11-1. Overload

창의적소프트웨어프로그래밍
2022년도 여름학기
Racin

10일차 내용

- **이름**들 사이의 **소속** 관계에 대해 살펴봤어요.
  - 얘 **선언**을 쟤 **정의** 중괄호 안에 적어 두었다면, 얘는 쟤 **소속**이 돼요
    - 아 물론 여기서도 인수 **선언**같이 어떤 **정의**에 붙은 **선언**은 그 **정의** 중괄호 안에 적은 셈 쳐요
    - 아무튼 이게 **소속** 관계를 구성하는 유일한 방법이에요

  - 컴파일러는 context에 따라 **lookup**을 해 줘요
    - **함수 정의** 내용물을 적을 때 그 **함수**의 소속에 따라 풀 네임을 적지 않아도 됐어요
      (물론 **구조체**/**class**/namespace **정의** 내용물 적을 때도 가능해요)

    - . **수식** 우항 자리 **이름**을 적을 때 좌항 **형식**에 따라 풀 네임을 적지 않아도 됐어요

    - 원한다면 : : **연산자**를 적어 가며 내가 직접 '저기 있는 쟤'와 같이 **이름**을 특정할 수 있어요

10일차 내용

- **Object**들 사이의 **포함** 관계에 대해 살펴봤어요.
  - 어떤 **구조체/class object**에는,
    해당 **형식** nonstatic Data **멤버 object**가 **포함**돼요
    - 이 때 구성되는 **포함** 관계는
      영속적(같이 살고 같이 죽음),
      물리적(실제로 **멤버 object**는 메모리 위의 그 **구조체/class object** '안'에 들어 있음)이에요
    - 다른 수업에서는 논리적 **포함** 관계를 논의할 수도 있어요

10일차 내용

- ODR(One Definition Rule)
  - 어떤 **이름**에 대한 **정의**는 단 한 번만 되어 있어야 해요
    - **함수 정의**가 여럿이면 뭔가 이상함
  - 그렇다 보니 static Data **멤버 선언**은 기본적으로 **정의**를 수반하지 않아요.
    그래서 다른 어딘가에 **정의**를 단 한 번 적어 주어야 해요
    - **Typename 정의**는 '연결(링크)'의 대상이 아니라
      그 **형식**을 사용하는 모든 .cpp 파일에 (복붙해 가며) 적어야 해서 그래요
  - 우회하고 싶다면 inline 붙이면 돼요
    - '재**정의** 오류'가 날 것 같은 상황에서 꼭 참아 줘요(**정의**들 중 하나만 골라 연결해 줘요)
    - **멤버 함수 정의**의 경우 VS가 종종 자동으로 붙여줬어요

10일차 내용

- Nonstatic / static, Data / Code **멤버**
  - 별 것 아닌 것 같지만 이 네 가지 케이스는 서로 완전히 달라요
  - 따라서 지금 시점에 미리 서로 구분해서 가져가 두면 좋겠어요
  - 아무튼, 이제 `static` specifier는 의미가 총 네 가지가 되었어요

오늘 내용

- 들어본 적 있는 단어 **overload**를 구경해 봅니다.
  - (일반), &, &&, 그리고 const &
  - 여러 버전 Code의 공존 가능성
  - 컴파일러가 어떤 **수식**에 대한 해석 방법을 context에 입각하여 찾는(**resolve**하는) 방법
  - '임시' **object**
- **이름**에 대한 **제한**(restriction) 이야기를 구경해 봅니다.
  - const: 그 **이름**을 사용하여 저 형식 **object**를 변경하는 것을 제한
  - private: **이름** 사용 자체를 제한
- 오늘은 최종 목표는 없어요.

이번 시간에는

- 들어본 적 있는 단어 **overload**를 구경해 봅니다.
  - (일반), &, &&, 그리고 const &
    - 새 Data **이름 선언**을 적을 때 어떤 버전을 고를 것인지에 대해 소개해요
  - 여러 버전 Code의 공존 가능성,
    컴파일러가 어떤 **수식**에 대한 해석 방법을 context에 입각하여 찾는(**resolve**하는) 방법
    - **함수 정의**를 적는 입장, **함수** 호출식을 적는(또는 해석하는) 입장으로 나누어 정리해요
    - **Lookup**이 그러했듯 **resolve** 이야기도 오늘은 기초적인 것만 다룰 예정이에요
  - '임시' **object**
    - 인수 **object**, return **object** 등 한시적으로 살아 있는 **object**를 구경해 봐요
  - 10일차 내용까지 종합해서 정리하는 느낌으로 구경해 보면 좋을 것 같아요

이번 시간에는

- 시작해 봅시다.

