

9-3. 클래스 다시 보기

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

이번 시간에는

- **클래스**에 대해 슬쩍 복습해 봅니다.
 - **클래스**는 갑자기 튀어나왔는데, 일단 여기서는 개념을 나타내는 명칭으로 쓰려 해요.
아무튼 복습 맞음
 - 숨어 있던 키워드 `this`에 대해 슬쩍 구경만 해 봅니다

클래스 다시 보기

- 단어 **object**가 **runtime**을 전제한다면,
단어 **클래스**는 그 이전 시점(코드 적는 시점 + compile time)이 배경이에요.
 - 코드 적는 시점에 고려 가능한 두 가지 핵심 관점인 Data와 Code를 엮어서
'Data & Code'라는 제목으로 구경해 보고 왔어요
 - 그 때는 **구조체 정의**를 다루었는데...

이거는 지금 당장 강사와 슬쩍 VS에 다녀와 볼게요

클래스 다시 보기

- 단어 **object**가 **runtime**을 전제한다면,
단어 **클래스**는 그 이전 시점(코드 적는 시점 + compile time)이 배경이에요.
 - 코드 적는 시점에 고려 가능한 두 가지 핵심 관점인 Data와 Code를 엮어서
'Data & Code'라는 제목으로 구경해 보고 왔어요
 - 그 때는 **구조체 정의**를 다루었는데...
구조체 정의 또한 **클래스** 개념을 도입하는 방법 중 하나라 보면 될 것 같아요
 - 슬쩍 나온 `public`은 다다음 시간에 구경해 볼게요
 - 우리 수업에서는...
한국어 단어 **클래스**는 개념,
구조체와 영단어 **class**는 **형식**,
구체적인 **형식 이름**을 적을 때 꼭 구분해야 한다면 `struct RN`이나 `class C` 등으로 표현할게요
 - 그런데 아마 귀찮아서 여간해서는 그냥 **구조체 정의** 위주로 적을 것 같아요

클래스 다시 보기

- 아래 C 코드를 봅시다:
 - 수식 `games[0]->Run`과 수식 `Press_3_to_Win`은 **계산 결과값**이 같아요

```
games[0]->Run(games[0]);
```

```
games[0]->Run(games[1]);
```

```
Press_3_to_Win(games[0]);
```

```
Press_3_to_Win(games[1]);
```

클래스 다시 보기

- 아래 C 코드를 봅시다:
 - 당시에, 수식 `games[0]->Run`과 수식 `Press_3_to_Win`은 **계산 결과값**이 같았어요

```
games[0]->Run(games[0]);
```

(정상) 0번 게임용 Code를 0번 게임용 Data와 연계

```
games[0]->Run(games[1]);
```

```
Press_3_to_Win(games[0]);
```

(결과적으로 정상)

```
Press_3_to_Win(games[1]);
```

클래스 다시 보기

- 아래 C 코드를 봅시다:
 - 당시에, 수식 `games[0]->Run`과 수식 `Press_3_to_Win`은 **계산 결과값**이 같았어요

```
games[0]->Run(games[0]);
```

(정상) 0번 게임용 Code를 0번 게임용 Data와 연계

```
games[0]->Run(games[1]);
```

(이상) 0번 게임용 Code를 1번 게임용 Data와 연계

```
Press_3_to_Win(games[0]);
```

(결과적으로 정상)

```
Press_3_to_Win(games[1]);
```

(이상)

클래스 다시 보기

- struct Game을 **정의**한 사람, Press_3_to_Win 게임을 만든 사람은 미래에 main() **정의** 적는 사람이 '자신들이 의도한 방식'으로 **함수** 호출식을 구성할 것이라 상정해요.
 - 만약 games[0] = New_Press_3_to_Win() 했다면
games[0]->Initialize(games[0]),
games[0]->Run(games[0]),
games[0]->Finalize(games[0]),
Delete_Press_3_to_Win() ...이 순서대로 **계산**되도록 틀림 없이 적어 둘 것이라 상정해요
 - 이렇게 적었을 때만 Data와 Code가 잘 연계되도록 설계해 놓았어요

