**항목 18. 인터페이스 설계는 제대로 쓰기엔 쉽게, 엉터리로 쓰기엔 어렵게 하자.**

**이슈**

어떤 인터페이스를 어떻게 써 봤는데, 결과 코드가 사용자가 생각한 대로 동작하지 않는다면, 그 코드는 컴파일 되지 않아야 맞습니다. 거꾸로 생각해서, 어떤 코드가 컴파일 된다면, 그 코드는 사용자가 원하는 대로 동작해야 할 것이다.

**‘제대로 쓰기에 쉽고 엉터리로 쓰기에 어려운’** 인터페이스를 개발하려면 우선 사용자가 저지를 만한 실수의 종류를 머리에 넣어두고 있어야 합니다.

**상황.**  
예를 들어, 날짜를 나타내는 어떤 클래스에 넣을 생성자를 설계하고 있다고 가정합시다

|  |
| --- |
| class Date {  public:  Date(int month, int day, int year);  ...  }; |

인터페이스가 별 문제 없을 것 같지만, 여기에는 사용자가 쉽게 저지를 수 있는 오류가 적어도 2개 정도 존재한다.

**1. 매개변수의 전달 순서가 잘못될 여지가 있다**.

|  |
| --- |
| Date d(**30, 3,** 1995); // “3, 30” 이어야 하는데 “30, 3”을 넣었음. |

**2. 월과 일에 해당하는 숫자가 어이없는 숫자일 수 있다는 점.**

|  |
| --- |
| Date d(**30, 40,** 1995); // “3, 30” 이어야 하는데 “30, 40”을 넣었음. |

새로운 타입을 들여와 인터페이스를 강화하면, 상당수의 사용자 실수를 막을 수 있습니다.  
어처구니없는 코드가 컴파일 되는 부조리로부터 지켜주는 오른팔이 바로 타입 시스템입니다.  
지금의 경우, 일, 월, 연을 구분하는 간단한 **랩퍼(wrapper) 타입을 각각 만들고 이 타입을 Date 생성자 안에 둘 수 있을 것**입니다.

|  |
| --- |
| **struct Day** {  explicit Day(int d) : val(d){}  int val;  };    **struct Month** {  explicit Month(int m) : val(m){}  int val;  };    **struct Year** {  explicit Year(int y) : val(y){}  int val;  };    class Date {  public:  Date(**const Month& month, const Day& day, const Year& year**);  ...  }; |
| Date d(30, 33, 1995); // 에러! 타입이 틀렸습니다.  Date d(Day(30), Month(3), Year(1995)); // 에러! 타입이 틀렸습니다.  Date d(Month(3), Day(30), Year(1995)); // 타입이 전부 맞습니다. |

물론, Day, Month, Year에 데이터를 이것저것 숨겨 넣어, 제 몫을 하는 온전한 클래스로 만들면 위의 단순한 구조체보다는 확실히 낫긴 하겠지만, 타입을 적절히 새로 준비해 두기만 해도, **인터페이스 사용 에러를 막는 데는 약발이 통한다는 점**을 보여주기에는 이 정도도 충분하다.

일단 적절한 타입만 제대로 준비되어 있으면, 각 타입의 값에 제약을 가하더라도 괜찮은 경우가 생기게 됩니다. 예를 들어 월(月)이 가질 수 있는 **유효한 값은 12개**뿐이므로, Month 타입은 이 사실을 제약으로 사용할 수 있다.

한가지 방법으로 월 표시 값을 나타내는 enum을 넣는 방법이 있는데, 타입 안정성은 그리 좋지 못합니다. 때로는 int 처럼 쓰일 수 있다. 타입 안정성이 신경 쓰인다면 유효한 Month의 집합을 미리 정의해 두어도 괜찮다.

|  |
| --- |
| class Month {  public:  static Month Jan() { return Month(1); } // 유효한 Month값을 반환하는 함수들.  static Month Feb() { return Month(2); }  // 이들이 어째서 함수로 되어 있으며, 객체가 되면 안 되는지에 대한 이야기는 아래를 보시면 있습니다.  ...  static Month Dec() { return Month(12); }  ...  private:  explicit Month(int m); // Month 값이 새로 생성되지 않도록 명시호출 생성자가 private 멤버입니다.  ... // 월 표현을 위한 내부 데이터  };    Date d(**Month::Mar()**, Day(30), Year(1995)); |

특정한 월을 나타내는데 객체를 쓰지 않고 함수를 쓴 것에 대해, **비 지역 정적 객체들의 초기화**를 믿고 밀고 나가는 것은 그 자체로 문제가 있다.

