Effective Modern C++ Study

Item 1 ~ Item 6

진행: 최규진

230219

목차

1. 템플릿 형식 연역

템플릿의 형식 연역은 c++-98부터 존재.
modern c++ (c++-11이후) 에선 auto가 추가
auto와 템플릿의 형식 연역은 비슷.
따라서 템플릿의 형식 연역을 복습하는 Item이다.

책에선 아래의 표현이 많이 나오므로 숙지하고 있으면 좋다.

```
template<typename T>
void f(ParamType param);
f(expr);
```

T는 템플릿의 타입이고
ParamType 은 T를 통해 나올 수 있는 모든 종류의 타입 (const T&, T, T&, ..)
expr은 함수에 넘겨주는 인자. f(x), f(*x), f(std::move(x)), etc..

1. 템플릿 형식 연역

다음을 보자.

```
template<typename T>
void f(const T& param);

int x = 0;
f(x);
```

T : **int**

ParamType = const int&

expr: x

x가 int 타입일 때,

T는 int로 연역되고, ParamType은 이에 따라 const int&로 연역된다.

다만 모든 경우에 이렇게 T와 ParamType이 매칭되는건 아니다.

1. 템플릿 형식 연역

ParamType의 3가지의 경우에 따라 템플릿의 연역 규칙이 변경된다.
즉, T랑 ParamType모두, ParamType에 뭘 넣느냐에 따라 달라진다.

1. ParamType이 포인터 or 참조
2. ParamType이 보편 참조 (이따 설명)
3. ParamType이 포인터도 참조도 아닌 경우. (위의 예시같은 왼값)

다음을 다시 기억하면서 하나씩 살펴봅시다.

```
template<typename T>
void f(ParamType param);
f(expr);
```

1. 템플릿 형식 연역 - 경우 1. ParamType이 참조 or 포인터

아래와 같은 경우라고 볼 수 있다.

```
template<typename T>
void f(T& param);
f(expr);
```

이 경우, expr이 참조 형식이라면 참조 부분을 무시한다.
그 후, expr의 형식을 ParamType에 대해 패턴 부합 방식으로 대응시키고,
T가 결정이된다.

이 부분은 이후 예제를 보며 파악해보자.

1. 템플릿 형식 연역 - 경우 1. ParamType이 참조 or 포인터

```
template<typename T>
void f(T& param);

int x = 27;
f(x);
```

x가 참조형식인가? X

x는 ParamType (T&) 로 어떻게 연역되는 지와 동시에 T가 정해진다.

이 경우는 ParamType이 int&로 연역되면서 T가 int로 정해진다.

1. 템플릿 형식 연역 - 경우 1. ParamType이 참조 or 포인터

```
template<typename T>
void f(T& param);

int x = 27;
const int cx = x;
f(cx);
```

cx가 참조형식인가? X

cx가 ParamType(T&)로 어떻게 연역되는 지와 동시에 T가 정해진다.

cx타입은 const int이고, 형식 연역은 const까지 고려하여 연역을 진행한다.

따라서 ParamType은 const int& 가 되며 (T&), 이와 동시에 추론된 T는 const int라고 볼 수 있다.

1. 템플릿 형식 연역 - 경우 1. ParamType이 참조 or 포인터

```
template<typename T>
void f(T& param);

int x = 27;
const int cx = x;
const int& rx = cx;
f(rx);
```

rx는 참조 형식인가? O
따라서 이는 무시된다. T& 에서 const int&&가 될 일은 없단 뜻이다.
참조가 무시되기 때문에 이전과 마찬가지로
ParamType은 const int&, T는 const int로 연역된다.

1. 템플릿 형식 연역 - 경우 1. ParamType이 참조 or 포인터

```
template<typename T>
void f(const T& param);

int x = 27;
const int cx = x;
const int& rx = cx;
f(x); f(cx); f(rx);
```

cx와 rx의 const성은 계속 유지가 되고,
x의 경우 ParamType이 const int&로 추론된다.
따라서 세 경우 모두 T는 int, ParamType은 const int&가 된다.

T가 int인 이유는 T에 대한 추론이 ParamType을 통해 이루어지는데,
패턴 부합 형태로 const "int"& 로 T가 정해졌기 때문에 T가 int이다.

