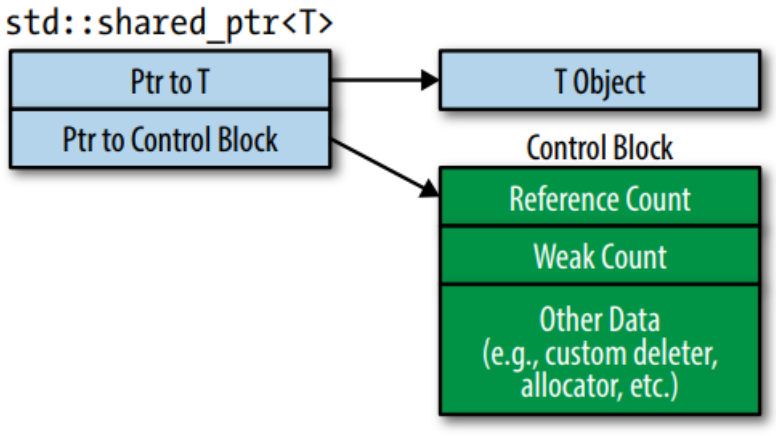
Item 19) Use shared_ptr for shared-ownership resource management.

`std::shared_ptr`에 대해



- `shared_ptr`를 통해 접근되는 객체는 특정 `std::shared_ptr`하나에 의해 소유되지 않고, 해당 객체를 가리키는 모든 `std::shared_ptr`들이 협력하여 그 객체의 생명주기를 관리한다.
- 객체를 가리키는 마지막 `std::shared_ptr`가 더 이상 해당 객체를 가리키지 않게 되면(`std::shared_ptr`가 파괴되거나 다른 객체를 가리키게 될 때)마지막 `std::shared_ptr`는 자신이 가리키던 객체를 자동으로 파괴한다.
- `std::shared_ptr`는 내부적으로 참조 횟수(reference count)를 유지하여, 몇 개의 `std::shared_ptr`가 특정 객체를 가리키고 있는지를 추적한다. 새로운 `std::shared_ptr`가 객체를 가리키면 참조 횟수가 증가하고, 기존의 `std::shared_ptr`가 파괴되거나 다른 객체를 가리키면 참조 횟수가 감소한다. 참조 횟수가 0이 되면 소멸된다.
- `std::shared_ptr` 생성자와 복사 생성자는 참조 횟수를 증가시킨다. 새로운 `std::shared_ptr`가 기존 객체를 가리키게 되면, 그 객체의 참조 횟수가 증가한다.
- 반면, `std::shared_ptr` 소멸자와 복사 대입 연산자는 참조 횟수를 감소시킨다. `std::shared_ptr`가 파괴되거나 다른 객체를 가리키게 되면, 이전 객체의 참조 횟수가 감소한다.

`std::shared_ptr`의 특징

`std::shared_ptr`의 크기는 raw pointer의 2배이다

- raw pointer의 resource와 reference count를 포함하기 때문이다.

reference count의 메모리는 동적으로 할당된다

- 참조 횟수는 `std::shared_ptr`가 관리하는 리소스와 "개념적으로" 연결되어 있지만, 실제로는 리소스 자체와 물리적으로 연ㄴ결되지 않는다. 리소스 객체는 자신의 참조 횟수에 대해 아무것도 모른다.

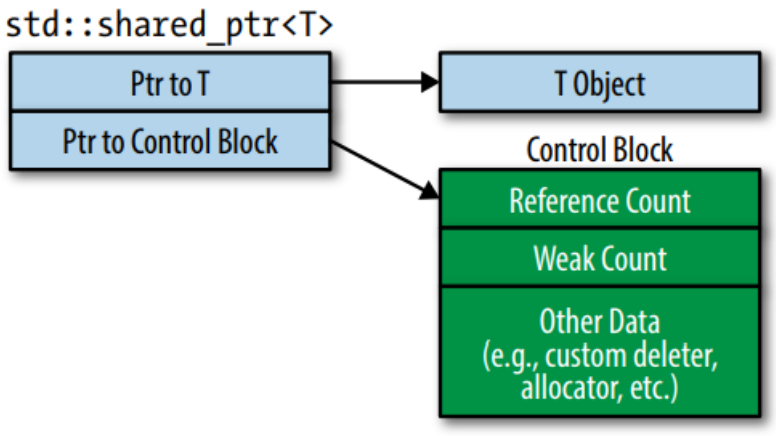
reference count의 Increments와 decrements는 atomic이어야 한다

