
}
```

C++에서는 이렇게 표현하는 것도 가능해요.
n1 **선언**은 조금 요상하게 느껴지긴 하지만...

복습 겸 실습

- 답을 확인해 봅시다:

```
int main()
{
    RationalNumber n1(3, 5);
    RationalNumber n2
    {
        3,
        5,
    };

    // ...
}
```

C++에서는 이렇게 표현하는 것도 가능해요.
n1 **선언**은 조금 요상하게 느껴지긴 하지만...
n2 **선언**은 엔터 좀 치고 , 하나 더 붙이면
정의같은 그럴싸한 모양이 나오고 있지요?

새 object

- 우리 수업에서는, 새 **object**의 내용을 정하기 위해 무언가를 적을 때는 기본적으로 { }를 써서 표현해 볼게요.
  - 옛날 C++에서는 불가능했어요
    → 옛날 C++를 다루는 다른 수업에서 나온 코드와 구분하기 쉬울듯
  - 사실 VS도 지금은 기본적으로 { }를 적었다 치는 편이에요
    - 이건 강사와 직접 확인해 봅시다

새 object의 권리

- 아무튼, 우리는 이렇게 칸권 중 하나인
  '사용되기 전에 유효한 **값**을 지닐 권리'를 지켜줄 수 있었어요.
  - 우리는 기본적으로 **문장** 단위로 Code 흐름을 구성하고 있고,
    이전 버전은 n **선언**과 = **수식**(담긴 **문장**) 사이에 다른 **수식/문장**을 적을 수 있던 반면,
    현재 버전은 n **선언**의 declarator와 initializer 사이에 다른 **수식/문장**을 적을 수 없어요

  - 컴파일러는 우리가 직접 시키지 않는 한 칸권을 침해하지 않으니,
    프로그래머가 칸 형성 시점과 값 담는 시점 사이에 다른 코드를 못 적게 막는 것 만으로
    칸권을 충분히 지킬 수 있을 것 같아요

유리수 object 출력하기

- 이번엔 출력 부분을 봅시다. 여러분도 아마 이런 식으로 적어 놨을 거예요:

```
int main()
{
    // ...

    std::cout << '(' << n.boonja << ',' << n.boonmo << ')' << std::endl;

    // ...
}
```

유리수 object 출력하기

- 이번엔 출력 부분을 봅시다. 여러분도 아마 이런 식으로 적어 놨을 거예요:

```
int main()
{
    // ...
    // std::cout << n << std::endl;
    std::cout << '(' << n.boonja << ',' << n.boonmo << ')' << std::endl;

    // ...
}
```

아쉽게도 이렇게 적으면 바로 빨간 줄 떴을 거예요.

그 이유를 설명한다면 뭐라고 말할 수 있을까요?

유리수 object 출력하기

- 이번엔 출력 부분을 봅시다. 여러분도 아마 이런 식으로 적어 놨을 거예요:

```
int main()
{
    // ...
    // std::cout << n << std::endl;
    std::cout << '(' << n.boonja << ',' << n.boonmo << std::endl;

    // ...
}
```

맞아요. 가장 간단하게 설명한다면,
'컴파일러가 몰라서'예요.

그러면, **수식** std::cout << n.boonja가 가능한 이유는
뭐라고 말할 수 있을까요?

수식 std::cout << '('를 **계산**한 결과**값**은
수식 std::cout 으로 특정되는 그 **object**예요.

일단 지금은 'cout 나온다' 정도로 생각해 둡시다.

유리수 object 출력하기

- 이번엔 출력 부분을 봅시다. 여러분도 아마 이런 식으로 적어 놨을 거예요:

'컴파일러가 알아서'라 답하면 반답이고,
'std::cout 만드신 분(또는 그 선배님)이 미리 컴파일러에게 알려줘서'
...라 말하면 적당할 것 같아요!

아무튼 main() **정의** 적는 지금 시점보다는 과거에 이미 **정의**해 놔서 그래요.

```
int m
{
    // ...

    std::cout << '(' << n.boonja << ',' << n.boonmo << ')' << std::endl;

    // ...
}
```

유리수 object 출력하기

- 방금 전에 문제가 생긴 **수식**을 들고 와 봤어요:

```
std::cout << n
```

  - 이 **수식**에는 자그마치 세 가지 요소가 한 데 뭉쳐 있어요:
    - 좌항
    - **연산자**
    - 우항

유리수 object 출력하기

- 방금 전에 문제가 생긴 **수식**을 들고 와 봤어요:

```
std::cout << n
```

  - 따라서, 이 **수식**을 유효하게 만드는 방법도 크게 세 가지로 볼 수 있어요:
    - 좌항을 건드린다: `std::cout` Mk. 2를 만들자
    - **연산자**: 아니다. 딱 이럴 때 사용할 만한 Code를 하나 더 만들자
    - 우항: 유리수 **object**에 대한, `std::cout`과 호환되는 `std::string` **object**를 얻는 Code를 만들어 두고 deduction 해 주기를 바라자

유리수 object 출력하기

- 방금 전에 문제가 생긴 **수식**을 들고 와 봤어요:

```
std::cout << n
```

  - 따라서, 이 **수식**을 유효하게 만드는 방법도 크게 세 가지로 볼 수 있어요:
    - ~~좌항을 건드린다: std::cout Mk. 2를 만들자~~ → 이건 어려움
    - **연산자**: 아니다. 딱 이럴 때 사용할 만한 Code를 하나 더 만들자
    - 우항: 유리수 **object**에 대한, std::cout과 호환되는 std::string **object**를 얻는 Code를 만들어 두고 deduction 해 주기를 바라자
    - 잠시 강사와 함께 이 두 가지 방법을 시도해 볼께요 (오늘은 구경만 하고 설명은 나중에)

유리수 object 출력하기

- 갑자기 `friend`, `operator`같은 키워드, **선언**에 & 붙인 표현 등이 등장했어요.
  - 일단 새 << **연산자** 구성하는 부분은 당분간은 복붙해서 써요

  - `operator std::string() { ... }`은 새로운 deduction 방안을 추가한 셈이에요
    - '새 casting **연산자**'를 도입한 셈이에요

유리수 object 출력하기

- 갑자기 `friend`, `operator`같은 키워드, **선언**에 & 붙인 표현 등이 등장했어요.
  - 일단 새 << **연산자** 구성하는 부분은 당분간은 복붙해서 써요
  - `operator std::string() { ... }`은 새로운 deduction 방안을 추가한 셈이에요
    - '새 casting **연산자**'를 도입한 셈이에요
  - 아무튼 우리는 두 방식을 통해,
    형식 문제로 Data 흐름이 끊기지 않도록 적절한 Code를 도입할 수 있었어요

유리수 object 출력하기

- 갑자기 `friend`, `operator`같은 키워드, **선언**에 & 붙인 표현 등이 등장했어요.
  - 일단 새 << **연산자** 구성하는 부분은 당분간은 복붙해서 써요

  - `operator std::string() { ... }`은 새로운 deduction 방안을 추가한 셈이에요
    - '새 casting **연산자**'를 도입한 셈이에요

  - 아무튼 우리는 두 방식을 통해,
    형식 문제로 Data 흐름이 끊기지 않도록 적절한 Code를 도입할 수 있었어요
    - 둘 다 첫 줄이 특이하게 생기긴 했지만,
      아무튼 { }를 붙였고, 그 안에 목표를 달성하기 위한 문장들을 적어 보았어요
      - 달리 말하면 얘네들도 전부 **함수**라 말할 수 있어요!

유리수 object 출력하기

- 갑자기 friend, operator같은 키워드, **선언**에 & 붙인 표현 등이 등장했어요.
  - 일단 새 << **연산자** 구성하는 부분은 당분간은 복붙해서 써요

  - operator std::string()
    - '새 casting **연산자**'를 도

  - 아무튼 우리는 두 방식을
    형식 문제로 Data 흐름이
    - 둘 다 첫 줄이 특이하게
      아무튼 { }를 붙였고, 그 안에 목표
      - 달리 말하면 얘네들도 전부 **함수**라 말할 수 있어요!

'친구'인 operator <<는 아니었지만, operator std::string()의 경우
어엿한 struct RationalNumber의 **멤버**라 할 수 있어요.
실제로 그 **함수 정의** 첫 줄에 마우스 포인터 갖다 대면
RationalNumber::operator std::string()
...이라 적혀 있는 것을 볼 수 있을 거예요!

C++의 형식

- 오… C++에서는
  어떤 새 **형식**에 대한 Code **멤버**도 **선언/정의**할 수 있는 것 같아요.
  - 해당 **형식 object**를 다룰 때 곁들여 사용할 수 있어요

  - 심지어 어떤 Code **멤버**는 컴파일러가 필요에 따라 골라 사용하는 것 같아요
    - 어떤 **수식**을 컴파일할 때
      Code **멤버 정의** 내용물이 **실행**될 수 있도록 적당한 명령어를 추가해 줄 수 있어요

C++의 형식

- 오... C++에서는
  어떤 새 **형식**에 대한 Code **멤버**도 **선언/정의**할 수 있는 것 같아요.
  - 해당 **형식 object**를 다룰 때 곁들여 사용할 수 있어요
  - 심지어 어떤 Code **멤버**는 컴파일러가 필요에 따라 골라 사용하는 것 같아요
- 이렇게 놓고 보니, boonja랑 boonmo는 Data **멤버**라 불러도 좋을 것 같아요.

C++의 형식

- 맞아요. 옛날 C랑 다르게, 우리는 새 C++ **형식**을 **정의**할 때 Data **멤버**(**변수**, **배열** 등)와 Code **멤버**(**함수**, **연산자**?)를 둘 다 추가할 수 있어요.
  - 다음 시간에 조금 더 다양한 Code **멤버**들을 추가해 보기로 하고...

마무리

- 맞아요. 옛날 C랑 다르게, 우리는 새 C++ **형식**을 **정의**할 때 Data **멤버**(**변수**, **배열** 등)와 Code **멤버**(**함수**, **연산자**?)를 둘 다 추가할 수 있어요.
  - 다음 시간에 조금 더 다양한 Code **멤버**들을 추가해 보기로 하고...

지금은 일단 잠시 쉽시다