Item 23) Understand move and forward

Before we start

Move semantics

- 컴파일러에게 비용이 많이 들어가는 copy에서 move연산을 가능하게 하게 하는 것. (더 이상 이 객체가 필요 없어요!!!)
- move constructor와 move assignment operator는 "의미론적 이동"을 가능하게 한다.
- Move semantic은 "move-only type"객체의 이동을 가능하게 한다
  - Example) `std::unique_ptr`, `std::future`, `std::thread`

Perfect forwarding

- 임의의 인수를 받아들이는 함수 템플릿을 작성할 수 있게 해준다.
- 인수들을 다른 함수로 전달할 때, 전달되는 함수가 원래 전달받은 인수와 정확히 같은 인수를 받게 해준다.

Understand move and forward

`std::move`와 `std::forward`의 오해와 진실

`std::move`와 `std::forward`를 배우면서 가장 많이 하는 오해는, `std::move`가 실제로 특정 객체를 이동시킨다는 것과 `std::forward`가 객체를 전달(forward)한다는 것이다.

`std::move`는 어떠한 것도 이동시키지 않는다.
`std::forward`는 실제로 어떤 것을 "전달"하지 않는다.

특히 두 개의 함수 모두 실행 가능한 코드(executable code)를 생성하지 않는다, 이들은 그저 함수(함수 템플릿)이며, "형변환"을 진행한다.

런타임에는 2개 모두 실행 가능한 코드를 생성하지 않는다.

std::move : 무조건적으로 argument를 rvalue로 변경(casting)시킴.
std::forward : 특정 조건이 만족되면 casting을 진행함.

`std::move`의 구현

```
template<typename T>
typename std::remove_reference<T>::type&& move(T&& param) {
    using ReturnType = typename std::remove_reference<T>::type&&;
    return static_cast<ReturnType>(param);
}
```

위의 코드를 살펴보면, `std::move`는 객체를 실제로 "이동"시키는 것이 아니라,
객체를 `rvalue` 참조로 casting을 하여 이동 연산자나 이동 대입 연산자가 호출될 수 있게 하는 것이다

- `std::move`는 템플릿 함수로, `T`라는 임의의 타입을 받아들인다. 이 `T`는 전달된 `param`의 타입에 따라 결정된다.
- `T&&`는 **universal reference**로, `param`이 lvalue이면 lvalue참조, rvalue이면 rvalue참조가 된다.
- `std::remove_reference<T>`는 타입 `T`에서 참조를 제거한다.
- 이 작업은 `T&&`이 lvalue 참조가 될 위험을 방지한다.
- `return static_cast<ReturnType>(param);`은 `param`을 `rvalue` 참조로 캐스팅하여 반환한다. `param`이 항상 `rvalue` 참조로 캐스팅된다.

결론적으로 `std::move`는 argument를 `rvalue`로 casting을 하는 것이다.

- `rvalue_cast`가 더 맞는 이름일 수도 있다 ㅎㅎ

---

std::move의 이동가능성

`std::move`가 실행되었다고 해서, 모든 rvalue가 moving이 되는 것은 아니다. 즉, `std::move`를 사용하게 되면 컴파일러에게 "해당 객체가 이동될 수 있음을" 알려주는 것이다.

즉, rvalue는 moving의 `후보`가 되는 것이다.

그렇다면 rvalue가 된 객체가 moving이 되지 않는 경우가 있을까?? 아래의 코드를 살펴보자.

```
#include <iostream>
#include <string>
```

```
class Annotation {
public:
    // const std::string을 값으로 전달받음
    explicit Annotation(const std::string text)
        : value(std::move(text)) // text를 이동하려고 하지만, 복사됨
    {
        std::cout << "생성자 호출: value = " << value << std::endl;
    }

private:
    std::string value;
};

int main() {
    std::string annotation = "This is a sample annotation";
    Annotation note(annotation); // 복사 생성자 호출됨

    std::cout << "원본 문자열: " << annotation << std::endl; // 원본 문자열은 그대로 남아 있음
    return 0;
}
```

위의 생성자를 살펴보자, 생성자에 있는 `const`때문에 문제가 발생한다.

`text`를 복사하지 않고 이동(move)하기 위해 `std::move`를 사용하여 `text`를 `value`멤버 변수로 이동하려고 한다.

그러나 `text`가 `const`로 선언되었기 때문에, `std::move`는 `text`를 `const std::string&&` 타입의 rvalue로 변환할 뿐, const의 특성은 유지된다.