예상되는 사용자 실수를 막는 다른 방법으로는, **어떤 타입이 제약을 부여하여 그 타입을 통해 할 수 있는 일들을 묶어 버리는 방법**이 있다.  
제약 부여 방법으로 아주 흔히 쓰이는 예가 **‘const 붙이기’**이다.   
**항목3.** 에 설명되어 있는데, operator\*의 반환타입을 const로 한정함으로써, 사용자가 사용자 정의 타입에 대해 다음과 같은 실수를 저지르지 않도록 할 수 있다.

|  |
| --- |
| if (a \* b **=** c) … // 원래 비교하려고 그랬던 건데! |

위 예는 ‘**그렇게 하지 않을 번듯한 이유가 없다면 사용자 정의 타입은 기본제공 타입처럼 동작하게 만들지어다**’ 라고 하지요.  
int 등의 타입 정도는 사용자들이 그 성질을 이미 다 알고 있기 때문에, 여러분이 사용자를 위해 만드는 타입도 웬만하면 이들과 똑같이 동작하게 만드는 센스를 갖추어라 이것이다. 위의 문장처럼 a와 b가 int라면 a\*b에 대입한다는 게 말이 안되기 때문이다.   
그러니까, int와 굳이 다른 길을 걸어갈 이유가 없다면, 여러분이 만드는 타입에서도 이게 말이 안되어야 한다.

기본 제공 타입과 쓸데없이 어긋나는 동작을 피하는 실질적인 이유는 일관성 있는 인터페이스를 제공하기 위해서이다.  
제대로 쓰기에 괜찮은 인터페이스를 만들어 주는 요인 중에 일관성만큼 똑 부러지는 것이 별로 없으며, 편찮은 인터페이스를 더 나쁘게 만들어 버리는 요인 중에 비 일관성을 따라오는 것이 거의 없다.  
STL 컨테이너의 인터페이스는 전반적으로(완벽하진 않지만) 일관성을 갖고 있으며, 이 때문에 사용하는데 큰 부담이 없다.

사용자 쪽에서 뭔가를 외워야 제대로 쓸 수 있는 인터페이스는 잘못 쓰기 쉽습니다. 언제라도 잊어버릴 수 있으니까요. (13)의 팩토리 함수를 예로 보자  
이 함수는 Investment 클래스 계통에 속해 있는 어떤 객체를 동적 할당하고, 그 객체의 포인터를 반환하는 함수이다.

|  |
| --- |
| Investment \* createInvestment(); // 항목 (13)에서 가져온 함수. 매개변수는 편의상 생략. |

이 함수를 사용할 때는, 자원누출을 막기 위해 createInvestment에서 얻어낸 포인터를 삭제해야 합니다. 그런데 이점 때문에 사용자가 실수를 최소한 2가지 저지를 가능성이 만들어진다.  
**포인터 삭제를 깜박 잊을 수 있고, 똑 같은 포인터에 대해 delete가 2번이상 적용될 수 있다.**

createInvestment의 반환값을 auto\_ptr이나 tr1::shared\_ptr 등의 스마트 포인터에 저장한 후에, 해당 포인터의 삭제 작업을 스마트 포인터에게 떠넘길 수 있다.  
하지만 이 스마트 포인터를 사용해야 한다는 사실도 사용자가 잊어버리면 어떻게 하죠?  
차라리 처음부터 문제가 생길 여지를 일망타진해 놓는 편이 좋다. 즉, **애초부터 팩토리 함수가 스마트 포인터를 반환하게 만드는 것**이다.

