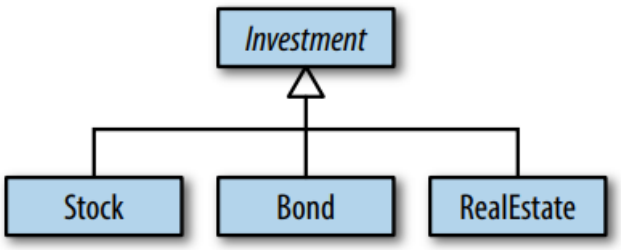
Item 18) Use unique_ptr for exclusive-ownership resource management.

`unique_ptr` 에 대하여

- header file : `#include <memory>`
- 일반적으로 `unique_ptr` 는 `raw pointer` 와 동일한 크기를 가진다. 즉, 메모리상에서 차지하는 공간이 같다.
- `unique_ptr` 를 사용하는 대부분의 연산(dereference)는 `raw pointer` 와 동일한 명령을 수행한다. 따라서 성능 차이도 거의 없다.
- `std::unique_ptr` 는 `exclusive ownership` 이란 개념을 함의한다.
- null이 아닌 `std::unique_ptr` 는 항상 포인팅 하는 객체를 소유한다.
- `unique_ptr` 에서 Moving은 객체의 소유권을 source pointer에서 destination pointer로 넘긴다(이후의 source pointer는 null이 된다).
- `unique_ptr` 에 Copy는 불가능하다. 왜냐하면 copy를 하게 된다면, `unique_ptr` 는 같은 자원을 pointing하기 때문이다. 따라서 `unique_ptr` 는 move-only type이다.
- destruction이 진행되는 경우, null이 아닌 `std::unique_ptr` 는 포인터가 가지고 있는 자원을 destroy한다.



위와 같은 계층을 가지고 있는 객체가 있다고 가정하자 (Base class : `Investment`)

```
class Investment {
    // Investment 클래스의 멤버와 메서드 정의
};
class Stock : public Investment {
    // Stock 클래스의 멤버와 메서드 정의
};
class Bond : public Investment {
    // Bond 클래스의 멤버와 메서드 정의
};
class RealEstate : public Investment {
    // RealEstate 클래스의 멤버와 메서드 정의
};
```

`unique_ptr` 는 객체에 대한 **독점 소유권(exclusive ownership)** 을 제공한다.
`unique_ptr` 가 소멸되면, 그 안에 있는 객체도 자동으로 소멸된다. 따라서 사용자는 더 이상 수동으로 `delete` 를 호출할 필요가 없다.
팩토리 함수에서 `unique_ptr` 를 반환하면, 함수 호출자는 이 포인터를 통해 객체의 소유권을 얻게 된다.
이 소유권은 **독점적**이므로, 호출자는 이 객체를 직접 관리할 필요가 없으며, `std::unique_ptr` 가 스코프를 벗어나거나 소멸될 때 자동으로 객체를 삭제한다.

코드 예시

```
template<typename... Ts>
std::unique_ptr<Investment>
makeInvestment(Ts&&... params);
```

- 위 함수는 가변인자를 받아서 `Investment` 객체를 생성하고, 이를 `std::unique_ptr<Investment>` 로 감싸서 반환한다.
- 이렇게 반환된 `std::unique_ptr<Investment>` 는 `Investment` 객체의 독점 소유권을 가지며, `unique_ptr` 가 소멸될 때, `Investment` 객체도 함께 소멸된다.

```
{
        auto pInvestment = makeInvestment(arguments);
        // pInvestment는 std::unique_ptr<Investment> 타입이며, 생성된 Investment 객체를 소유합니다. // pInvestment를 사용하여 Investment 객체에 접근하고 필요한 작업을 수행합니다.
}
```

- `makeInvestment(arguments);` 는 `std::unique_ptr<Investment>` 타입의 포인터를 반환하며, `pInvestment` 는 이 객체의 소유권을 가지게 된다.
- `pInvestment` 가 스코프를 벗어나거나 소멸될 때, `unique_ptr` 는 내부적으로 `delete` 를 호출하여 객체를 해제한다. 이로 인해 사용자는 메모리 관리에 대해 걱정할 필요가 없다.

소유권의 이동

`std::unique_ptr`는 특정 객체에 대한 독점적인 소유권을 가진다. 하지만 이 소유권은 `이동`할 수 있다. 즉, 소유권을 다른 `std::unique_ptr`로 넘길 수 있고, 이 과정을 소유권 이동(Ownership Migration)이라고 한다.

예를 들어서 다른 함수나 컨테이너로 이동 시키는 상황을 가정해보자.

```
std::unique_ptr<Investment> pInvestment = makeInvestment(args);
std::vector<std::unique_ptr<Investment>> investments;

// 팩토리에서 반환된 unique_ptr를 벡터로 이동
investments.push_back(std::move(pInvestment));

// 벡터의 요소를 객체의 멤버로 이동
MyClass myObject;
myObject.setInvestment(std::move(investments.back()));
```

- `makeInvestment`함수는 `Investment`객체를 생성하고 이를 `std::unique_ptr<Investment>`로 반환한다.
- 반환된 `std::unique_ptr`는 백터(`Investments`)에 삽입된다. 이때 소유권이 `pInvestment`에서 `investments`의 마지막 요소로 이동된다.
- 그 후, 벡터의 요소는 `myClass` 객체의 멤버 변수로 다시 이동되며, 이 객체가 소멸된 때 `unique_ptr`도 함께 소멸되어 자원이 해제된다.

---

deleter

- `std::unique_ptr`는 default로 `delete`를 진행한다. 그러나 사용자가 custimize를 할 수 있다.
- 그러나 `unique_ptr`는 custom deleter를 설정할 수 있다. custom deleter는 함수, 함수 객체, 람다가 될 수 있으며, 객체가 소멸될 때 호출되어 추가 작업(로그 작성)을 진행할 수 있다.
- custom deleter를 사용하면 `unique_ptr`의 크기가 증가할 수 있다.
  - 함수 포인터를 사용하는 deleter는 일반적으로 `unique_ptr`의 크기를 2배로 늘린다.
  - 상태를 가지지 않는 람다 표현식(캡쳐 없는 람다)은 크기를 증가시키지 않으므로, 가능하면 람다를 사용하는 것이 좋다 ⭐

스마트 포인터 배열

- `unique_ptr`는 배열 타입을 지원한다. 그러나 C++에서는 `std::array`, `std::vector`, `std::string`과 같은 STL 컨테이너가 거의 모든 경우에 `std::unique_ptr<T[]>`보다 더 나은 선택이다.

** `std::unique_ptr`와 `std::shared_ptr`간의 변환

`std::unique_ptr`는 독점 소유권을 표현하지만, 이는 필요에 따라 `std::shared_ptr`로 쉽게 변환이 가능하다.

```
std::shared_ptr<Investment> sp = makeInvestment(arguments);
```

- `makeInvestment`함수가 반환한 `std::unique_ptr`가 `std::shared_ptr`로 반환되어 `sp`에 저장된다.

예시코드 (연습용.. ㅎ)

```
#include <bits/stdc++.h>
using namespace std;

vector<unique_ptr<int>> vec;

void input() noexcept
{
    for (int i = 1; i <= 5; i++)
        vec.push_back(std::move(make_unique<int>(i)));
}

int main(void) noexcept
{
    ios::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);
    input();

    for(const auto& elem: vec)
        cout << *elem << " ";

    return 0;
}
```