1. 템플릿 형식 연역 - 경우 1. ParamType이 참조 or 포인터

```
template<typename T>
void f(T* param);

int x = 27;
const int* cx = x;
f(&x); f(cx);
```

&x의 타입은 (int*) 이기 때문에 ParamType은 int*, 이에 따라 T는 int로 연역된다.

cx는 애초에 포인터 타입 (const int*)이기 때문에, T는 const int*로 연역될 수 없다.
ParamType 자체에 *가 달려있기 때문이다.
따라서 이 경우는 T가 const int로 연역된다.

1. 템플릿 형식 연역 - 경우 2. ParamType이 보편 참조

보편참조를 알기 위해 기억해야되는 왼값과 오른값을 먼저 간단하게 보자.
그 후 보편참조가 뭔질 간단하게 보자.
여기선 정말 간단하게만 설명하고, 참조된 링크를 통해 추가로 공부하면 좋다.

왼값과 오른값

> "참고 링크"
> 왼값과 오른값
> 오른값 참조?

왼값은 메모리상에 실체가 있어서, 표현식 이후에도 살아있는 친구고,
오른값은 메모리상에 데이터가 있더라도, 표현식 이후에는 알아서 사라져주는 친구다.

```
int x = 30; // 30은 오른값. 메모리나 레지스터에 값은 있겠지만 이 표현식 이후 사라짐.
int x2 = x; // x는 왼값. 표현식 이후에도 x라는 공간자체가 유효하다.
```

1. 템플릿 형식 연역 - 경우 2. ParamType이 보편 참조

왼값과 오른값

```
int func(int x);
func(30); // 리턴값이 오른값. 이 statement 이후에 증발함.
```

오른값 참조?

오른값이 증발하는건 맞는데, 증발하기 전에 이 값을 참조해서 어딘가에 쓰고싶으면 쓰기 좋다.
함수의 리턴값을 추가 할당 없이 옮긴다거나, 날아가버릴 상수값등을 함수의 인자로 넘길 때 좋다.

```
// 자세한건 rvalue_ref.cc 를 확인하자.
int f1(int &x) {
    A = x;
}
f2(40);
f2(ret());
```

1. 템플릿 형식 연역 - 경우 2. ParamType이 보편 참조

보편 참조?

보편 참조는 template이나 auto에서 쓰이는데,

```
// 1. Template
template<typename T>
void f(T&& param);

// 2. auto
auto&& value = some_function();
```

위 두 경우 처럼 들어오는 인자나 함수의 리턴이 lv인지 rv인지 확실하게 명시할 수는 없지만, 참조는 쓰고싶을 때 보편참조를 쓸 수 있다.

1. 템플릿 형식 연역 - 경우 2. ParamType이 보편 참조

아래의 형식을 기억합시다.

```
template<typename T>
void f(T&& param);
f(300); // rvalue
int x = 300;
f(x); // lvalue
```

앞서 말했듯, 템플릿에서 위와같이 사용하는 것은 보편참조를 의미한다.
expr이 왼값이냐 오른값이냐에 따라 ParamType(T&&)과 T가 다르게 연역될 것이다.

expr이 오른값이면 사실 아무 문제 없다.
T&&은 300이 오른값임을 보고 컴파일러는 ParamType(int&&)로 연역을 할 것이다. 이에 따라 T도 int로 연역된다.

expr이 왼값이면 좀 많이 이상해진다.
왼값 참조에 &&를 때릴 순 없어서 ParamType(int&)로 연역이 되고,
T는 int&로 연역된다. T가 참조형식이 되는 유일한 경우라고 한다.