(일반), &, &&

- 아래 코드를 봅시다:
  - 간만에 하는 **선언** 공부예요

```
int number { 3 };

int &ref   { number };

int &&rref { 3 };
```

(일반), &, &&

- 아래 코드를 봅시다:
  - 간만에 하는 **선언** 공부예요

```
int number { 3 };
int &ref   { number };
int &&rref { 3 };
```

number는 int **변수 이름**입니다.

ref는 int & **이름**입니다?

rref는 int && **이름**입니다?

(일반), &, &&

- 아래 코드를 봅시다:
  - 간만에 하는 **선언** 공부예요

```
int number { 3 };
int &ref   { number };
int &&rref { 3 };
```

수식 number는 int **형식 수식**입니다.

수식 ref는 int & **형식 수식**입니다?

수식 rref는 int && **형식 수식**입니다?

(일반), &, &&

- 아래 코드를 봅시다:
  - 간만에 하는 **선언** 공부예요

```
int number { 3 };
int &ref   { number };
int &&rref { 3 };
```

number는 int **변수 이름**입니다.

ref는
int **object** 하나에 대한 **lvalue reference 이름**입니다!

rref는
int **object** 하나에 대한 rvalue **reference 이름**입니다!

(일반), &, &&

- C 구경하던 당시의 원칙이 깨지긴 했지만...
  아무튼 우리는 declarator에 & 또는 &&를 붙여
  그 **이름**의 **형식**이 **lvalue** / rvalue **reference 형식**이 되도록 만들 수 있어요.
  - *, [ ], ( ), *const 에 추가로 두 가지 선택지가 생긴 셈이에요

  - 일단 우리 수업에서는,
    형식을 논의할 때는 &, && **형식** 또는 **reference 형식**(구분할 필요가 없는 경우)
    해당 **형식 이름**을 논의할 때는 그냥 &, && 또는 **reference**라 부를께요
    - 예를 들어, int &func( ) 는 int **reference**를 return해요.

(일반), &, &&

- &, &&

(일반), &, &&

- &, &&
  - (중요)둘 다 어떤 **object**를 특정하기 위해 사용할 수 있어요
  - (중요)둘 다, **정의**하는 시점에 바로 initialize까지 해 주어야 해요

(일반), &, &&

- &, &&
  - (중요)둘 다 어떤 **object**를 특정하기 위해 사용할 수 있어요
    - 그렇다 보니 실제 메모리에는 결국 **위치 값**이 담기게 돼요
  - (중요)둘 다, **정의**하는 시점에 바로 initialize까지 해 주어야 해요
    - 일반 **선언**이면 initializer를 반드시 적어 두어야 해요
    - 어떤 **함수 정의**의 인수 **선언**이면 호출 과정에서 initialize하게 돼요
    - (매우 중요)따라서, **reference 이름**을 사용할 수 있다면
      이를 통해 언제든 대상 **형식 object**를 하나 특정할 수 있어요

(일반), &, &&

- &, &&
  - (중요)둘 다 어떤 **object**를 특정하기 위해 사용할 수 있어요
  - (중요)둘 다, **정의**하는 시점에 바로 initialize까지 해 주어야 해요
    - (매우 중요)따라서, **reference 이름**을 사용할 수 있다면
      이를 통해 언제든 대상 **형식 object**를 하나 특정할 수 있어요
  - (중요)기본적으로 '**reference** (자리에 담긴) **값**'은 변경할 수 없고,
    address of / size of **reference 값**이 나오는 **수식**은 적을 수조차 없어요
    - ?

(일반), &, &&

- 아래 코드를 봅시다:

```
int number { 3 };
int &ref   { number };

    &  ref   ==      & number
sizeof  ref   == sizeof number
sizeof(int &) == sizeof( int )
```

(일반), &, &&

- 아래 코드를 봅시다:

```
int number { 3 };
int &ref   { number };

      &  ref
sizeof   ref
sizeof(int &)
```

수식 ref는, 뒤에 & **연산자**가 붙어 있다 하더라도
이 시점에 바로 int **형식** (**lvalue**) **수식**으로 간주돼요.
Object 측면에서는, initialize를 저렇게 해 놨으니
바로 number **object**를 특정한 셈이 돼요.