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

이렇게 해두면, 이 함수의 반환 값은 tr1::shared\_ptr에 넣어둘 수밖에 없을 뿐더러, 나중에 Investment 객체가 필요 없어졌을 때 이 객체를 삭제하는 것을 깜빡하고 넘어가는 불상사도 생기지 않을 것입니다.

**tr1::shared\_ptr 을 반환하는 구조는 자원 해제에 관련된 상당수의 사용자 실수를 사전 봉쇄할 수도 있어서 여러모로 인터페이스 설계자에게 좋습**니다.  
이유는 (14)에서 언급되었듯이 tr1::shared\_ptr은 생성 시점에 자원 해제 함수(일명 ‘삭제자’)를 직접 엮을 수 있는 기능을 갖고 있기 때문입니다(auto\_ptr은 이런 기능이 없음)

createInvestment를 통해 얻은 Investment\* 를 직접 삭제하지 않게 하고, getRidOfInvestment 라는 이름의 함수를 준비해서 여기에 넘기게 하면 어떨까요.  
왠지 더 깔끔해 보이지만 **이런 인터페이스는 되레 사용자 실수를 하나 더 열어놓는 결과**입니다.  
자원 해제 메커니즘을 잘못 사용할 수가 있거든요(getRidOfInvestment를 잊어버리고 delete를 쓴다든지).  
createInvestment를 살짝 고쳐서, getRidOfInvestment 가 삭제자로 묶인 tr1::shared\_ptr을 반환하도록 구현해 둔다면 이런 문제는 발도 못 들여놓을 것입니다.

tr1::shared\_ptr 에는 2개의 인자를 받는 생성자가 있다.   
**첫 번째 인자는 이 스마트 포인터로 관리할 실제 포인터**이고,   
**두 번째 인자는 참조 카운터가 0이 될 때 호출될 삭제자** 입니다.  
그래서 tr1::shared\_ptr이 널(null) 포인터를 물게 함과 동시에 삭제자로 getRidOfInvestment를 갖게 하는 방법으로 다음과 같은 코드를 생각해 봅니다.

|  |
| --- |
| std::tr1::shared\_ptr<Investment> pInv(**0, getRidOfInvestment**); // 이렇게 해서 사용자 정의 삭제자를 가진 널 shared\_ptr을 생성했으면 좋겠습니다. 그런데 컴파일이 안됩니다. |

이것은 제대로 쓴 C++ 코드가 아닙니다.   
tr1::shared\_ptr의 ‘그’ 생성자는 첫번째 매개변수로 **포인터**를 받아야 합니다.  
그런데 0은 포인터가 아니라 int이죠.  
물론 0은 포인터로 **변환할 수 있지만** 지금의 경우에는 이것만으로는 부족합니다.  
tr1::shared\_ptr이 요구하는 포인터는 Investment\* 타입의 실제 포인터이기 때문입니다.   
그래서 캐스트를 적용하여 사태를 해결합니다.

|  |
| --- |
| std::tr1::shared\_ptr<Investment> pInv(**static\_cast<Investment\*>(0)**, getRidOfInvestment); // getRidOfInvestment를 삭제자로 갖는 널 shared\_ptr을 생성합니다. static\_cast은 (27)에서 확인 |

createInvestment에서 getRidOfInvestment를 삭제자로 갖는 tr1::shared\_ptr을 반환하도록 구현 예.

|  |
| --- |
| std::tr1::shared\_ptr<Investment> createInvestment()  {             std::tr1::shared\_ptr<Investment> retVal(static\_cast<Investment\*>(0), getRidOfInvestment);             retVal = ...; // retVal은 실제 객체를 가리키도록 만듭니다.             return retVal;  } |

retVal 로 관리할, 실제 객체의 포인터를 결정하는 시점이,   
retVal 을 생성하는 시점보다 앞설 수 있으면, 위의 코드처럼 retVal을 널로 초기화하고 나서 나중에 대입하는 방법보다, 실제 객체의 포인터를 바로 retVal의 생성자에 넘겨버리는 게 더 낫습니다. (26)