1. 템플릿 형식 연역 - 경우 2. ParamType이 보편 참조

정리

```
template<typename T>
void f(T&& param);
int x = 27;
const int cx = x;
const int& rx = cx;

f(x); // ParamType(T&& => int&, T => int&)
f(cx); // ParamType(T&& => const int&, T => const int&)
f(rx); // ParamType(T&& => const int&, T => const int&)
f(27); // ParamType(T&& => int&&, T => int)
```

1. 템플릿 형식 연역 - 경우 3. ParamType이 참조도 포인터도 아님.

```
template<typename T>
void f(T param);
f(x)
```

같은 형태. 그냥 pass by value라고 생각하면 된다.
이 때는 경우 1과 마찬가지로 expr이 참조면 무시된다. 참조가 무시된 이후 const가 붙었다면 const도 무시된다. (volatile도 무시)

결국 아래와 같이 정리하면 되며, 우리가 const를 넘겨줘도 pass-by-value에 의해 값복사가 일어나서, 복사된 객체에 대한 수정은 허용된다.

```
int x = 30;
const int cx = x;
const int& rx = x;
f(x); // ParamType(int), T int로 연역.
f(cx); // ParamType(int), T int로 연역.
f(rx); // ParamType(int), T int로 연역.
```

1. 템플릿 형식 연역 - 경우 3. ParamType이 참조도 포인터도 아님.

아래의 예를 통해 경우 3을 좀 더 확실히 알 수 있다.

```
template<typename T>
void f(T param);

// const (char *) ptr 은 char *인 ptr이 가리키는 데이터가 const라는 의미이다.
// const (char *) const ptr은 데이터 뿐만아니라 가리키는 주체인 ptr도 const라는 의미다.
// 따라서 ptr = {sum_value} 같은 대입연산이 불허된다.
const char* const ptr = "hihi";
f(ptr);
```

이 경우 const가 두 개라 헷갈리는데,
const 포인터가 넘어간거고 const char*는 상관이 없다.

따라서 불변 데이터 "hihi"의 주소를 가지는 ptr가 하나 복사된 것이고,
이 복사된 포인터에 대한 값변경 (메모리 주소 변경)은 가능하다.
즉 f 내부에선 ptr = null을 해도 상관없다.

1. 템플릿 형식 연역 - 배열 인수

배열과 포인터의 Decay

c++은 c에 뿌리를 두고 있기 때문에 다음 두 구문이 달라보이더라도 실제 넘어가는 주소 등은 모두 똑같이 사용된다.

이로 인해 혼동이 발생하고 이를 decay라고 책에선 표현하고 있다.

```
// 아래 두 경우는 사실상 동치이다.
// 애초에 배열 형식의 매개변수라는건 존재하지 않는다고 한다 (표현 자체는 혀용해준다.)
void myFunc(int param[]);
void myFunc(int *param);
```

실제로 배열의 대한 template 연역도 모두 *로 연역된다.

```
template<typename T>
void f(T param);
int arr[30] = {0,};

f(arr); // T = int* 로 연역
```

1. 템플릿 형식 연역 - 배열 인수

배열 자체의 참조형식 인수는 허용해준다.

```
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) nonexcept {
  return N;
}

int keyVals[] = {1,3,5,7,9,11} // 원소 6개

// keyVals의 타입은 int[]이고, 이 때 ParamType은 int (&) [6] 으로 연역된다.
// T는 int로.연역될 것이고, N에는 6이 들어갈 것이다 (컴파일 타임에 아는 값이라).
// 따라서 이 값을 리턴해주는 함수를 만들면
// static array의 크기를 반환하는 함수를 짤 수 있다.
arraySize(keyVals);
```

1. 템플릿 형식 연역 - 함수 인수

책에도 짧게만 나오고 크게 문제될 부분은 없다고 해서 코드만 짧게 보자.

```
void someFunc(int, double); // 함수 정의.

template<typename T>
void f1(T param); // 경우 3.

template<typename T>
void f2(T& param); // 경우 1.

// 기본 pass-by-value지만 함수 포인터로 연역됨.
// 이게 자연스럽긴 하다.
// void (*) (int, doubule) 로 T와 ParamType이 연역된다.;
f1(someFunc);

// 얘는 void (&) (int, doubule) 로 ParamType이 연역되는 것 같다.
// T는 어떻게 연역되는진 안나온다.
f2(someFunc);
```