(일반), &, &&

- 아래 코드를 봅시다:

```
int number { 3 };
int &ref   { number };
```

이러한 목표를 달성하기 위해서는
아무튼 '뭐로 initialize했음' 정보를 담아 둘 필요가 있어요.
지금은 항상 number **object**를 특정하지만,
이게 인수 **이름**이었다면 매 호출마다 다른 **object**를 특정할 수도 있지요.

그래서 실제로는 해당 **object**에 대한 **위치 값**을 'ref 자리'에 담아요.

(일반), &, &&

- 아래 코드를 봅시다:

```
int numb
int &ref
```

그러나 **reference**는 & **연산자** 붙기도 전에 deduce되어버리기 때문에
그 'ref 자리'에 대한 **위치 값**이 나오는 **수식**은
우리가 직접적으로 적을 수는 없어요.
물론, 그 자리 자체가 나오는 (**lvalue**) **수식**은 더더욱 못 적어요!

```
    & ref    ==      & number
sizeof ref   == sizeof number
sizeof(int &) == sizeof( int )
```

(일반), &, &&

- 아래 코드를 봅시다:

```
int numb
int &ref

    &  ref   =      & number
sizeof  ref  == sizeof number
sizeof(int &) == sizeof( int )
```

비슷한 느낌으로, sizeof **연산자**를 사용할 때도
우리는 'ref 자리'가 몇B인지 직접 확인할 수 없어요.
수식 sizeof(int &)조차도, 자동으로 **수식** sizeof(int)로 간주돼요!

(일반), &, &&

- &, &&
  - (중요)둘 다 어떤 **object**를 특정하기 위해 사용할 수 있어요
  - (중요)둘 다, **정의**하는 시점에 바로 initialize까지 해 주어야 해요
    - (매우 중요)따라서, **reference 이름**을 사용할 수 있다면
      이를 통해 언제든 대상 **형식 object**를 하나 특정할 수 있어요
  - (중요)기본적으로 '**reference** (자리에 담긴) **값**'은 변경할 수 없고,
    address of / size of **reference 값**이 나오는 **수식**은 적을 수조차 없어요
    - !

(일반), &, &&

- &, &&
  - (중요)둘 다 어떤 **object**를 특정하기 위해 사용할 수 있어요
  - (중요)둘 다, **정의**하는 시점에 바로 initialize까지 해 주어야 해요
    - (매우 중요)따라서, **reference 이름**을 사용할 수 있다면
      이를 통해 언제든 대상 **형식 object**를 하나 특정할 수 있어요
  - (중요)기본적으로 '**reference** (자리에 담긴) **값**'은 변경할 수 없고,
    address of / size of **reference 값**이 나오는 **수식**은 적을 수조차 없어요
  - (치명적으로 중요)따라서, **reference 이름**을 사용할 수 있다면
    이를 통해 언제든, 동일한, 대상 **형식 object**를 하나 특정할 수 있어요

(일반), &, &&

- (일반), &, &&

(일반), &, &&

- (일반), &, &&
  - 아래 코드를 봅시다:

```
void func(int arg)   { arg = 0; }

void func(int &arg)  { arg = 0; }

void func(int &&arg) { arg = 0; }
```

(일반), &, &&

- (일반), &, &&
  - 아래 코드를 봅시다:

세 버전 모두, 각각의 **정의** 내용물을 적는 동안에는
인수 이름 arg를 어디서 몇 번 사용하든
반드시 동일한 int **object**를 특정할 수 있어요!

```
void func(int arg)   { arg = 0; }

void func(int &arg)  { arg = 0; }

void func(int &&arg) { arg = 0; }
```

(일반), &, &&

- (일반), &, &&
  - 아래 코드를 봅시다:

소유 이야기를 떠올려 보면,
이 arg **object**는 100% func()꺼라고 말할 수 있어요.

```
void func(int arg)  { arg = 0; }

void func(int &arg)  { arg = 0; }

void func(int &&arg) { arg = 0; }
```

(일반), &, &&

- (일반), &, &&
  - 아래 코드를 봅시다:

```
void func( int arg)
void func( int &arg)
void func( int &&arg) { arg = 0; }
```

소유 이야기를 떠올려 보면,
이 arg **object**는 100% func()꺼라고 말할 수 있어요.

이 버전도, arg (로 특정되는) **object**는
호출 과정에서 방금 갓 만든 **object**거나
원 **소유**권자가 잠시 양도한 **object**니까
아무튼 내용물 **실행**하는 동안에는 func()꺼라고 말할 수 있어요.

