**항목 15. 자원 관리 클래스에서 관리되는 자원은 외부에서 접근할 수 있도록 하자.**

|  |
| --- |
| **std::tr1::shared\_ptr<Investment>** pInv(createInvestment()); |

이때 어떤 Investment 객체를 사용하는 함수로서, 사용자가 사용하려고 하는 것이 다음과 같다고 가정.

|  |
| --- |
| Int dayHeld(const Investment \*pi) // 투자금이 유입된 이후로 경과한 날수 |

그리고 이렇게 호출하고 싶을 것이다..

|  |
| --- |
| Int days = daysHeld(pInv); // 에러! |

애석하게도 이 코드는 컴파일이 되지 않는다.   
daysHeld 는 Investment\* 를 원하는데, **여러분은 tr1::shared\_ptr<Investment> 을 넘기고 있기 때문**이다.

그래서, RAII 클래스(지금의 tr1::shared\_ptr)의 객체를, 그 객체가 감싸고 있는 실제 자원(그러니까 Investment\*)으로 변환할 방법이 필요해진다.

이런 목적에 일반적인 방법을 쓴다면 2가지가 있는데,

하나는 **명시적 변환(explicit conversion)**이고,

다른 하나는 **암시적 변환(implicit conversion)**이다.

tr1::shared\_ptr 및 auto\_ptr 은 **명시적 변환을 수행하는 get이라는 멤버 함수를 제공**한다.  
이 함수를 사용하면, 각 타입으로 만든 스마트 포인터 객체에 들어 있는 실제 포인터(의 사본)를 얻어낼 수 있다.

|  |
| --- |
| int days = daysHeld(pInv.**get()**); // 이제 문제없습니다. pInv에 들어 있는 실제 포인터를 daysHeld에 넘기니까요. |

제대로 만들어진 스마트 포인터 클래스라면 거의 모두가 그렇듯,  
tr1::shared\_ptr과 auto\_ptr은 **포인터 역참조 연산자(operator-> 및 operator\*)**도 오버로딩하고 있다. 따라서 자신이 관리하는 실제 포인터에 대한 **암시적 변환도 쉽게 할 수 있다**.

|  |
| --- |
| class Investment { // 여러 형태의 투자를 모델링한 클래스 계통의 최상위 클래스  public:  bool isTaxFree() const;  };    Investment\* createInvestment(); // 팩토리 함수.    std::tr1::shared\_ptr<Investment> pi1(createInvestment()); // tr1::shared\_ptr이 자원관리를 맡도록 합니다.  bool taxable1 = !(pi1**->**isTaxFree()); // operator->를 써서 자원에 접근합니다.  std::tr1::shared\_ptr<Investment> pi2(pInv1); // auto\_ptr이 자원관리를 맡도록 합니다.  bool taxable2 = !((\*pi2)**.**isTaxFree()); // operator\*를 써서 자원에 접근합니다. |

RAII 객체 안에 들어 있는 실제 자원을 얻어낼 필요가 종종 생기기 때문에,

RAII 클래스 설계자 중에는 **암시적 변환함수를 제공하여 자원 접근을 매끄럽게 할 수 있도록 하기도 한다**.

하부 수준 C API로 직접 조작이 **가능한 폰트를 RAII 클래스로 둘러싸서 쓰는 경우**.

|  |
| --- |
| class Font { // RAII 클래스  public:             explicit Font(FontHandle fh) : f(fh) {}  // 자원을 획득합니다. 여기서 값에 의한 전달이 수행되는 것에 주의.  // 자원해제를 C API로 하기 때문.  ~Font() { releaseFont(f); }  private:  FontHandle f; // 실제 폰트 자원  }; |

Font 클래스에서는 **명시적 변환 함수로 get을 제공할 수 있다**.

|  |
| --- |
| class Font {  public:  **FontHandle get() const { return f; }** // 명시적 변환 함수  }; |

이렇게 해두면 쓸 수 있긴 한데, 사용자는 하부 수준 API를 쓰고 싶을 때마다 get을 호출해야 할 것이다.

|  |
| --- |
| void changeFontSize(FontHandle f, int newSize); // 폰트 API의 일부    Font f(getFont());  int newFontSize;  changeFontSize(f.**get()**, newFontSize); // Font에서 FontHandle로 명시적으로 바꾼 후에 넘깁니다. |

