Item 5) Prefer auto to explicit type declaration

auto는 초기화가 되어야 한다

- `auto`로 선언된 변수들은 initializer에 의해서 타입 추론이 진행된다. 따라서 변수의 초기화가 필연적이다.

```
int x1; // uninitialized but ok!
auto x2; // error -> 초기화가 필요합니다.
auto x3 = 0;
```

auto와 std::function과의 비교

- `auto`는 타입 추론을 사용하기 때문에, 컴파일러만 알고 있는 타입도 나타낼 수 있다.

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

auto derefLess =
[](const auto& p1, const auto& p2)
{return *p1 < *p2;};
```

- 위의 코드 예시를 보면 타입을 명시하지 않고도 `lambda object`를 auto로 표현할 수 있다.

---

std::function과 function pointer에 관련해서

`std::function`은 C++11에서 제공하는 템플릿 클래스이다. 기존의 함수 포인터 개념을 확장한 것으로, 단순히 함수에만 포인터를 가리킬 수 있는 함수 포인터와 달리, `호출 가능한 모든 객체`를 참조할 수 있다. 즉, `std::function`은 함수 뿐만 아니라, 람다 표현식, 함수 객체등 "함수처럼 호출할 수 있는 모든 것"을 다룰 수 있다.

```
bool(const std::unique_ptr<Widget>&, const std::unique_ptr<Widget>&)
// C++11의 comparision function
```

```
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;};
```

그러나 `std::function`과 `auto`를 사용하는 것은 중요한 차이점이 있다.

⭐⭐⭐⭐⭐
일반적으로 `std::function`는 `auto` 객체보다 더 많은 메모리를 사용한다. 또한 모종의 이유로, `std::function` 객체는 `auto` 변수보다 속도가 더 느리다.
결론적으로 **std::function을 사용하는 것은 auto객체를 사용하는 것 보다 속도와 메모리적으로 더 느리다**
또한 **auto를 사용하는 것은 적어야 할 코드가 줄어든다는 장점이 있다**

---

type shortcut과 관련된 문제

```
std::vector<int> v;

unsigned sz = v.size();
```

`v.size()`의 원래 return값은 `std::vector<int>::size_type`이다. 그러나 이 타입은 `unsigned integral type`과 같은 의미이다.

32bit Windows시스템 커퓨터에서는 `unsigned`와 `std::vector<int>::size_type`은 같은 size를 가진다. 그러나 64bits Windows시스템 컴퓨터에서는 다른 size를 가진다. 그러므로 `auto`를 사용하는 것이 더 적합하다.

```
auto sz = v.size();
```

타입 불일치에 관련한 문제

`std::unordered_map`은 키-값 쌍을 저장하는 해시 테이블이다. 여기서 중요한 점은 **key가 항상 const**로 선언된다는 것이다.

```
std::unordered_map<std::string, int> m;
```

```
for(const std::pair<std::string, int>& p: m)
{
 // do something
}
```

위의 코드를 살펴본다면, m에 있는 각 요소를 참조하려는 의도로 작성되었다. 그러나 문제는 `std::unordered_map`은 `std::pair<const std::string, int>` 타입의 요소를 저장지만, p는 `std::pair<std::string,int>`로 선언되어 있다. 즉, 키가 `const`가 아니므로 타입 불일치가 발생하게 된다.

컴파일러는 타입 불일치를 해결하기 위해, `std::pair<std::string,int>`로 변환하려고 시도한다. 그 결과, 매 반복마다 `임시 객체`가 생성되고, `p`는 그 임시 객체에 바인딩 된다.

`auto`의 사용

`auto`를 사용하게 된다면 이러한 타입 불일치를 방지할 수 있다.

```
for(const auto& p: m)
{
        // do something
}
```