(일반), &, &&

- (일반), &, &&
  - 아래 코드를 봅시다:

기본적으로, **object**에 대한 **소유**권자는
해당 object에 write할 권한 또한 가진다고 말할 수 있어요.

아 물론, 내용물 **실행** 후 반납할 예정이니
함부로 delete하는 건 선 넘는거임

```
void func(int arg)    { arg = 0; }

void func(int &arg)   { arg = 0; }

void func(int &&arg)  { arg = 0; }
```

(일반), &, &&

- (일반), &, &&
  - 아래 코드를 봅시다:

```
void func(int arg)

void func(int &arg)   { arg = 0; }

void func(int &&arg) { arg = 0; }
```

반면 이 경우, arg **object**는 내꺼가 아니에요.
그래서, 어쩌면, 원 소유권자가 그 **object**를 '변경'하는 것을 희망하지 않을 지도 몰라요.

(일반), &, &&, 그리고 const &

- (일반), &, &&, 그리고 const &
  - 아래 코드를 봅시다:

```
void func(int arg)    { arg = 0; }

void func(int &arg)

void func(int &&arg)  { arg

void func(const int &arg)   { }
```

그런 분들을 위해 우리는 이런 버전을 추가로 **정의**해둘 수 있어요!
arg **object**를 변경할 수 없으므로, 필요한 경우 우리는
복사 등의 우회 방법을 통해 목표를 달성하게 될 거예요.

(일반), &, &&, 그리고 const &

- (일반), &, &&, 그리고 const &
  - (중요)이 친구들은 **소유** 관점에서 차이를 가져요
    - 일반 버전 **이름**으로 특정되는 **object**는 '내 것'이 맞아요
    - &로 특정되는 **object**는 내 것은 아니에요. 그러나 변경 가능해요
    - &&로 특정되는 **object**는 내 내용물 **실행** 도중에는 내 것이 맞고, 변경 가능해요
    - const &(보통 const **reference**라 부름)로 특정되는 **object**는 내 것이 아니며, 변경 불가능해요

(일반), &, &&, 그리고 const &

- (일반), &, &&, 그리고 const &
  - **소유** 관점에서 차이를 가져요
    - 일반 버전 **이름**으로 특정되는 **object**는 '내 것'이 맞아요
    - &로 특정되는 **object**는 내 것은 아니에요. 그러나 변경 가능해요
    - &&로 특정되는 **object**는 내 내용물 **실행** 도중에는 내 것이 맞고, 변경 가능해요
      - (주의) 'rvalue' **reference**지만, 그 **이름**을 **수식**으로써 적으면 그 **수식**은 **lvalue 수식**이에요!
        - » 인수 **이름**을 **수식**으로써 적을 수 있는 동안에는 그 **object**는 내 것이 맞다고 생각하면 그럴싸 함
    - const &(보통 const **reference**라 부름)로 특정되는 **object**는 내 것이 아니며, 변경 불가능해요

(일반), &, &&, 그리고 const &

- 그럴싸 하지요?
  - 방금 정리해본 내용은 기본적으로 일반 **선언**이나 인수 **선언**을 전제하고 있는데, '**reference**를 return하는 **함수**' 등등도 그럭저럭 비슷하려니 해 둡시다

(일반), &, &&, 그리고 const &

- 그럴싸 하지요?
  - 방금 정리해본 내용은 기본적으로 일반 **선언**이나 인수 **선언**을 전제하고 있는데, '**reference**를 return하는 **함수**' 등등도 그럭저럭 비슷하려니 해 둡시다

- 달리 말하면, **선언** 읽는 법 자체는 변하지 않았어요. 따라서...

(일반), &, &&, 그리고 const &의 어두운 면

- 아래 코드를 봅시다:
  - 간만에 하는 지독한 **선언** 공부예요. 여긴 강사만 해 볼께요

```
int f1();
int &f2();
int (&rf)() { f1 };

int *p1;
int &*p2;
int *&rp  { p1 };

int a1[3];
int &a2[3];
int (&ra)[3] { a1 };
```

(일반), &, &&, 그리고 const &의 어두운 면

- 잊으세요.
  - C로 구경해 본 그 규칙은 여전히 유효하고, 원한다면 활용할 수 있어요
  - 그렇기는 하지만 보통은 **선언**의 declarator 적을 때
    &, &&를 다른 기호들과 심각하게 섞어 가며 사용하지는 않는 편이에요
  - 뭐 그렇기는 하지만 '왜 안 되나'를 생각해 보면 그럭저럭 복습이 될 수 있을 것 같아요

코드 구경

- 대충 정리해 보았으니,
  CSP_11_1_yeshi_1.cpp를 열고 같이 구경해 봅시다.
  - 방금 슬쩍 다시 튀어나온 const specifier를 곁들여서
    선언들을 여럿 준비해 왔어요

- 주욱 본 다음 뒷 슬라이드에서 정리해 볼께요.