클래스 다시 보기

- struct Game을 **정의**한 사람, Press_3_to_Win 게임을 만든 사람은 미래에 main() **정의** 적는 사람이 '자신들이 의도한 방식'으로 **함수** 호출식을 구성할 것이라 상정해요.
 - 만약 games[0] = New_Press_3_to_Win() 했다면
games[0]->Initialize(games[0]),
games[0]->Run(games[0]),
games[0]->Finalize(games[0]),
Delete_Press_3_to_Win() ...이 순서대로 **계산**되도록 틀림 없이 적어 둘 것이라 상정해요
 - 이렇게 적었을 때만 Data와 Code가 잘 연계되도록 설계해 놓았어요
- 하지만 이러한 믿음은 종종 배신으로 이어졌고,
나 대신 컴파일러가 main() **정의** 적는 사람의 실수를 차단할 수 있는 새로운 프로그래밍 언어에 대한 수요 또한 그만큼 높아지게 되었어요.

클래스 다시 보기

- 그래서 C++에서는...
 - 정상적인 수식은 더 짧게 적어도 되도록 하고, 그렇지 않은 수식은 못 적게 막아 놔어요!

```
games[0]->Run(games[0]);
```

```
games[0]->Run(games[1]);
```

```
Press_3_to_Win(games[0]);
```

```
Press_3_to_Win(games[1]);
```



```
games[0]->Run();
```

컴파일러가
대신 적어 줌!

나머지 시도는
(결과적으로 정상일 수 있더라도)
전부 막혀 있음!

클래스 다시 보기

- 그래서 C++에서는...
 - 정상적인 수식은 더 짧게 적어도 되도록 하고, 그렇지 않은 수식은 못 적게 막아 뒀어요!

```
games[0]->Run(games[0]);
```



```
games[0]->Run();
```

이렇게 컴파일러가 대신 적어 둔 인수가,
첫 페이지에서 살짝 언급한 `this`라는 친구예요!
앞으로도 다양한 **구조체/class 정의**들을 만들어 볼 예정이니
일단은 그냥 그러려니 하고 넘어가도 좋을 듯 해요.

클래스 다시 보기

- 그래서 C++에서는...
 - 정상적인 수식은 더 짧게 적어도 되도록 하고, 그렇지 않은 수식은 못 적게 막아 뒀어요!

games[0]->Run(games[0]);



games[0]->Run();

Press_3_to_Win(games[0]);



Press_3_to_Win();

C 시절에는 결과적으로 정상이었다 하더라도
C++에서는 '어떤 **object** 내 Data와 연계하여 동작하도록 지정된 Code',
다시 말하면 어떤 **클래스**의 '**멤버 함수**'들은
이런 식으로 '**object** 안 거치고 직접 호출'하는 것이 막혀 있어요!
(이 부분은 다다음 시간에 '**제한**' 이야기를 하면서 더 명확하게 정리해 볼게요)

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber other)
    {
        boonja *= other.boonja;
        boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.Multiply(n2);

    return 0;
}
```

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

// 미리 적어 둔 유리수 class 정의

```
class RationalNumber
```

```
{
```

```
public:
```

```
    int boonja;
```

```
    int boonmo;
```

```
    void Multiply(RationalNumber other)
```

```
{
```

```
        boonja *= other.boonja;
```

```
        boonmo *= other.boonmo;
```

```
}
```

```
};
```

이 Multiply()는 유리수 **class 정의 '안'에 선언 및 정의**되어 있어요.

// 나중에 적게 될 main() 예시

```
int main()
```

```
{
```

```
    RationalNumber n1, n2;
```

```
    n1.Multiply(n2);
```

```
    return 0;
```

```
}
```

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

// 미리 적어 둔 유리수 class 정의

```
class RationalNumber
```

```
{
```

```
public:
```

```
    int boonja;
```

```
    int boonmo;
```

```
    void Multiply(RationalNumber other)
```

```
{
```

```
        boonja *= other.boonja;
```

```
        boonmo
```

```
}
```

```
};
```

// 나중에 적게 될 main() 예시

```
int main()
```

```
    n2;
```

```
    r1.Multiply(n2);
```

이 Multiply()는 유리수 **class** 정의 '안'에 선언 및 정의되어 있어요.
따라서 애는 유리수 **class**의 '멤버 함수'예요.

예시 코드의 이름 Multiply에 마우스를 갖다 대면
RationalNumber::Multiply라 뜨는 것을 볼 수 있어요.
std::string 등에 빗대어 생각해 본다면 이게 일종의 '풀 네임'이라 볼 수 있을 듯 해요!

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber other)
    {
        boonja *= other.boonja;
        boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.Multiply(n2);
}
```

Multiply()를 호출하려면 이런 식으로 '**멤버 함수 호출식**'을 적어야 해요.
· **연산자**(또는 -> **연산자**)와 조합하여 좌항을 구성해야 해요!

