**항목 25. 예외를 던지지 않는 swap에 대한 지원도 생각해 보자.**

Swap은 초창기부터 STL에 포함된 이래로 예외 안전성 프로그래밍(29)에 없어선 안될 감초 역할로서, 자기대입 현상(11)의 가능성에 대처하기 위한 대표적인 메커니즘으로서 널리 사랑 받아 왔다.  
하지만 이 함수에 관련된 특이한 말썽거리들도 도사리고 있다.

**두 객체의 값을 ‘맞바꾸기(swap)’ 한다는 것은, 각자의 값을 상대방에게 주는 동작입니다.**  
기본적으로는 이 맞바꾸기 동작을 위해, 표준 라이브러리에서 제공하는 swap 알고리즘을 쓰는데, 이 알고리즘이 구현된 모습을 보면 여러분이 알고 있는 그 ‘swap’과 하나도 다르지 않다는 것을 알 수 있다.

|  |
| --- |
| namespace std {             templatee<typename T> // std::swap의 전형적인 구현             void swap(T& a, T& b) // a의 값과ㅏ b의 값을 맞바꿉니다.             {                       T temp(a);                       a = b;                       b = temp;             }  } |

표준에서 기본적으로 제공하는 swap(이하 표준 swap)은 구현 코드와 같이 복사만 제대로 지원하는(복사 생성자 및 복사 대입 연산자를 통해) 타입이기만 하면 어떤 타입의 객체이든 맞바꾸기 동작을 수행해 줍니다. Swap을 위해 특별히 추가 코드를 마련하거나 할 필요가 없습니다.

표준 swap의 동작은 한번 호출에 복사가 3번 일어납니다.   
타입에 따라서는 이런 temp 사본이 필요 없는 경우도 있다.

복사하면 손해를 보는 타입들 중 으뜸은, 다른 타입의 실제 데이터를 가리키는 포인터가 주성분(!)일 것이다.  
이러한 개념을 설계의 미학으로 끌어올려 많이들 쓰고 있는 기법이 바로 pimpl (pointer to implementation) 이지요.

pimpl 설계를 차용하여 Widget 클래스를 만든 예

|  |
| --- |
| class WidgetImpl { // Widget 의 실제 데이터를 나타내는 클래스.  public:             ...    private:             int a, b, c,;             std::vector<double> v;             ..  };;    class Widget { // pimpl 관용구를 사용한 클래스  public:             Widget(const Widget& rhs);             Widget& operator=(const Widget& rhs)             {                       // Widget을 복사하기 위해 자신의 WidgetImpl 객체를 복사합니다.                       // operator=의 일반적인 구현 방법에 대한 자세한 사항은 (10),(11),(12)                       ...                       \*pImpl = \*(rhs.pImpl);                       ...             }             ...  private:  **WidgetImpl \*pImpl;** // Widget의 실제 데이터를 가진 객체에 대한 포인터.  }; |

이렇게 만들어진 Widget 객체를 우리가 직접 맞바꾼다면, pImpl 포인터만 살짝 바꾸는 것 말고는 실제로 할 일이 없습니다.   
하지만 표준 swap 알고리즘은 Widget 객체 3개를 복사하고 WidgetImpl 객체 3개도 복사할 것입니다.

Widget 객체를 맞바꿀 때는 일반적인 방법을 쓰지 말고, 내부의 pImpl 포인터만 맞바꾸라고 std::swap에게 뭔가를 알려줍니다.  
C++에서 std::swap을 Widget에 대해 특수화(specialize) 하는 것.

|  |
| --- |
| namespace std {  **template<>**             void  swap**<Widget>**(Widget& a, Widget& b)             {                       // 이 코드는 T가 Widget일 경우에 대해                       // std::swap을 특수화한 것입니다.                       swap(a.pImpl, b.pImpl); // Widget을 'swap'하기 위해, 각자의 pImpl만 맞바꿈.             }  } |

함수 시작부분에 있는 ‘template<>’.  
이 함수가 std::swap의 **완전 템플릿 특수화(total template specialization)** 함수라는 것을 컴파일러에게 알려주는 부분입니다.  
그리고 함수 이름 뒤에 있는 ‘<Widget>’은 T가 Widget일 경우에 대한 특수화라는 사실을 알려주는 부분.  
즉, 타입에 무관한 swap 템플릿이 Widget에 적용될 때는 위의 함수 구현을 사용해야 한다는 뜻.