결과적으로 `text`가 이동되지 않고 **복사**된다. 즉, `const`로 인해 `std::move`가 의미하는 "이동"은 불가능해진다.

여기서는 2가지의 결론이 나타난다.

==1. std::move를 하고 싶으면 object를 const로 선언하지 말아라. std::move객체는 copy가 될 것이다.

2.std::move는 실제로 객체를 이동시키지 않는다. 즉 std::move를 실행했다고 해서, 반드시 객체가 이동하는 것은 아니다.==

`std::forward`에 대해

`std::forward`는 "conditional cast"이다.
아래의 코드를 살펴보면서 `std::forward`에 대해 알아보자.

예시코드

```
void process(const Widget& lvalArg); // lvalue를 처리하는 함수
void process(Widget&& rvalArg); // rvalue를 처리하는 함수

template<typename T> // 템플릿 함수로 인수를 전달
void logAndProcess(T&& param) // param을 process로 전달
{
    auto now = // 현재 시간 가져오기
        std::chrono::system_clock::now();
    makeLogEntry("Calling 'process'", now);
    process(std::forward<T>(param)); // param을 process로 전달 (완벽한 전달)
}

Widget w;

logAndProcess(w); // call with lvalue
logAndProcess(std::move(w)); // call with rvalue
```

`logAndProcess`안에서, `param`의 parameter들은 `process`라는 function으로 진행된다. `process`는 lvaue와 rvalue모두 오버로딩 되어 있다.
`logAndProcess`를 lvalue로 호출하게 되면 일반적으로 lvalue로 전해진다.
그리고 `logAndProcess`를 rvalue로 호출하게 된다면, rvalue로 오버로딩된 `process`가 진행되게 된다.

`std::forward`가 `lvalue`인지 `rvalue`인지 어떻게 알 수 있나요??

- `logAndProcess` 함수의 템플릿 매개변수 `T`에 그 정보가 인코딩 되어 있다.
- `T`는 `logAndProcess`가 호출될 때 전달된 인수의 타입에 따라 결정된다.

그렇다면 뭐 사용해요??

std::move의 장점

- `std::move`를 사용하면 `lvalue`객체를 `rvalue` 참조로 변환하여 이동 생성자나 이동 대입 연산자가 호출되도록 한다. 이를 통해 불필요한 복사를 방지하고 효율적으로 리소스를 관리할 수 있다.

** `std::move` 와 `std::forward` 의 차이점

- **std::move**
  - 인수를 무조건 `rvalue` 참조로 변환한다. 즉, `lvalue` 객체를 `rvalue` 로 변환하여 이동이 가능하도록 한다.
- **std::forward**
  - 특정 조건에 따라 인수를 `rvalue` 또는 `lvalue` 로 전달한다.
  - `std::forward` 는 인수의 원래 참조 타입(lvalue인지 rvalue인지)을 유지하면서 전달한다.

따라서, `std::forward` 는 완벽한 전다(perfect forwarding)을 위해 설계된 함수이며, `std::move` 는 무조건적인 rvalue 변환을 위해 설계된 함수이다.

PS) `std::move` 를 사용하고도 이동이 안되는 경우(by gpt)

1. `const` 객체에 `std::move` 를 사용한 경우

`const` 객체는 수정될 수 없으므로 이동 연산을 수행할 수 없습니다. `std::move` 를 사용해도 `const` 객체는 이동되지 않고 복사됩니다.

```cpp
#include <iostream>
#include <string>

int main() {
    const std::string str = "Hello";
    std::string movedStr = std::move(str); // 이동이 아닌 복사 생성자 호출

    std::cout << "movedStr: " << movedStr << std::endl; // "Hello"
    std::cout << "str: " << str << std::endl; // "Hello" (원본 유지)

    return 0;
}
```

- **설명**: `str` 은 `const` 특성을 가지므로, `std::move` 를 사용해도 이동 생성자가 호출되지 않습니다. 대신 복사 생성자가 호출되어 `movedStr` 에 `str` 의 내용이 복사됩니다. `str` 은 `const` 이기 때문에 원본 값이 그대로 유지됩니다.