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber other)
    {
        boonja *= other.boonja;
        boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.Multiply(n2);
    //Multiply(n2);
    //Multiply(n1, n2);
}
```

이런 용법은 허용되지 않아요!
main()같은 일반적인 **함수**와 Multiply()같은 **멤버 함수**는
둘 다 **함수**긴 하지만 사용 방법이 약간 달라요.

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber other)
    {
        boonja *= other.boonja;
        boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.boonja = 3;
    //boonja = 3;
```

사실 이 규칙은 '멤버 변수'에 대해서도 유사하게 적용된다 볼 수 있어요.
예시 코드의 boonja에 마우스를 갖다 대면
애 역시 RationalNumber::boonja와 같은 풀 네임을 가지고 있어요!

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber other)
    {
        boonja *= other.boonja;
        boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.Multiply(n2);
}
```

다시 **멤버 함수** 이야기로 돌아와서,
이번엔 **함수 정의** 내용물을 잠시 살펴 봅시다.

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber other)
    {
        boonja *= other.boonja;
        boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.Multiply(n2);
}
```

어... 방금 말한 것과 다르게
여기서는 **멤버 이름**을 그냥 적고 있어요.

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

사실 이 수식은 이렇게 간주돼요!

컴파일러가 몰래 만들어 둔, **계산**하면 '0번째 인수' 값 나오는 특별한 **이름** this와
-> **연산자**를 조합하여

'**멤버 함수** 호출식의 . 연산자 좌항 수식으로 특정되는 **object**'의 분자/분모 **object**를 사용해요.

```
//  
class RationalNumber  
{  
public:  
    int boonja,  
    int boonmo;  
  
    void Multiply(RationalNumber other)  
    {  
        this->boonja *= other.boonja;  
        this->boonmo *= other.boonmo;  
    }  
};
```

n1.Multiply(n2);

return 0;

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber *this, RationalNumber other)
    {
        this->boonja *= other.boonja;
        this->boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.Multiply(&n1, n2);
}
```

코드로 표현한다면 이런 느낌이 돼요.
(아쉽게 C++20에서는 실제로 프로그래머가 이렇게 적을 순 없어요)
(이런 느낌이긴 한데 실제로는 this = 0은 안 돼요)

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

```
// 미리 적어 둔 유리수 class 정의
class RationalNumber
{
public:
    int boonja;
    int boonmo;

    void Multiply(RationalNumber *this, RationalNumber other)
    {
        this->boonja *= other.boonja;
        this->boonmo *= other.boonmo;
    }
};
```

```
// 나중에 적게 될 main() 예시
int main()
{
    RationalNumber n1, n2;

    n1.Multiply(&n1, n2);
}
```

RationalNumber::Multiply()를 호출하는 수식에는 반드시 . 연산자가 붙어 있어야 하며, 이 수식은 . 연산자의 좌항 수식을 계산한 결과값을 0번째 인수 자리에 담도록 컴파일돼요. (보통은 편의상 register를 쓰긴 해요. 아무튼 담는 건 맞아요)

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

// 미리 적어 둔 유리수 class 정의

```
class RationalNumber {  
public:  
    int boonja;  
    int boonmo;  
  
    void Multiply(RationalNumber *this, RationalNumber other)  
    {  
        this->boonja *= other.boonja;  
        this->boonmo *= other.boonmo;  
    }  
};
```

RationalNumber::Multiply() 정의 내용물을 작성하는 입장에서는,
마치 '컴파일러가 자동으로 지정해 둔 0번째 인수 이름'인 것처럼
this 키워드를 사용할 수 있어요.

n1.Multiply(&n1, n2);

return 0;

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

// 미리 적어 둔 유리수 class 정의