일반적으로 std 네임스페이스의 구성요소는 함부로 변경하거나 할 수 없지만,   
프로그래머가 직접 만든 타입(Widget 등)에 대해, 표준 템플릿(swap 같은)을 완전 특수화하는 것은 허용이 됩니다.

위 함수는 컴파일되지 않는다.   
문법이 틀린 것이 아니라, a와 b에 들어 있는 pImpl 포인터에 접근하려고 하는데, 이들 포인터가 private 멤버이기 때문이다.   
특수화 함수를 프렌드로 선언할 수도 있지만, 이렇게 하면 표준 템플릿들에 쓰인 규칙과 어긋나므로 좋은 모양은 아니다.   
그래서 Widget 안에 swap이라는 public 멤버 함수를 선언하고, 그 함수가 실제 맞바꾸기를 수행하도록 만든 후에, std::swap 의 특수화 함수에게 그 멤버 함수를 호출하는 일을 맡깁니다.

|  |
| --- |
| clas Widget { // 앞의 예와 같은데, swap 멤버 함수가 추가된 것만 다름.  public:             ...  **void swap(Widget& other)**  **{**  **using std::swap; // 이 선언문이 필요한 이유는 이후의 설명에서 확인가능.**    **swap(pImpl, other.pImpl); // Widget을 맞바꾸기 위해, 각 Widget의 pImpl 포인터를 맞바꿉니다.**  **}**             ...  };    namespace std {             template<>             void  swap<Widget>(Widget& a, Widget& b)             {                       // std::swap 템플릿의 특수화 함수를 살짝 고친 결과.  **a.swap(b);** // Widget을 맞바꾸기 위해, swap 멤버함수 호출.             }  } |

컴파일 되고, 기존의 STL 컨테이너와 일관성도 유지되는 착한 코드가 되었습니다.  
public 멤버 함수 버전의 swap과 이 멤버함수를 호출하는 std::swap의 특수화 함수 모두 지원하고 있고요.

만약, Widget과 WidgetImpl이 클래스가 아니라 클래스 템플릿으로 만들어져 있어서, WidgetImpl에 저장된 데이터의 타입을 매개변수로 바꿀 수 있다면 어떻게 될까요?

|  |
| --- |
| **template<typename T>**  class WidgetImpl {...};    **template<typename T>**  class Widget {...}; |

Swap 멤버 함수를 Widget에 (필요하면 WidgetImpl에도) 넣는 것은 어렵지 않으나,   
std::swap을 특수화하는 데서 좌절입니다.

작성하려던 코드는 이런 것이었으니까요.

|  |
| --- |
| namespace std {  **template<typename T>**  **void swap<Widget <T> >**(Widget<T>&a, Widget<T>& b) // 에러! 적법하지 않은 코드.             {                       a.swap(b);             }  } |

C++의 기준에는 적법하지 않습니다.  
지금 함수 템플릿(std::swap)을 부분적으로 특수화해 달라고 컴파일러에게 요청한 것인데,   
C++는 클래스 템플릿에 대해서는 부분 특수화(partial specialization)를 허용하지만,   
함수 템플릿에 대해서는 허용하지 않도록 정해져 있습니다.  
(함수 템플릿의 부분 특수화를 받아들이는 어리버리 컴파일러도 있긴 합니다).