- `atomic`이란, 프로그래밍에서 **더 이상 나눌 수 없는 하나의 단위 작업**을 의미한다. 이는 해당 연산이 수행되는 동안, 중간에 끼어들거나 다른 작업에 의해 방해받지 않는다는 것을 보장한다.
- `std::shared_ptr`의 참조 횟수는 다중 스레드 환경에서도 안전하게 관리되어야 한다. 예를 들어, 한 스레드에서 `std::share_ptr`가 파괴되거나 다른 객체를 가리키게 되면서 참조 횟수를 감소시키는 동안, 다른 스레드에서 동일한 객체를 가리키는 `std::shared_ptr`가 생성되는 경우, 참조 횟수는 한 스레드에서는 감소 되고, 다른 스레드에서는 증가될 수 있다. 이런 상황을 안전하게 관리하기 위해 참조 횟수의 증가와 감소는 반드시 **원자적**이어햐 한다.
  - PS) `atomic`한 연산은 일반적으로 더 많은 비용이 든다.

복사 생성과 이동 생성에서의 reference count변화

- `std::shared_ptr`를 복사 생성하는 경우, 원본 `shared_ptr`이 가리키는 객체의 참조 횟수가 증가한다.
- 이동 생성은 `std::shared_ptr`를 다른 `shared_ptr`로 옮길 때, 원본 포인터(source shared_ptr)를 null로 설정하고 새로운 포인터(destination shared_ptr)가 리소스를 가리키도록 한다. 이때, 참조 횟수를 증가시킬 필요가 없다.

`std::shared_ptr`의 구조에 대해



`std::shared_ptr`은 두 개의 포인터 크기의 메모리를 사용한다.

1. **리소스 포인터** : 실제로 관리하고 있는 객체를 가리키는 포인터.
2. **컨트롤 블록 포인터** : 객체의 참조 횟소, 커스텀 소멸자, 커스텀 할당자.

컴트롤 블록은 `shared_ptr`의 핵심 구조로, 다음과 같은 정보가 저장된다.

1. 참조 횟수 : `shared_ptr`가 관리하는 객체를 가리키는 모든 `shared_ptr`의 수를 추적한다.
2. 커스텀 소멸자 : 사용자가 지정한 커스텀 소멸자가 있으면, 객체를 파괴할 때 이 소멸자를 사용한다.
3. 커스텀 할당자 : 사용자가 지정한 커스텀 할당자가 있는 경우, 메모리 할당과 해제 작업이 이 할당자가 사용됨.
4. 약한 참조 횟수 : `std::weak_ptr`가 관리하는 약한 참조의 수를 추적한다.

⭐컨트롤 블록은 `std::shared_ptr` 자체의 크기에는 영향을 미치지 않으며, 힙(Heap) 영역에 별도로 할당된다. 따라서 `std::shared_ptr`의 크기는 항상 두 개의 포인터 크기로 일정하게 유지된다.

```
+----------------------------+
| 컨트롤 블록 (Control Block) |
+----------------------------+
| Reference Count            |  <-- 객체의 강한 참조 횟수
+----------------------------+
| Weak Count                 |  <-- 약한 참조 횟수
+----------------------------+
| Custom Deleter (옵션)      |  <-- 커스텀 소멸자가 있다면 포함
+----------------------------+
| Custom Allocator (옵션)    |  <-- 커스텀 할당자가 있다면 포함
+----------------------------+
| Resource Pointer           |  <-- 실제 객체를 가리키는 포인터
+----------------------------+
```

그렇다면 컨트롤 블록은 언제 생성되나요??

1. `std::make_shared`를 사용할 때
   - `std::make_shared`는 항상 새로운 컨트롤 블록을 생성한다.
2. `std::unique_ptr`에서 생성할 때
   - `std::shared_ptr`를 `std::unique_ptr`에서 생성하면, 컨트롤 블록이 새로 생성된다. `std::unique_ptr`은 unique_ownership을 갖는 스마트 포인터로, 기존의 컨트롤 블록을 사용하지 않는다.

```
std::unique_ptr<int> uptr = std::make_unique<int>(10);
std::shared_ptr<int> sptr = std::move(uptr);
```

   - 위의 경우, `uptr`이 가리키던 객체는 `sptr`로 소유권이 이전되면서 새로운 컨트롤 블록이 생성된다. `uptr`은 더 이상 객체를 가리키지 않으며, null 상태가 된다.

3. `raw pointer`로 생성을 할 때
   - `std::shared_ptr`를 raw poineter(일반 포인터)로 생성할 때, 새로운 컨트롤 블록이 생선된다.

```
int* rawPtr = new int(10);
std::shared_ptr<int> sptr(rawPtr);
```

📌shared_ptr 혹은, weak_ptr로 부터 shared_ptr가 만들어지면 컨트롤 블록이 만들어지지 않는다.
📌raw pointer type으로부터 `std::shared_ptr`변수를 만드는 것은 좋은 습관은 아니다.