2. auto 형식 연역

경우 1개 빼곤 일단 템플릿이랑 다 똑같다.
일단 조금 익숙해질겸 항목 1과의 대응관계를 보자.

```
template<typename T> // case 3
void f(T param);
f(x);
auto x;

template<typename T> // case 3
void f(const T param);
f(cx);
const auto cx;

template<typename T> // case 1
void f(const T& param);
f(rx);
const auto& rx;

template<typename T> // case 2
void f(T&& param);
f(rx);
auto&& rx;
```

2. auto 형식 연역

```
auto x = 27; // 경우 3. x는 int로 연역, 과 같다. 걍 pass by value
const auto cx = x; // 경우 3. pass by value. auto는 int
const auto& rx = cx; // 경우 1. auto(T)는 int로 연역된다.
auto&& uref1 = x; // 경우 2. 왼값 참조. auto&&(ParamType)는 int&로 연역
auto&& uref2 = cx; // 경우 2. 마찬가지로 왼값 참조. auto&&(ParamType)은 const int&로.
auto&& uref3 = 27; // 경우 2. 오른값 참조. auto&&(ParamType)은 int&&가 된다.

// 배열 및 함수 케이스
const char name[] = "hihi";
auto arr1 = name; // arr의 타입은 const char* (decay되는 케이스)
auto& arr2 = name; // const char(&)[5];

void sumefunc(int, double);
auto func1 = somefunc; // void * 함수포인터
auto& func2 = somefunc; // void& 참조함수.
```

2. auto 형식 연역

초기화 리스트에 대한 연역은 예외케이스이다.

일단 초기화 리스트 예제를 봐보자.

```
vector<int> vc{3,2,1,2,3}; // 초기화 리스트를 이용한 초기화. c의 배열초기화 같은 모습
// c++11 이후 아래의 초기화도 가능
int a{3};
int b={3};

// 객체에 대한 초기화 리스트
// 이 경우엔 std::initializer_list<int> 혹은 <T> 등을 인자로 하는 초기화 리스트에 대한
// 생성자가 클래스 내에 정의되어 있어야 쓸 수 있다.
UserClass c1 {10, 10, 10, 10, 10};
```

2. auto 형식 연역

```
auto a1 = {27};
auto a2{27};
auto a3{27, 0.5}; // Compile error
```

일단 {}를 이용한 초기화에서 {}들의 리턴 타입은 모두 std::initializer_list<T> 이다.
따라서 auto 입장에선 int같은 타입으로 보게되는게 아니라 모두 std::initializer_list<T> 타입으로 바라보게 된다.

이에 따라 auto에 대한 연역도 std::initializer_list<T>로 이루어진다.

추가로 초기화 리스트는 템플릿으로 정의되어 있고,
해당 템플릿의 타입은 1개로 고정되어있기 때문에 세번째와 같이 쓰면 컴파일에러다.

2. auto 형식 연역

템플릿은 이런 연역 자체가 안된다.

```
template<typename T>
void f1(T param);

template<typename T>
void f2(std::initializer_list<T> param);

f1({11,23,9}); // Error
f2({11,23,9}); // OK
```

템플릿에선 초기화 리스트에 대한 연역 자체가 불가능하다.
이중 연역? 에 대해서 auto는 허용하는데 템플릿은 안하는 것 같기도 하다.

ParamType에 초기화 리스트임을 명시해주는 f2는 연역이 1번만 들어가서 그런가 된다.
저자도 안되는 이유는 모른댄다.

3. decltype의 작동 방식

decltype(expr)은 expr에 대해 연역된 타입을 표현한다.
아래 예들을 보자.

```
const int i = 0; // decltype(i) => const int
bool f(const Widget& w);  // decltype(f) => bool(const Widget&)
                          // decltype(w) => const Widget&
struct Point {
  int x, y; // decltype(Point::x) => int
};
Widget w; // decltype(w) => Widget
if (f(w)) .. // decltype(f(w)) => bool
vector<int> v; // decltype(v) => vector<int>
if (v[0] == 0) .. // decltype(v[0]) => int& , vector<int> operator[]가 int&를 반환함.
```

3. decltype의 작동 방식 - 함수의 반환형식

앞선 예시들은 변수의 타입과 동일한 타입을 리턴해주는데 사용된다.
decltype은 함수의 반환방식을 변경하는 데에도 사용된다.

c++11에선 아래와 같은 문법이 가능하다.

```
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i)
-> decltype(c[i]) {
  authenticateUser(); // 하는건 없는 것 같다.
  return c[i];
}
```

c++11에서 이와 같은 형태의 auto는 기존 형식연역으로써의 의미는 가지지 않는다.
"후행 반환 형식" 이라는 구문이 쓰인 것으로, 이 함수의 반환 형식은 이 위치가 아니라 매개변수 목록 다음에 -> 형태로 나온다는 의미로 쓰인다.