함수 템플릿을 ‘부분적으로 특수화’하고 싶을 때, 흔히 취하는 방법은, 그냥 오버로드 버전을 하나 추가하는 것입니다.

|  |
| --- |
| namespace std {             template<typename T>             void swap(Widget<T>&a, Widget<T>& b)             {                       // std::swap을 오버로드한 함수.                       // "swap" 뒤에 "<...>"가 없는 것에 주의.                       // 이 함수가 왜 유효하지 않은지는 뒤에...                       a.swap(b);             }  } |

일반적으로 함수 템플릿의 오버로딩은 해도 별 문제가 없지만,  
std는 조금 특별한 네임스페이스이기 때문에, 규칙도 다소 특별합니다.  
즉, std 내의 템플릿에 대한 완전 특수화는 OK이지만,   
std에 새로운 템플릿을 추가하는 것은 NOK입니다 (혹은 클래스이든 함수이든 어떤 것도 안됩니다).  
std에 들어가는 구성요소의 결정은 전적으로 C++ 표준화 위원회에 달려 있기 때문에, 금지하고 있는 것이죠.  
그런데, 그 ‘금지’하는 모양이 사람 마음을 불편하게 하는데, std의 영역을 침범하더라도 일단 컴파일까지는 거의 다 되고 실행도 됩니다. 그런데 실행되는 결과가 **미정의 사항**이라는 것입니다.

사용자들이 swap을 호출해서 우리만의 효율 좋은 ‘템플릿 전용 버전’을 쓰고 싶다.   
방법은, 멤버 swap을 호출하는 비멤버 swap을 선언해 놓되, 이 비멤버 함수를 std::swap의 특수화 버전이나 오버로딩 버전으로 선언하지만 않으면 됩니다.

예를 들어, Widget 관련 기능이 WidgetStuff 네임스페이스에 들어 있다고 가정하면, 다음과 같이 만들라는 것입니다.

|  |
| --- |
| namespace WidgetStuff {             ... // 템플릿으로 만들어진 WidgetImpl 및 기타등등.             template<typename T> // 이전과 마찬가지로 swap이란 이름의 멤버 함수가 들어 있습니다.             class Widget { ... };             ...  **template<typename T>** // 비멤버 swap 함수.  **void swap(Widget<T>& a, Widget<T>& b)** // std의 일부가 아님.  **{**  **a.swap(b);**  **}**  } |

이제는 어떤 코드가 두 Widget 객체에 대해 swap을 호출하더라도,   
컴파일러는 C++의 이름 탐색 규칙   
[이 규칙은 인자 기반 탐색(argument-dependent lookup) 혹은 쾨니그 탐색(Koenig lookup)이란 이름으로 알려져 있습니다] (어떤 함수에 어떤 타입의 인자가 있으면, 그 함수의 이름을 찾기 위해, 해당 타입의 인자가 위치한 네임스페이스 내부의 이름을 탐색해 들어간다는 간단한 규칙이다. ADL(Argument Dependent Lookup)이란 약자로 많이 불린다. ‘쾨니그’으이 Koenig는 C++ 표준화 위원회 임원이자, 이 규칙을 창안한 앤드류 쾨니그(Andrew Koenig)의 이름에서 따온 것이다)  
에 의해 WidgetStuff 네임스페이스 안에서 Widget 특수화 버전을 찾아냅니다.

이 간단한 방법은 클래스 템플릿뿐만 아니라, 클래스에 대해서도 잘 통합니다.  
클래스에 대해 std::swap을 특수화해야 할 이유가 생길 때,   
여러분이 만든 ‘클래스 타입 전용의 swap’이 되도록 많은 곳에서 호출되도록 만들고 싶으시면 (그리고 그런 swap을 갖고 있다면),   
그 클래스와 동일한 네임스페이스 안에, 비멤버 버전의 swap을 만들어 넣고,   
그와 동시에 std::swap의 특수화 버전도 준비해 둬야 합니다.

위의 모든 사항들은 여러분이 네임스페이스를 안 쓰고 있어도, 여전히 유효합니다 (멤버 swap을 호출하는 비멤버 swap이 필요하다는 뜻). 하지만 전역 네임스페이스(global namespace)를 못 잡아먹어서 안달복달할 필요가 있을까요?   
클래스, 템플릿, 함수, 나열자 타입, 나열자 상수, typedef 등의 온갖 이름을 전역 네임스페이스에 들이대면서 말이죠.

