**항목 13. 자원 관리에는 객체가 그만!**

**이슈**.

투자 등을 모델링 해주는 클래스 라이브러리를 가지고 어떤 작업을 한다고 가정하여, Investment라는 최상위 클래스가 있고, 이것을 기반으로 구체적인 파생 클래스가 정해 진다고 하자. 또, 이 라이브러리는 **Investment에서 파생된 클래스의 객체를 사용자가 얻어내는 팩토리 함수만을 쓰도록 만들어져 있다고 하자**.

|  |
| --- |
| class Investment {}; // 투자를 모델링한 클래스 계통의 최상위 클래스  Investment\* createInvestment(); // Investment 클래스 계통에 속한  // 클래스의 객체를 동적 할당하고  // 그 포인터를 반환  // 이 객체의 해제는 호출자 쪽에서 실행되어야함  void f()  {  Investment \*pInv = createInvestment(); // 팩토리 함수를 호출  // ... // pInv 객체를 사용  delete pInv; // 객체를 해제  } |

잘짜여져 있는 것 같으나, createInvestment 함수로부터 얻은 투자 객체의 삭제에 실패할 수 있는 경우가 몇 가지 존재 한다.

1. ‘…’ 부분에 return문 같은 도중하차 코드가 존재할 수 있다.

2. continue, goto 문에 의해 코드를 점프하는 구문이 존재할 수 있다.

3. ‘…’ 안의 어떤 문장에서 예외를 던질 수 있다

위 같은 경우 **메모리가 누수 되며, 객체 안의 모든 자원은 모두 누수로 이어짐**

**해결법.**

createInvestment 함수로 얻어낸 자원이 항상 해제되도록 만들 방법은, **자원을 객체에 넣고 그 자원 해제를 소멸자가 맡도록 하며, 그 소멸자는 실행 제어가 f를 떠날 때 호출되도록 하는 것.**

동적으로 할당 되는 자원은 힙에서 달당 되며, 하나의 블록 혹은 함수 안에서만 쓰이는 경우가 잦기 때문에 그 블록 혹은 함수로부터 실행 제어가 빠져 나올 때 자원을 해제 하는 게 정상이다.

또, **표준 라이브러리의 auto\_ptr를 사용**할 수 있다. **auto\_ptr은 포인터와 비슷하게 동작하는 객체(스마트 포인터)로써, 가리키고 있는 대상에 대해 소멸자가 자동으로 delete를 불러주도록 설계**되어 있다. f에서 생기는 자원 누출을 auto\_ptr을 사용해서 막을 수 있다.

|  |
| --- |
| void f()  {  std::auto\_ptr<Investment> pInv(createInvestment()); // 팩토리 함수를 호출  // ... // pInv 객체를 사용  // auto\_ptr의 소멸자를 통해 pInv를 삭제  } |

위의 코드에서 **자원 관리 객체를 사용하는 방법의 두 가지 특징**을 찾을 수 있다.

**1. 자원을 획득한 후에 자원 관리 객체에게 넘깁니다.**

createInvestment 함수가 만들어준 자원은 그 자원을 관리할 auto\_ptr 객체를 초기화 하는데 사용된다. 자원 획득과 자원 관리 객체의 초기화가 바로 한 문장에서 이루어지는 것을 **자원 획득 즉 초기화**(Resource Acquisition Is Initialization:RAII) 하고 합니다. **“자원을 획득하고 나서 바로 자원 관리 객체에 넘겨 준다는”** 의미 이다.

**2. 자원 관리 객체는 자신의 소멸자를 사용해서 자원이 확실히 해제되도록 한다.**

소멸자는 객체가 소멸될 때 자동으로 호출되기 때문에, 실행 제어가 어떤 경위로 블록을 떠나는가에 상관없이 자원 해제가 제대로 이루어지게 된다.

**auto\_ptr은 자신이 소멸될 때 자신이 가리키고 있는 대상에 대해 자동으로 delete**를 먹이기 때문에, 어떤 객체를 가리키는 auto\_ptr의 개수가 둘 이상이면 안 된다. 만약 그런 경우라면 자원이 두 번 삭제되는 결과를 낳게 되며, 프로그램은 미정의 동작을 수행하게 된다.

이런 특성 때문에 **auto\_ptr은 객체를 복사하면(복사 생성자나, 복사 대입 연산자를 사용하여) 원본 객체는 null** 된다. 복사하는 객체만이 그 자원의 유일한 소유권을 갖는다.

|  |
| --- |
| std::auto\_ptr<Investment> pInv1(createInvestment());  std::auto\_ptr<Investment> pInv2(pInv1); // pInv2는 현재 그 객체를 가리키고  // pinv1은 null 값이 됨  pInv1 = pInv2; // pInv1이 현재 객체를 가리키고  // pInv2는 null 값이 됨 |

이런 특징 때문에 auto\_ptr을 사용하는 데에는 제약이 많이 따르게 됩니다. auto\_ptr을 쓸 수 없는 상황 이라면 **참조 카운팅 방식 스마트 포인터(reference-counting smart pointer: RCSP)**를 사용할 수 있습니다. RCSP는 어떤 자원을 가리키는 외부 객체의 개수를 유지하고 있다가 그 개수가 0이 되면 해당 자원을 삭제하는 스마트 포인터 입니다. 가비지 컬렉션과 비슷하나. 서로가 서로를 가리키는 경우 자원의 해제가 안 된다는 것이 가비지 컬렉션과 다른 점입니다.

**TR1에서 제공되는 tr1::shared\_ptr이 가장 대표적인 RCSP 이다.**

|  |
| --- |
| std::tr1::shared\_ptr<Investment> pInv1(createInvestment());  // ...  // shared\_ptr의 소멸자를 통해 pInv를 자동으로 삭제 |

auto\_ptr과 같아 보이나, shared\_ptr은 복사, 해제가 자유롭습니다.

|  |
| --- |
| std::tr1::shared\_ptr<Investment> pInv1(createInvestment());  std::tr1::shared\_ptr<Investment> pInv2(pInv1);  pInv1 = pInv2; |

스마트 포인터에 대한 내용이지만 요점은 **“자원을 관리하는 객체를 써서 자원을 관리하는 것이 중요하다.”** 입니다.

**주의 할 점은 auto\_ptr, shared\_ptr은 소멸자 내부에서 delete 연산자를 사용합니다. delete[] 연산자가 아닙니다.** 고로, 동적으로 할당된 배열을 auto\_ptr, shared\_ptr에서 사용하면 안된다는 말입니다.

|  |
| --- |
| std::auto\_ptr<std::string> aps(new std::string[10]); // 잘못된 delete가 사용됨  std::tr1::shared\_ptr<int> spi(new int[10]); // 잘못된 delete가 사용됨 |

왜냐면 동적으로 할당된 배열은 vector 및 string으로 거의 대체할 수 있기 때문입니다.

배열에 쓸 수 있는 auto\_ptr, shared\_ptr은 부스트 라이브러리에 존재한다. boost::scoped\_array, boost::shared\_array가 그것이다.

**요점.**

\* 자원 누출을 막기 위해, 생성자 안에서 자원을 획득하고 소멸자에서 그것을 해제하는 RAII객체를 사용합니다.

\* 일반적으로 널리 쓰이는 RAII클래스는 tr1::shared\_ptr 그리고 auto\_ptr이다. 이둘 가운데 tr1::shared\_ptr이 복사 시의 동작이 직관적이기 때문에 대개 더 좋습니다. 반면, auto\_ptr은 복사되는 객체를 null로 만들어 버립니다.