tr1::shared\_ptr에는 좋은 특징이 하나 있습니다.  
바로 포인터 별(per-pointer) 삭제자를 자동으로 씀으로써 사용자가 저지를 수 있는 또 하나의 잘못을 미연에 없애준다는 점인데,   
이 또 하나의 잘못이란 바로 ‘**교차 DLL 문제(cross-DLL problem)**’입니다.   
이 문제가 생기는 경우가 언제냐 하면, **객체 생성시에 어떤 동적 링크 라이브러리의 new를 썼는데, 그 객체를 삭제할 때는 이전의 DLL과 다른 DLL에 있는 delete를 썼을 경우**입니다.   
이렇게 new/delete 짝이 실행되는 DLL이 달라서 꼬이게 되면 대다수의 플랫폼에서 런타임 에러가 일어납니다.   
그런데 tr1::shared\_ptr은 이 문제를 피할 수 있습니다.   
이 클래스의 **기본 삭제자는 tr1::shared\_ptr이 생성된 DLL과 동일한 DLL에서 delete를 사용하도록 만들어져 있기 때문** 입니다.

예를 들어 Stock이라는 클래스가 Investment에서 파생된 클래스이고 createInvestment 함수가 아래와 같이 구현되어 있다고 할 때,

|  |
| --- |
| std::tr1::shared\_ptr<Investment> createInvestment()  {  return std::tr1::shared\_ptr<Investment> (new Stock);  } |

이 함수가 반환하는 tr1::shared\_ptr은 다른 DLL들 사이에 이리저리 넘겨지더라도, 교차 DLL문제를 걱정하지 않아도 된다는 뜻입니다.  
Stock 객체를 가리키는 tr1::shared\_ptr은 그 Stock 객체의 참조 카운트가 0이 될 때 어떤 DLL의 delete를 사용해야 하는지를 꼭 붙들고 잊지 않습니다.

참고로, tr1::shared\_ptr을 구현한 제품 중 가장 흔히 쓰이는 것은 boost 라이브러리(55) 입니다.  
부스트의 shared\_ptr은 일단 크기가 **원시 포인터의 2배입니다**. 그리고 내부 관리 데이터 및 삭제자 메커니즘을 돌릴 데이터를 위해 동적 할당 메모리를 사용하며, **다중 스레드로 돌아가는 프로그램을 지원할 경우에는 참조 카운트를 변경할 때 스레드 동기화 오버헤드**를 일으킵니다(다중 스레드 지원을 비활성화 시킬 수도 있습니다. 선행 처리자 기호를 정의하면 됩니다).   
간단히 말해 이 클래스를 사용하면 원시 포인터보다 크고 느리며 게다가 내부 관리용 동적 메모리까지 추가로 매달립니다. 하지만 이런 것들 때문에 응용프로그램에서 런타임 비용이 눈에 띄게 늘어나는 경우는 어지간해서는 찾기 힘들 것입니다.   
반면에 사용자 실수가 눈에 띄게 줄어드는 경우는 모든 사람들이 잡아낼 수 있을 정도죠.

**요약**

- 좋은 인터페이스는 제대로 쓰기에 쉬우며 엉터리로 쓰기에 어렵습니다. 인터페이스를 만들 때 이 특성을 지닐 수 있도록 고민하고 또 고민합시다.

- 인터페이스의 올바른 사용을 이끄는 방법으로는 인터페이스 사이의 일관성 잡아주기, 그리고 기본제공 타입과의 동작 호환성 유지하기가 있습니다.

- 사용자의 실수를 방지하는 방법으로는 새로운 타입 만들기, 타입에 대한 연산을 제한하기, 객체의 값에 대해 제약 걸기, 자원 관리 작업을 사용자 책임으로 놓지 않기가 있습니다.

- tr1::shared\_ptr은 사용자 정의 삭제자를 지원합니다. 이 특징 때문에 tr1::shared\_ptr은 교차 DLL 문제를 막아주며, 뮤텍스 등을 자동으로 잠금 해제하는 데 쓸 수 있습니다.