변환할 때마다 함수를 호출하는 것이 번거로울 수 있다.  
대신 FontHandle로의 암시적 변환 함수를 Font에서 제공하도록 하면 된다.

|  |
| --- |
| class Font {  public:  **operator FontHandle() const { return f; }** // 암시적 변환 함수  }; |

암시적 변환 함수로 C API를 사용하기가 훨씬 쉬워지고 자연스러워진다.

|  |
| --- |
| void changeFontSize(FontHandle f, int newSize); // 폰트 API의 일부    Font f(getFont());  int newFontSize;  changeFontSize(**f**, newFontSize); // Font에서 FontHandle로 암시적 변환을 수행합니다. |

그러나, 암시적 변환이 들어가면 실수를 저지를 여지가 많아진다.  
Font를 쓰려고 한 부분에서 원하지도 않았는데, FontHandle로 바뀔 수도 있다.

|  |
| --- |
| Font f1(getFont());  **FontHandle** f2 = f1;  // 원래 의도는 Font 객체를 복사하는 것이었는데, 엉뚱하게도 f1이 FontHandle로 바뀌고 나서 복사되어 버렸습니다. |

이렇게 되면 Font 객체인 f1이 관리하고 있는 폰트(FontHandle)가 f2를 통해서도 직접 사용할 수 있는 상태가 된다.  
하나의 자원이 양다리를 걸치고 있는, 이와 같은 상황은 좋지 않습니다.  
f1이 소멸될 시점에 폰트가 해제될 텐데, 그럼 f2는 해제된 폰트에 매달려 있는 모양이 된다.

RAII 클래스를 실제 자원으로 바꾸는 방법으로서, 명시적 변환을 제공할 것인지(get 멤버 함수 등) 아니면 암시적 변환을 허용할 것인지에 대한 결정은, 그 RAII 클래스만의 특정한 용도와 사용환경에 따라 달라진다.   
어쨌든 가장 잘 설계한 클래스라면 (18)에 따라 “**맞게 쓰기에는 쉽게, 틀리게 쓰기에는 어렵게**” 만들어져야 할 것입니다.

늘 그런 것은 아니지만, 암시적 변환보다는 get 등의 명시적 변환함수를 제공하는 쪽이 나을 때가 많습니다. 원하지 않은 타입 변환이 일어날 여지를 줄여주는 것은 확실하니까요.  
하지만 암시적 타입 변환에서 생기는 사용 시의 자연스러움이 빛을 발하는 경우도 있다는 점도 알아두세요.

RAII 클래스에서 자원 접근 함수를 열어 주는 설계가, 혹시 캡슐화에 위배되는 것이 아닌지 생각할 수도 있다. 그렇긴 하지만 틀린 설계도 아니다.  
**RAII 클래스는 애초부터 데이터 은닉이 목적이 아니다. 원하는 동작(자원 해제)이 실수 없이 이루어지도록 하면 OK인 것.**  
굳이 원한다면 ‘자원 해제’라는 기본 기능 위에 캡슐화 기능을 얹을 수는 있겠지만, 꼭 필요한 것은 아니다.

시중의 RAII 클래스 중에는 이미 **자원의 엄격한 캡슐화와 느슨한 캡슐화를 동시에 지원하는 것들 것** 꽤 있다. tr1::shared\_ptr이 대표적 예인데, 이 클래스는 참조 카운팅 매커니즘에 필요한 장치들은 모두 캡슐화하고 있지만, 그와 동시에 자신이 관리하는 포인터를 쉽게 접근할 수 있는 통로도 여전히 제공하고 있다.

꼼꼼히 제대로 설계된 클래스가 그렇듯, 사용자가 볼 필요가 없는 데이터는 가리지만, 고객차원에서 꼭 접근해야 하는 데이터는 열어 주는 것이다.

**요약**

**\*** 실제 자원을 직접 접근해야 하는 기존 API들도 많기 때문에, RAII 클래스를 만들 때는, 그 클래스가 관리하는 자원을 얻을 수 있는 방법을 열어 주어야 합니다.

**\***  자원 접근은 명시적 변환 혹은 암시적 변환을 통해 가능합니다. 안전성만 따지면 명시적 변환이 대체적으로 더 낫지만, 고객 편의성을 놓고 보면 암시적 변환이 괜찮습니다.