이와 같은 문법이 존재하는 이유는 c도 i도 무엇인지 모르는 상태에서 c와 i와 관련된 리턴을 하고 싶을 때 사용하기 위함이다.

3. decltype의 작동 방식 - 함수의 반환형식

c++ 14에선 이와 같은 한계를 적당히 극복하고, 함수 리턴값을 auto로 주면 실제 형식 연역을 일으켜준다. (앞선 c++11에선 decltype으로 사실상 우리가 지정해 준다.)

```
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i) { // decltype이 없음.
  authenticateUser(); // 하는건 없는 것 같다.
  return c[i];
}
```

이 경우 c[i]의 타입에 따라 auto가 연역된다. 연역 규칙은 템플릿이 연역되는 규칙과 동일하게 작동한다.

c[i]의 리턴 타입은 보통 operator[]에 의해 Container&가 될 것이다.

```
auto return_value = c[i];
```

위와 동일한 형태라고 보면 좋은데, c[i]가 Container&를 줘도 auto의 연역에선 경우1, 경우3 모두 참조성에 대해서는 무시가 된다. 즉 Container로 타입을 연역하게된다.

3. decltype의 작동 방식 - 함수의 반환형식

다시 이 예제를 보자.

```
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i) { // decltype이 없음.
  authenticateUser(); // 하는건 없는 것 같다.
  return c[i];
}
```

리턴 값이 Container로 연역이 된다면 함수를 사용하는 실제 사례에서 문제가 되는 경우가 발생한다.

```
std::deque<int> d;
authAndAccess(d,5) = 10; // authAndAccess의 리턴은 Container. 이 값은 '우측값'이다.
                         // 우측값에 대한 대입은 c++에서 금지되어 있다.
```

3. decltype의 작동 방식 - 함수의 반환형식

decltype을 함수 반환 형식에 사용하게되면 연역하려는 변수와 완전히 동일한 리턴값을 가지게 만들 수 있다. c++ 14를 만든 장인들이 이런 문제를 인식하고 반영해 놨다.

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i) {
  authenticateUser(); // 하는건 없는 것 같다.
  return c[i];
} // c[i]와 완전히 같은 형식이 리턴된다. 즉 Container&가 제대로 리턴이 된다.
```

이는 대입연산을 사용할 때도 유용하게 쓸 수 있다.

```
Widget w;
const Widget& cw = w;
auto myWidget1 = cw; // myWidget1은 Widget으로 연역됨. 참조성, const성 모두 제거.
decltype(auto) myWidget2 = cw; // const Widget&로 연역됨.
}
```

3. decltype의 작동 방식 - 함수의 반환형식

앞선 예제들은 모두 왼값참조가 되는 컨테이너에 대한 예제였다.
따라서 아래와 같은 예시들은 동작을 하지 못한다.

```
std::deque<std::string> makeStringDeque();
auto s = authAndAccess(makeStringDeque(), 5);
```

따라서 보편참조를 사용한 형태로 변경하여 쓸 수 있다.

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container&& c, Index i) {
  authenticateUser(); // 하는건 없는 것 같다.
  return std::forward<Container>(c)[i];
}

//c++11
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container&& c, Index i)
 -> decltype(std::forward<Container>(c)[i]) {
  authenticateUser(); // 하는건 없는 것 같다.
  return std::forward<Container>(c)[i];
}
```

3. decltype의 작동 방식 - 함수의 반환형식

std::move<> ? std::forward<> ?

> 참고링크
> std::move, std::forward

링크를 들어가서 보는게 제일 좋은데,
std::move는

```
return static_cast<remove_reference_t<Person>&&>(_Arg);
return static_cast<remove_reference_t<Person &>&&>(_Arg);
```

와 같이 lvalue던 rvalue던 참조를 제거하고 우측값참조로 static_cast로 형변환을 시켜버린뒤 리턴을 한다.