2. 이동 생성자 또는 이동 대입 연산자가 정의되지 않은 경우

클래스에 이동 생성자나 이동 대입 연산자가 정의되지 않은 경우, `std::move` 를 사용해도 복사 생성자나 복사 대입 연산자가 호출됩니다.

```cpp
#include <iostream>
#include <string>

class MyClass {
public:
    MyClass(const std::string& str) : data(str) {}
    MyClass(const MyClass& other) {
        std::cout << "복사 생성자 호출" << std::endl;
        data = other.data;
    }
    // 이동 생성자가 정의되지 않음
    MyClass& operator=(const MyClass& other) {
        std::cout << "복사 대입 연산자 호출" << std::endl;
        if (this != &other) {
            data = other.data;
        }
        return *this;
    }

private:
    std::string data;
};

int main() {
    MyClass obj1("Hello");
    MyClass obj2 = std::move(obj1); // 복사 생성자 호출

    return 0;
}
```

- **설명**: `MyClass` 에 이동 생성자가 정의되어 있지 않기 때문에 `std::move` 를 사용해도 복사 생성자가 호출됩니다. `obj1` 의 내용이 `obj2` 로 복사되고, 이동은 이루어지지 않습니다.

3. 기본 제공 이동 생성자가 삭제된 경우

클래스에 특정 조건이 맞지 않아 이동 생성자나 이동 대입 연산자가 자동으로 삭제된 경우에도 이동이 이루어지지 않습니다. 예를 들어, `const` 멤버 변수를 가지거나, `std::unique_ptr`과 같은 이동 불가능한 멤버 변수를 가진 클래스의 경우입니다.

```
#include <iostream>
#include <memory>

class MyClass {
public:
    // 기본 생성자
    MyClass() : ptr(new int(10)) {}

    // 복사 생성자 정의
    MyClass(const MyClass& other) : ptr(new int(*other.ptr)) {
        std::cout << "복사 생성자 호출" << std::endl;
    }

    // 이동 생성자가 자동으로 삭제됨
    MyClass& operator=(MyClass&&) = delete;

private:
    std::unique_ptr<int> ptr; // 이동 불가능한 멤버
};

int main() {
    MyClass obj1;
    // MyClass obj2 = std::move(obj1); // 컴파일 오류: 이동 생성자가 삭제됨

    return 0;
}
```

- **설명**: `MyClass`는 `std::unique_ptr`와 같은 이동 불가능한 멤버를 가지고 있습니다. 이 경우, 이동 생성자가 자동으로 삭제되며, `std::move`를 사용해도 이동이 이루어지지 않습니다.

4. `std::move` 이후 객체를 사용하는 경우

`std::move`는 객체의 소유권을 이전할 가능성을 표시할 뿐 실제로는 아무것도 하지 않습니다. 따라서, `std::move` 이후에도 객체가 사용된다면, 이동 생성자가 호출되지 않을 수 있습니다.

```
#include <iostream>
#include <vector>

void process(std::vector<int>& vec) {
    std::cout << "lvalue 참조로 process 호출" << std::endl;
}

void process(std::vector<int>&& vec) {
    std::cout << "rvalue 참조로 process 호출" << std::endl;
}

int main() {
    std::vector<int> vec = {1, 2, 3};

    process(std::move(vec)); // rvalue 참조로 호출
    process(vec); // 다시 lvalue로 호출

    return 0;
}
```

- **설명**: `std::move(vec)`는 `vec`을 `rvalue`로 캐스팅하여 첫 번째 `process` 함수가 호출됩니다. 그러나 그 이후에 `vec`을 다시 사용하면, `vec`은 `lvalue`로 처리됩니다. 따라서 `std::move`가 실제로 `vec`을 이동시키지 않았다는 것을 알 수 있습니다.

5. 추상 클래스의 포인터나 참조를 `std::move`한 경우

추상 클래스의 포인터나 참조는 객체 자체를 이동할 수 없으므로, `std::move`는 단순히 포인터나 참조를 변환할 뿐 실제 이동은 이루어지지 않습니다.

```
#include <iostream>

class Base {
public:
    virtual void show() const = 0; // 순수 가상 함수
};

class Derived : public Base {
public:
    void show() const override {
```

```
        std::cout << "Derived 클래스" << std::endl;
    }
};

int main() {
    Derived d;
    Base* basePtr = &d;

    Base* movedPtr = std::move(basePtr); // 단순히 포인터를 이동
    movedPtr->show(); // 실제 객체는 이동되지 않음, 여전히 d를 가리킴

    return 0;
}
```

- **설명**: `std::move(basePtr)`는 단순히 포인터를 `rvalue`로 변환할 뿐, 실제로 `Derived` 객체의 이동은 일어나지 않습니다. `basePtr`과 `movedPtr`은 여전히 같은 객체를 가리킵니다.