다음 함수 템플릿은 실행 중에 swap을 써서 2 객체의 값을 맞바꾸어야 한다고 가정합니다.

|  |
| --- |
| template<typename T>  void doSomething(T& obj1, T& obj2)  {             ...  **swap(obj1, obj2);**             ...  } |

이 부분에서 어떤 swap을 호출해야 할까요? 가능성은 3가지 입니다.

1.     Std에 있는 일반형 버전 (이것은 확실히 있습니다)

2.     Std의 일반형을 특수화한 버전 (있을 수도, 없을 수도 있습니다)

3.     T 타입 전용의 버전 (있거나 없거나 할 수 있으며, 어떤 네임스페이스안에 있거나 없거나 할 수 있습니다(하지만 확실히 std 안에는 없어야 하겠지요))

여러분은 타입 T 전용 버전이 있으면 그것이 호출되도록 하고,   
T 타입 전용 버전이 없으면 std의 일반형 버전이 호출되도록 만들고 싶습니다.  
어떻게 할 수 있을까요? 아래 코드가 답입니다.

|  |
| --- |
| template<typename T>  void doSomething(T& obj1, T& obj2)  {  **using std::swap;** // std::swap을 이 함수 안으로 끌어올 수 있도록 만드는 문장             ...             swap(obj1, obj2); // T 타입 전용의 swap을 호출합니다.             ...  } |

컴파일러가 위의 swap 호출문을 만났을 때 하는 일은, 현재의 상황에 딱 맞는 swap을 찾는 것입니다.  
C++의 이름 탐색 규칙에 따라, 우선 전역 유효범위 혹은 타입 T와 동일한 네임스페이스 안에 T전용의 swap이 있는지 찾습니다.   
[예를 들어, T가 WidgetStuff 네임스페이스 내의 Widget 이라면,   
컴파일러는 인자 의존 규칙을 적용하여 WidgetStuff의 swap을 찾아낼 것입니다]  
T 전용 swap이 없으면, 컴파일러는 그 다음 순서를 밟는데, 이 함수가 std::swap을 볼 수 있게 해 주는 using 선언(using declaration)이 함수 앞부분에 있기 때문에, std의 swap을 쓰게끔 결정할 수도 있습니다.  
하지만 이런 상황이 되더라도, 컴파일러는 std::swap의 T 전용 버전을, 일반형 템플릿보다 더 우선적으로 선택하도록 정해져 있기 때문에, T에 대한 std::swap의 특수화 버전이 이미 준비되어 있으면 결국 그 특수화 버전이 쓰이게 됩니다.

원하는 swap이 호출되도록 만드는 작업은 별로 어렵지 않습니다.  
이거 딱 하나만 조심하면 됩니다.   
호출문에 한정자를 잘못 붙이거나 하지는 마세요.  
한정자가 붙게 되면, C++가 호출될 함수를 결정하는 메커니즘에 바로 영향이 가기 때문입니다.

예를 들어, 위의 호출문을 아래와 같이 써버리면,

|  |
| --- |
| **std::**swap(obj1, obj2) // swap을 호출하는 잘못된 방법. |

Std의 swap(그 어떤 템플릿 특수화 버전들도 포함해서) 외의 다른 것은 거들떠보지도 말라고 컴파일러를 구속하게 됩니다. 더 딱 맞을 수 있는 T 전용 버전이 다른 곳에 있을 가능성을 완전히 무시하는 것이죠.  
클래스에 대해 std::swap을 완전히 특수화하는 게 중요한 이유가 바로 이것입니다.  
이렇게 해두면 잘못 한정화된 호출문으로도 타입 T 전용의 swap 함수를 끌어와 쓸 수 있기 때문입니다 (시중의 표준 라이브러리 중에도 이런 코드가 들어 있는 예가 꽤 있기 때문에, 이런 코드가 가능한 효율적으로 동작하는 데 도움을 주는 편이 여러분에게 이익이 됩니다)

표준 swap, 멤버 swap, 비멤버 swap, 특수화한 std::swap, swap호출시의 상황.  
정리,

첫째, 표준에서 제공하는 swap이, 여러분의 클래스 및 클래스 템플릿에 대해 납득할 만한 효율을 보이면, 그냥 아무것도 하지 말고 지내세요, 여러분이 만든 타입으로 만든 객체에 대해 ‘swap’을 시도하는 사용자 코드는 표준 swap을 호출하게 될 것입니다. 그리고 아무 문제도 없을 거고요.