3. decltype의 작동 방식 - 함수의 반환형식

std::move<> ? std::forward<> ?

std::forward는 lvalue참조면 lvalue를 리턴해주고, rvalue참조면 rvalue참조를 리턴해준다.
rvalue를 파라미터로 넘겨줘도 매개변수를 받는쪽에선 lvalue이기 때문에 쓴다.

```
atemplate<typename T>
void ForwardingObj(T&& obj, const char* name) {
    // 누군가 FowrwardingObj(std::move(obj), "name");으로 rvalue를 넘겨줘도.
    // 함수 내에선 obj라는 이름의 객체가 lvalue로 해당 값을 이동대입해서 가지고 있다.
    // 현재 scope에선 obj가 rvalue참조타입의 lvalue인거다. 따라서 이동대입을 또시켜주려면
    // rvalue 참조가 될 수 있게 명시적으로 한 번 더 조져야한다.
    Catch(std::forward<T>(obj), name);
}
```

3. decltype의 작동 방식 - 함수의 반환형식

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container&& c, Index i)
 -> decltype(std::forward<Container>(c)[i]) {
  authenticateUser(); // 하는건 없는 것 같다.
  return std::forward<Container>(c)[i];
}
```

따라서 이 예시에서 operator[]에 우측값 참조로 넣어주기 위해 std::forward를 명시적으로 한 번 더 선언해 주는것이다. 이렇게 해야

```
std::deque<std::string> makeStringDeque();
auto s = authAndAccess(makeStringDeque(), 5);
```

가 동작한다.

3. decltype의 작동 방식 - lvalue와 lvalue표현식에서의 차이

decltype은 lvalue에선 그 타입을 리턴해주는데,
lvalue 표현식에선 해당 타입의 참조를 리턴하도록 되어있다.

```
decltype(auto) f1() {
  int x = 0;
  return x; // decltype(x)는 int로 연역되고 f1은 int를 반환
}

decltype(auto) f2() {
  int x = 0;
  return (x); // decltype(x)는 int&로 연역되고 f2은 int&를 반환
}
```

4. 추론된 형식을 어떻게 파악하나?

내용이 길긴한데, IDE를 쓰면 마우스 갖다 대거나 하는 동작이나 디버거로 타입을 바로바로 알 수 있고, printf를 하고싶으면 c++ 내장기능을 사용하면 가능하다는 챕터인데,

OS 벤더 별로 조금씩 차이가 있기도 하고 신뢰할 수 있는 결과를 주지 않는다는 내용이다.

제일 좋은 방법은 boost라이브러리에 있는 함수를 쓰라는게 결론이다.

5. 명시적인 형식의 선언보단 auto가 좋다.

auto의 장점은 글자 수를 줄이는 것을 넘어서 정확성 문제, 성능 문제 등을 방지할 수 있다.

```
template<typename It>
void dwim(It b, It e) {
  for (; b!=e; b++) {
    // typename bad
    std::iterator_traits<it>::value_type currValue = *b;
    // typename good
    auto currValue = *b;
  }
}
```

```
int x; // 쓰레기값이 있음
auto x; // 컴파일에러로 잡아줌
auto x = 33; // 올바른 예
```

5. 명시적인 형식의 선언보단 auto가 좋다.

람다함수에서도 auto가 유용하다.

> **람다함수 관련 링크**

```
auto derefUPLess = [] (  // c++11, 람다함수 자체를 auto로 감싼다.
  const std::unique_ptr<Widget>& p1,
  const std::unique_ptr<Widget>& p2)
  { return *p1 < *p2; };

auto derefUPLess = [] (  // c++14, 람다함수 + 매개변수도 auto로 감싼다.
  auto& p1, // 타입에 대한 연역이 일어나 주기 때문에,
  auto& p2) // 앞의 예처럼 타입 지정을 해줄 필요가 없다.
  { return *p1 < *p2; };
```

5. 명시적인 형식의 선언보단 auto가 좋다.

auto를 안쓰고 std::function으로 함수를 저장하면 어떻게 되나?

