Item 20) Use weak_ptr for shared_ptrlike pointers that can dangle

---

weak_ptr의 필요성과 역활

`std::share_ptr`는 객체의 생명 주기를 관리하고, 여러 포인터가 객체를 참조할 수 있도록 소유권을 공유하는 스마트 포인터이다. 그러나 때로는 객체의 참조를 유지하면서도 **공유 소유권에 참여하지 않는** 포인터가 필요할 때가 있다.

`std::weak_ptr`는 `std::shared_ptr`와 유사하지만, 객체의 참조 횟수(reference count)에 영향을 미치지 않는다.

즉, `std::weak_ptr`는 객체의 생명 주기를 관리하지 않고, **객체가 존재하는지 여부를 확인**하는 용도로 사용된다.

특히 `std::weak_ptr`는 댕글링 포인터 문제를 해결하기 위해 설계되었다. `std::weak_ptr`는 객체가 파괴되었는지 여부를 추적할 수 있으며, 객체가 존재하지 않으면 이를 감지하고 안전하게 처리할 수 있다.

⭐객체를 안전하게 참조하는 방법

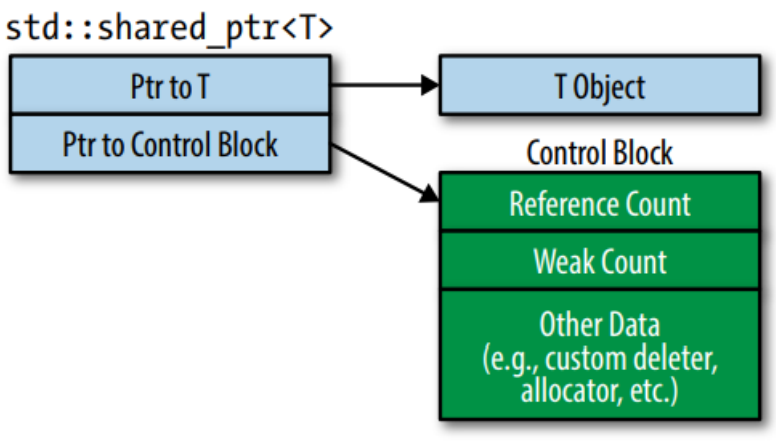
- `lock()` 함수는 내부적으로 `std::shared_ptr`의 참조 횟수(reference count)가 0이 아닌지 확인한다. 참조 횟수가 0이 아니라면, `std::shared_ptr`를 생성하여 반환한다.
- 만약 객체가 이미 소멸된 경우 : `lock()` 함수는 `nullptr`을 반환하여 객체가 더 이상 존재하지 않음을 나타낸다.

```cpp
#include <iostream>
#include <memory> // std::shared_ptr, std::weak_ptr

class Resource {
public:
    Resource() { std::cout << "Resource 생성\n"; }
    ~Resource() { std::cout << "Resource 소멸\n"; }
    void sayHello() const { std::cout << "안녕하세요!\n"; }
};

int main() {
    std::shared_ptr<Resource> sptr = std::make_shared<Resource>();
    std::weak_ptr<Resource> wptr = sptr; // weak_ptr가 shared_ptr를 참조하지만 소유권은 없음

    if (auto spt = wptr.lock()) { // weak_ptr를 shared_ptr로 잠금 (lock)
        std::cout << "객체가 여전히 유효합니다:\n";
        spt->sayHello(); // 객체가 유효하므로 접근 가능
    } else {
        std::cout << "객체가 더 이상 존재하지 않습니다.\n";
    }

    sptr.reset(); // shared_ptr의 소유권 해제, 객체 소멸

    if (auto spt = wptr.lock()) {
        std::cout << "객체가 여전히 유효합니다:\n";
        spt->sayHello();
    } else {
        std::cout << "객체가 더 이상 존재하지 않습니다.\n"; // 이 부분 출력
    }

    return 0;
}
```

`std::weak_ptr`는 `std::shared_ptr`처럼 객체를 직접 가리키는 역활을 하지만, `*`나 `->` 연산자를 통한 객체를 직접 참조할 수 없고, null 여부도 직접 확인할 수 없다.

`weak_ptr`는 독립적인 스마트 포인터라기 보다는 `std::shared_ptr`의 확장된 기능을 제공한다.

`std::weak_ptr`와 `std::shared_ptr`의 크기와 컨트롤 블록



`std::weak_ptr`는 객체의 **소유권**에는 참여하지 않지만, 컨트롤 블록의 약한 참조 횟수에 영향을 미친다. 즉, `std::weak_ptr`가 생성되거나 파괴될 때 약한 참조 횟수는 증가하거나 감소한다.

약한 참조 횟수는 `std::weak_ptr`가 존재하는 동안 객체의 컨트롤 블록이 차괴되지 않도록 보조하는 역활도 한다.

- 예를 들어, `std::share_ptr`의 참조 횟수가 0이 되어 객체가 소멸되더라도, `std::weak_ptr`가 여전히 존재하면 컨트롤 블록은 유지되며 약한 참조 ㅅ횟수가 0이 될 때까지 파괴되지 않는다.

```cpp
#include <iostream>
#include <memory>

class Resource {
public:
    Resource() {
        std::cout << "Resource 생성\n";
    }
    ~Resource() {
        std::cout << "Resource 소멸\n";
    }
};

int main() {
    std::weak_ptr<Resource> weakPtr; // 초기 weak_ptr 생성 (아직 아무 객체도 가리키지 않음)

    {
        std::shared_ptr<Resource> sharedPtr = std::make_shared<Resource>(); // Resource 객체 생성, 참조 횟수(RC) = 1
        weakPtr = sharedPtr; // weakPtr이 sharedPtr을 참조, 약한 참조 횟수(WC) = 1, 참조 횟수는 그대로 1

        std::cout << "sharedPtr 참조 횟수: " << sharedPtr.use_count() << "\n"; // 출력: 1
        std::cout << "weakPtr 유효 여부: " << (weakPtr.expired() ? "만료됨" : "유효함") << "\n"; // 출력: 유효함
    } // sharedPtr 범위 벗어남, 참조 횟수(RC) = 0, Resource 객체 소멸

    // 여기서는 Resource 객체가 이미 소멸되었지만, weakPtr는 여전히 컨트롤 블록을 참조하고 있음
    std::cout << "sharedPtr가 범위를 벗어난 후:\n";
    std::cout << "weakPtr 유효 여부: " << (weakPtr.expired() ? "만료됨" : "유효함") << "\n"; // 출력: 만료됨

    // weakPtr이 소멸될 때까지 컨트롤 블록이 유지됨
    {
        std::weak_ptr<Resource> anotherWeakPtr = weakPtr; // weakPtr를 복사하여 anotherWeakPtr 생성
        std::cout << "weakPtr를 복사한 anotherWeakPtr 생성\n";
    } // anotherWeakPtr 소멸, 약한 참조 횟수(WC) = 0

    std::cout << "모든 weakPtr 소멸 후:\n";
    // 이 시점에서 약한 참조 횟수(WC)가 0이 되어 컨트롤 블록도 소멸됨

    return 0;
}
```

📌 `new`를 사용하는 것 보다는, `make`함수를 사용하는 습관을 들이자!!!