## **Rvalue(우측값) 레퍼런스에 관해**

우측값 참조가 왜 필요한가

1. [move](https://modoocode.com/301) 의 구현 (move semantics)
2. 완벽한 전달(perfect forwarding)

C에서의 정의

좌측값은 대입(assignment) 시에 왼쪽 혹은 오른쪽에 오는 식(expression)이고,

우측값은 대입 시에 오직 오른쪽에만 오는 식이다

**Cpp에서의 정의**

좌측값은 어떠한 메모리 위치를 가리키는데, & 연산자를 통해 그 위치를 참조할 수 있다. 우측값은 좌측값이 아닌 값들이다

**우측값**

int foobar(); // int 를 리턴하는 함수  
int j = 0;  
j = foobar(); // ok. foobar() 는 우측값이다  
int\* p2 = &foobar(); // error. 우측값의 주소는 참조할 수 없다.  
j = 42; // 42 는 우측값이다.

**좌측값**

int i = 42;  
i = 43; // ok, i 는 좌측값  
int\* p = &i; //&i 를 쓸 수 있다.  
int& foo(); // int& 을 리턴하는 함수  
foo() = 42; // ok, foo() 는 좌측값  
int\* p1 = &foo(); // ok, &foo() 를 할 수 있다.

**이동대입연산**

X foo();

X x1;

x1= foo();

Foo()가 반환한 X클래스 임시객체를 x1에 복사대입을 하면

임시 객체가 가진 리소스(동적할당된) 를 복사해서 새로 생성하고 포인터로 가리키게한 후

임시객체가 소멸하며 소멸자가 임시객체와 가지고있는 동적할당 리소스를 해제한다.

그런데 임시객체가 가진 리소스를 그대로 x1이 가리키게 하면 더 빠르다.

이를위해 복사대입연산자를 오버로딩한다

X& X::operator=(const X& rhs)와

X& X::operator=(X&& rhs) 를 오버로딩하는 것이다.

즉 대입하는 객체가 임시객체(우측값)이냐 그냥객체(좌측값)이냐에 따라 다른함수를 작동시키는 것이다.

우측값레퍼런스 X&&는 좌측값 레퍼런스와 거의 유사하게 사용할 수 있지만, 특징은 좌측값은 좌측값레퍼런스를, 우측값은 우측값레퍼런스를 우선해서 매칭된다는 것이다.

즉 좌측값은 복사대입연산자로 우측값은 이동대입연산자로 보내버릴 수 있다.

대다수의 경우 move연산을 위해 우측값참조를 받는경우 복사생성자와 대입연산자밖에 없다.

void foo(X&); //좌측값만 받음

void foo(X const &);/

const T&에 한해서만 우측값도 레퍼런스로 받을 수 있음

그런데 문제는 const로 받기 때문에 임시객체의 동적할당리소스를 해제하는 것이 불가능.

Void foo(X&&)

//이동생성자 우측값(소멸되는 임시객체이거나, move로 강제로 우측값으로 만든 경우)

위의 두함수 중 하나라도 정의해야 좌측값도 양쪽 다 처리가 가능함.

Perfect fowarding

template <typename T, typename Arg>  
shared\_ptr<T> factory(Arg arg) {  
 return shared\_ptr<T>(new T(arg));  
}

위의 경우 arg가 복사되어 shared\_ptr에 전달된다.

마치 factory함수가 없는 것처럼 arg가 shared\_ptr<T>에 전달되면 좋겠을 때

template <typename T, typename Arg>  
shared\_ptr<T> factory(Arg& arg) {  
 return shared\_ptr<T>(new T(arg));  
}

레퍼런스를 받으면 또 우측값을 못받음.

그러면

template <typename T, typename Arg>  
shared\_ptr<T> factory(Arg const& arg) {  
 return shared\_ptr<T>(new T(arg));  
}  
이러면 또 const를 인자로 받으니 factory인자가 한개가 아니면 const와 const가 아닌 인자들 모든 조합에 대해 오버로딩을 해야함.

뿐만 아니라 move연산도 못함 T의 복사생성자에 전달되는 인자는 Arg const& (좌측값 상수레퍼런스 = 좌측값)이므로 우측값을 넣어줘도 좌측값이되어 전달되고, 이동생성자(우측값을 받음)가 실행이 안됨.

&겹침 규칙

& & -> &

& && -> &

&& & -> &

&& && -> &&

& : 1

&&:0

으로 보고 or연산 한다고 생각하면 쉽다.

특수 템플릿 인자 유추 규칙(special template argument deduction rule)

template <typename T>  
void foo(T&&);

만일 foo가 A의 좌측값으로 호출된다면

A a; //좌측값.

A bar();//우측값리턴

Foo(a) -> foo(A& &&) ->foo(A&)

Foo(bar()) ->foo(A&& &&) -> foo(A&&)

template <typename T, typename Arg>  
shared\_ptr<T> factory(Arg&& arg) {  
 return shared\_ptr<T>(new T(std::forward<Arg>(arg)));  
}

이러면 좌측값이면 좌측값레퍼런스, 우측값이면 우측값레퍼런스가 forward에 전달된다.

문제는 좌측값레퍼런스든 우측값레퍼런스든 좌측값이라는 것.

그래서 forward함수를 거쳐서 우측값레퍼런스는 우측값으로 캐스팅해줘야한다.

Forward함수는 좌측값레퍼런스일때는 냅두고 우측값일때만 우측값으로 바꿔준다.

std::remove\_reference 는 타입의 레퍼런스를 지워주는 템플릿 메타 함수

Forward 함수

template <class S>  
S&& forward(typename remove\_reference<S>::type& a) noexcept {  
 return static\_cast<S&&>(a);  
}

좌측값들어왔을때 arg는 X& 타입이 되고->

X& && forward(remove\_reference<X&>::type& a) noexcept{

Return static\_cast<X& &&>(a);

}

->

X& forward(remove\_reference<X&>::type& a) noexcept{

Return static\_cast<X&>(a);

} 이때 forward는 좌측값레퍼런스를 그대로 리턴하게되므로 좌측값이 그대로 복사생성자에 전달됨.

우측값이들어왔을때는 arg는 X&& 타입이 되고->

X&& && forward(remove\_reference<X&&>::type& a) noexcept{

Return static\_cast<X&& &&>(a);

}

->

X&& forward(X& a) noexcept{

Return static\_cast<X&&>(a);

}

우측값이 캐스팅되어 리턴된다.