복습을 위한 슬라이드

- **Overload**
  - Code에 대해 성립해요(Data **overload**나 **형식 overload**는 성립하지 않아요)

  - **이름**이 같지만 **인수 형식**이 다른 여러 버전을 준비하는 것을 의미해요
    - **Overload** 자체는 자유롭게 할 수 있어요

복습을 위한 슬라이드

- 컴파일러의 **resolve**
  - 컴파일러가, 어떤 **수식**을 어떻게 컴파일할 것인지를,
    context에 입각하여 정하는 작업을 의미해요

  - (중요)만약 **overload** 등으로 인해 선택지가 여럿 존재한다면,
    컴파일러가 여러 가능한 버전들 중 적합한 것 단 하나를 고를 수 있어야 해요
    - 적합한 버전이 여럿 존재한다면 오류가 나요
    - 가능한 버전이 하나도 없다면 오류가 나요

  - **Resolve** 과정에서 **lookup** 또한 수행될 수 있어요
    - 덧셈식의 좌항/우항이 유리수 **형식**이라면, + **연산자 정의**를 유리수 **정의**에서 찾기 시작해요

  - 적합한 버전이 여럿 존재하는 상황에서 필요하다면 내 **수식**의 **형식**을 조절할 수 있어요
    - Casting **연산자**나 `std::move<>()` 등을 사용할 수 있어요

복습을 위한 슬라이드

- 컴파일러의 **resolve**
  - 기본적으로 컴파일러는...
    - **함수** 호출식 괄호 안 **수식**(**멤버** 함수 호출식의 경우 . 연산자 좌항 포함)
      ...의 **형식**에 const가 붙어 있다면 const 붙은 버전을 골라줘요
    - const가 안 붙어 있다면, 가급적 const가 안 붙은 버전을 우선적으로 골라줘요
    - 인수 **형식**이 적합한 버전이 없는 경우,
      함수 호출식 괄호 안 **수식**을 해당 **형식 수식**으로 간주할 수 있는 방안을 모색해요
      - 예를 들어, 나는 3을 적었지만 유리수 받는 버전밖에 없는 경우,
        3으로 initialize하는 '임시 유리수 **object**'를 **생성**해 전달하도록 컴파일할 수 있어요
  - 그 외에 여러 규칙들이 있지만, 지금은 일단 이 정도만 가져가면 될 것 같아요

복습을 위한 슬라이드

- '임시' **object**
  - C++에서는 '**object** = **값** + 칸'이므로,
    함수 호출식, 달리 말하면 **수식**을 **계산**하는 과정에서
    '인수 **reference**로 특정할 **object**'나 'return **object**'를 **생성**하게 될 수 있어요

  - 이러한 **object**들은 automatic **위치**에 생성되며,
    따라서 **파괴** 또한 적당한 시점에 이루어지도록 컴파일러가 잘 케어해 줘요

복습을 위한 슬라이드

- '임시' **object**
  - C++에서는 '**object** = **값** + 칸'이므로,
    함수 호출식, 달리 말하면 **수식**을 **계산**하는 과정에서
    '인수 **reference**로 특정할 **object**'나 'return **object**'를 **생성**하게 될 수 있어요

  - 이러한 **object**들은 automatic **위치**에 생성되며,
    따라서 **파괴** 또한 적당한 시점에 이루어지도록 컴파일러가 잘 케어해 줘요

  - 기본적으로 임시 **object**들은, 특별한 생명 연장 기법을 적용하지 않는 한은
    생성을 야기한 그 **수식**이 포함된 전체 **수식**의 **계산**이 끝나는 시점까지만 살 수 있어요
    - 따라서, 덧셈 및 곱셈을 연거푸 하더라도,
      그 전체 **수식 계산** 끝날 때까지는 방금 **생성**한 임시 **object**를 재활용할 수 있어요
      - 11-2에서 또 유리수 다룰 예정이니 그 때 한 번 더 확인해 봐요

마무리

- 이 정도 해 두면
  C++에서 여러 버전 Code들을 구성할 준비는 어느 정도 끝날 것 같아요.

- 잠시 쉬었다가 이번에는 다시 **이름** 이야기로 돌아가 봅시다.