```
class RationalNumber {
public:
    int boonmo;

    void Multiply(RationalNumber *this, RationalNumber other)
    {
        this->boonja *= other.boonja;
        this->boonmo *= other.boonmo;
    }
};
```

다시 말하면, 해당 키워드를 **함수 정의** 안에서 **수식**으로써 적을 수 있어요.
물론 이를 **연산자**(여기선 `->`긴 하지만) 좌항 자리에 적는 것 또한 가능하고,
이를 통해 **멤버 변수**를 건드리거나 다른 **멤버 함수**를 호출하는 것 또한 가능해요.

// 나중에 적게 될 main() 예시

```
n1.Multiply(&n1, n2);
```

```
return 0;
```

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

// 미리 적어 둔 유리수 class 정의

```
class RationalNumber {
public:
    int boonmo;

    void Multiply(RationalNumber *this, RationalNumber other)
    {
        this->boonja *= other.boonja;
        this->boonmo *= other.boonmo;
    }
};
```

멤버 함수 정의를 적을 때는
이러한 느낌의 수식(이번 호출과 연계된 Data를 다루는 수식)을 적을 일이 많으므로...

// 나중에 적게 될 main() 예시

```
n1.Multiply(&n1, n2);
```

```
return 0;
```

클래스 다시 보기

- 약간만 더 들여다 보고 넘어갑시다.
이제까지 봤던 예시 코드들 중 일부를 간추려 왔어요.

// 미리 적어 둔 유리수 class 정의

```
class RationalNumber {
public:
    int boonmo;

    void Multiply(RationalNumber *this, RationalNumber *other) {
        boonja *= other.boonja;
        boonmo *= other.boonmo;
    }
};
```

멤버 함수 정의를 적을 때는
이러한 느낌의 수식(이번 호출과 연계된 Data를 다루는 수식)을 적을 일이 많으므로...
C++에서는 꼭 필요하지 않은 경우 this->를 생략해도 되도록 만들어 놓았어요.

// 나중에 적게 될 main() 예시

```
n1.Multiply(&n1, n2);
```

Python에서 그냥 int 적었을 때 적당히 __builtins__.int로 간주했듯
함수 정의 안에 이름 boonja에 대한 선언이 적혀 있거나 하지 않다면
this->boonja로 간주해 줘요(일반적으로 그게 의도에 부합함!)

클래스 다시 보기

- 요점은, 기본적으로 **멤버 이름**들은
 - . **연산자** 우항 자리에 적어야 의미를 갖는다는 것이에요.
 - **멤버 Data 이름**을 적는 것은 전반부에서 구경했듯 **offset**을 적용하는 셈이 돼요
- **멤버 Code 이름**을 적어 **멤버 함수** 호출식을 구성할 때
 - . **연산자** 왼쪽에 적어 둔 **수식**을 **계산**한 **결과값**을 0번째 인수 자리에 담아 호출
- 이렇게 구성되어 있기에,
우리는 **클래스** 개념을 활용하여 Data와 Code의 연계를 보다 수월하게 할 수 있어요
 - '클래스 = Data & Code'라 불려도 될 듯!

마무리

- 요점은, 기본적으로 **멤버 이름**들은
 - **연산자** 우항 자리에 적어야 의미를 갖는다는 것이에요.
 - **멤버 Data 이름**을 적는 것은 전반부에서 구경했듯 **offset**을 적용하는 셈이 돼요
 - **멤버 Code 이름**을 적어 **멤버 함수** 호출식을 구성할 때
 - **연산자** 왼쪽에 적어 둔 **수식**을 **계산**한 **결과값**을 0번째 인수 자리에 담아 호출
 - 이렇게 구성되어 있기에,
우리는 **클래스** 개념을 활용하여 Data와 Code의 연계를 보다 수월하게 할 수 있어요
 - '클래스 = Data & Code'라 불려도 될 듯!
- 후반부 시작을 위해 다시 볼만한 내용은 여기까지예요.
 - 오늘은 최종 목표는 없어요. 느낀 점 적고 집에 갑시다