Item 41)Consider pass by value for copyable parameters that are cheap to move and always copied.

몇몇의 함수는 copy를 염두하고 작성된다

`addName` 이라는 함수가 있다고 가정하자, 이 함수는 parameter를 private container에 복사하는 함수다.

```
class Widget {
public:
    void addName(const std::string& newName) // take lvalue;
    {
        names.push_back(newName); // copy it
    }

    void addName(std::string&& newName) // take rvalue;
    {
        names.push_back(std::move(newName)); // move it; see
        // ... // Item 25 for use of std::move
    }

private:
    std::vector<std::string> names;
};
```

위의 함수의 작동에는 문제가 없다.

그러나 몇가지 문제점이 있다.

1. **중복성** : 두 함수는 기본적으로 비슷한 작업을 수행한다. 하나는 lvalue 참조를, 다른 하나는 rvalue 참조를 받아들인다. 이런 중복성은 더 많은 코드를 작성하고, 관리를 해야하는 문제가 있다.
2. **객체 코드에 미치는 영향** : 동일한 기능을 하는 두 개의 별도 함수가 존재함으로써 프로그램의 크기가 증가할 수 있다. 두 함수가 컴판일러에 의해 인라인 처리될 수 있지만, 인라인 처리되지 않는 경우에는 실제로 두 개의 서로 다른 함수가 객체 코드에 존재하게 된다.

위의 문제를 해결하기 위한 방법은 `universal reference`를 사용하는 것이다.

```
class Widget{
public:
        template<typename T>
        void addName(T&& newName)
        {
                names.push_back(std::forward<T>(newName));
        }

}
```

위의 방식은 관리해야 할 소스 코드의 내용을 줄일 수 있다는 장점이 있다.

그러나 `universal reference`를 사용하는 것은 다른 문제를 야기한다.

template인 `addName`의 구현은 대부분 header file안에 있을 것이다. 이는 몇몇의 함수의 목적 코드를 만들 것이다. 즉, 하나의 템플릿 함수가 여러 타입에 대해 인스턴스화되면, 최종적으로 객체 코드에 여러 함수가 생성된다. 이는 프로그램의 크기와 성능에 영향을 줄 수 있다.

또한 `universal reference`를 사용하는 경우, 잘못된 인자 타입이 전달 되면 컴파일러의 오류 메시지가 복잡해져 이해하기 어려울 수 있다.

이런 문제를 해결하기 위해, 매개변수를 `값`으로 전달해야한다. (매개변수를 값으로 전달하라는 것은 좋지 않다고 했지만), 이런 경우 lvalue는 자동으로 복사되고, rvalue는 이동된다. 이로 인해 코드는 단순해지고, 중복된 함수가 필요 없어진다.

예시코드

```
class Widget{
public:
        void addName(std::string newName)
        {
                names.push_back(std::move(newName));
        }
}
```

`newName`이라는 함수는 매개변수로 전달된 `std::string`객체를 값으로 받는다. 따라서 lvalue와 rvalue모두 처리할 수 있다.

함수 내부에서는 `std::move`를 사용하여 `newName`을 이동(move)한다. 이는 `newName`이 복사된 독립적인 객체라는 점에서 안전하다.

이런 접근 방식은 `universal reference`를 사용하지 않으므로 `universal reference`의 단점을 고려하지 않아도 된다.

Isn't that expensive??

C++98에서는 복사 비용에 대한 우려가 합리적이다. 왜냐면 parameter에 전달된 값을 C++98에서는 `copy constructor`를 호출하기 때문이다.

그러나 C++11에서는 lvalue에서만 `copy constructor`를 호출할 것이며, rvalue에서는 `move constructor`를 호출할 것이다.

```
std::string name("bart");
w.addName(name); // lvalue를 사용한 호출

w.addName(name + "jenne"); // rvalue를 사용한 호출
```