둘째, 그런데 표준 swap의 효율이 기대한 만큼 충분하지 않다면 (여러분의 클래스 혹은 클래스 템플릿이 pimpl 관용구와 비슷하게 만들어져 있을 경우가 십중팔구입니다), 다음과 같이 하십시오

.

1.     여러분의 타입으로 만들어진 2객체의 값을 빛나게 빨리 맞바꾸는 함수를 swap이라는 이름으로 만들고, 이것을 public 멤버 함수로 두십시오. 단, 이 함수는 절대로 예외를 던져서는 안됩니다.

2.     여러분의 클래스 혹은 템플릿이 들어 있는 네임스페이스와 같은 네임스페에스에 비멤버 swap을 만들어 넣습니다. 그리고 1번에서 만든 swap 멤버 함수를 이 비멤버 함수가 호출하도록 만듭니다.

3.     새로운 클래스(클래스 템플릿이 아니라)를 만들고 있다면, 그 클래스에 대한 std::swap의 특수화 버전을 준비해 둡니다. 그리고 이 특수화 버전에서도 swap 멤버 함수를 호출하도록 만듭니다.

셋째, 사용자 입장에서 swap을 호출할 때, swap을 호출하는 함수가 std::swap을 볼 수 있도록 using 선언을 반드시 포함시킵니다. 그 다음에 swap을 호출하되, 네임스페이스 한정자를 붙이지 않도록 하십시오.

멤버 버전의 swap은 절대로 예외를 던지지 않도록 만들라고 했습니다.   
그 이유는 swap을 진짜 쓸모 있게 응용하는 방법들 중에   
클래스(및 클래스 템플릿)가 강력한 예외 안전성 보장(string exception-safety guarantee, 어떤 연산이 실행되다가 예외가 발생되면 그 연산이 시작되기 전의 상태로 돌릴 수 있다는 보장)을 제공하도록 도움을 주는 방법이 있기 때문입니다(29).  
그런데 이 기법은 멤버 버전 swap이 예외를 던지지 않아야 한다는 가정을 깔고 있습니다. 멤버 버전만 이렇습니다.  
비멤버 버전의 경우, 표준 swap은 복사 생성과 복사 대입에 기반하고 있는데,   
일반적으로 복사 생성 및 복사 대입 함수는 예외 발생이 허용되기 때문에 이런 제약을 받지 않습니다.  
따라서 swap을 직접 만들 분은 2값을 빠르게 바꾸는 방법만 구현하고 끝내면 안됩니다.   
예외를 던지지 않는 방법도 함께 준비하는 센스가 필요합니다.   
다행히 ‘효율’과 ‘예외 금지’의 2가지 특성은 함께 붙어 다니는 경우가 대부분입니다.  
효율이 대단히 좋은 swap 함수는 거의 항상 기본제공 타입(pimpl 관용구 기반의 설계에서 쓰이는 포인터처럼) 을 사용한 연산으로 만들어지기 때문입니다.   
그리고 기본제공 타입을 사용한 연산은 절대로 예외를 던지지 않거든요.

**요약.**

\* std::swap이 여러분의 타입에 대해 느리게 동작할 여지가 있다면 swap 멤버 함수를 제공합시다. 이 멤버 swap은 예외를 던지지 않도록 만듭시다.

\* 멤버 swap을 제공했으면, 이 멤버를 호출하는 비멤버 swap도 제공합니다. 클래스(템플릿이 아닌)에 대해서는, std::swap도 특수화해 둡시다.

\* 사용자 입장에서 swap을 호출할때는, std::swap에 대한 using 선언을 넣어 준 후에 네임스페이스 한정 없이 swap을 호출합시다.

\* 사용자 정의 타입에 대한 std 템플릿을 완전 특수화하는 것은 가능합니다. 그러나 std에 어떤 것이라도 새로 ‘추가’하려고 들지는 마십시오.