```
std::function<bool(
  const std::unique_ptr<Widget>&,
  const std::unique_ptr<Widget>&
)> derefUPLess = [] (
  const std::unique_ptr<Widget>& p1,
  const std::unique_ptr<Widget>& p2)
  { return *p1 < *p2; };
```

일단 이름이 긴것도 문젠데

std::function로 사용하면 std::function의 기본 인스턴스(객체 변수) 크기가 derefUPLess를 저장하는 데에 부족할 수도 있다. 이 경우 힙메모리를 추가로 할당해서 저장하는데 이렇게되면 결과적으로 auto보다 많은 메모리를 사용한다.

auto는 클로저가 반환되는걸 보고 해당 타입을 저장할 수 있는 크기만큼만 딱 사용한다.

> **클로저?**
> 람다 표현식의 리턴값이 클로저다.
> auto f = {람다함수} 에서 {람다함수}가 클로저고 auto f는 클로저를 담는 놈일 뿐이다.

5. 명시적인 형식의 선언보단 auto가 좋다.

```
std::vector<int> v;
unsigned sz = v.size(); // 32비트 64비트 컴퓨터에 따라 저장 형식이 변경
auto sz = v.size(); // 알아서 맞춰줌
```

```
std::unordered_map<std::string, int> m;
for (const std::pair<std::string, int> &p : m) {} // bad
for (const auto &p : m) {}
```

c++ std인 unordered_map의 key는 const로 저장된다.
따라서 위와 같이 쓰면 key값에 대해

```
const std::pair<const std::string, int> &p
```

로 써야된다. 따라서 타입이 맞지가 않게되고 이로 인해 어떻게든 맞춰주기 위해서
const std::pair<std::string, int> 타입의 임시 객체를 만들고 해당 임시 객체에 대한 참조값을 p에 저장하는 이상한 코드가 된다.

6. auto가 원하지 않는 형식으로 연역이 된다면?

책 앞부분에 잠깐 나오는데 표준 컨테이너 (vector, deque, ..) 등의 bool 타입에 대한 operator[] 함수는 bool&를 리턴하지 않는다라는 얘기가 있다.

따라서 아래 코드는 다르게 동작한다.

```
std::vector<bool> feature(const Widget& w);
bool highPriority = feature(w)[5]; // bool로 받음
auto highPriority = feature(w)[5]; // std::vector<bool>::reference로 받음
```

bool이 조금 특이해서 그런데, 1비트만 필요한 bool은 표준 컨테이너 상에서 1bit의 압축된 형태로 표현되어있다.

이로 인해 bool&이 필요한 시점에서 동작을 비슷하게 만들어 주기 위해 참조를 위한 임시 객체를 만드는데 그게 std::vector<bool>::reference 라고 보면 된다. 이런 클래스들을 대리자 클래스라고도 한다.

문제는 auto의 타입이 해당 임시객체 타입이고, operator[]를 어떻게 구현해놨는 지 알수가 없다. 이로인해 문장이 끝나면 사라질 임시객체의 특정 오프셋을 가리키게 되고, 이는 undefined behavior를 불러일으킨다.

6. auto가 원하지 않는 형식으로 연역이 된다면?

auto 자체가 문제는 아니다. auto는 오히려 리턴값으로 온 대리자 클래스의 타입으로 잘 연역했다.

따라서 이는 우리가 잘 캐스팅만 해주면 끝나는 문제다.

> 업캐스팅, 다운캐스팅. dynamic_cast, static_cast

즉 위 예제는 아래와 같이 변경하면 큰 문제가 없다.

```
auto highPriority = static_cast<bool>(feature(w)[5]);
// 대리자 클래스가 bool타입으로 static_cast되고, 해당 타입을 auto가 연역.

auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

```
double ep = calcEpsilon(); // double을 리턴하던 함수.
float  ep = calcEpsilon(); // 가끔은 이걸 쓰고 싶을 때, (최적화?)
auto   ep = static_cast<float>calcEpsilon(); // 이게 더 명시적이며, 다른 개발자가
                                             // 의도를 파악하기 쉽다.
```