A 환경과 도구

안타깝게도 모든 개발 환경에서 완전히 똑같은 표준 라이브러리를 기대할 수는 없다. 더구나 비주얼 C++, g++ 등 널리 알려진 통합 개발 환경이라 할지라도 제공되는 표준 템플릿 라이브러리는 세부적인 내용이 서로 조금씩 다르다. 국제 표준을 통해 작업 기준을 만들기는 했지만, 몇 가지 이유에서 서로 다를 수 밖에 없기 때문이다. 그 원인을 살펴보면 다음과 같다.

■ 역사적인 문제

C++ 프로그램의 사용 역사보다 표준화된 역사가 짧다. C++ 언어는 뱐 스트라우스트럽(Bjarne Stroustrup)이 C 언어에 클래스 기능을 넣어 다양한 네트워크 모델을 적절히 표현할 수 있는 도구로 사용한 것이 시초였다. 그 이후 점차 C++ 언어가 널리 사용되고, 표준화가 이루어지면서부터 ARM, ANSI C++, CD1(Committee Draft), CD2, FDIS(Final Draft International Standard) 등 서로 다른 많은 기준이 존재했었고, 각각의 규정에 따라 모습이 조금씩 달랐다. 물론 시대에 따라 새로 C++ 언어에 첨가된 기술의 종류가 서로 다른 것이 이들 사이의 가장 큰 차이였다.

아직도 표준화 이전 시대에 만들어진 많은 상용 임플리멘테이션이 존재하고, 널리 사용되고 있다. 각급 학교에서 교육용으로 보급되어 90년대를 풍미했던 터보 C++가 그 대표적인 예다(사실 필자는 아직도 이 컴파일러가 각급 학교에서 학습용으로 사용되는 것에 놀라고 있고, 분명 컴파일러는 최신 것으로 반드시 바꾸어야 한다고 생각한다).

■ 표준화는 현재 진행형

C++ 언어는 표준화 과정을 포함해 표준화가 된 이후에도 끊임없이 진화하고 있다. 1998년 C++ 언어가 국제 표준으로서 완성되었지만, 일정상의 제약을 이유로 논의가 제대로 다뤄지지 않은 채 다음 표준화 시점으로 미루어졌고, 표준의 이곳 저곳에는 여전히 많은 오류를 내포하고 있다.

따라서 표준화 기구는 C++98에 담겨있는 오류 사항을 수정한 오류 정정표를 내고, 2003년 표준을 한 차례 개정했으며, 또 차후 표준화에서 다뤄질 신기술 목록인 TR1을 발표하기도 하였다. 이에 따라 현재 표준을 준수하고, 표준화에 빠르게 대응하는 소수의 임플리멘테이션에서는 이미 개정된 내용을 바탕으로 임플리멘테이션을 수정하거나 기능을 추가함으로써 역시 서로 다른 표준 라이브러리가 존재할 수 있게 되었다. 예를 들어, Boost는 이미 차후 표준에서 다루어질 신기술을 지원하고 있다.

■ 임플리멘테이션 정의 사안의 존재

'임플리멘테이션에 구현의 자유를' 이라는 원칙에 입각하여 표준 라이브러리에서도 많은 부분들이 임플리멘테이션의 정의 사안으로 남아 있다. 그 결과 동일한 버전의 표준을 참고하여 만들었다 하여도 모든 임플리멘테이션이 완벽하게 같지는 않다.

예를 들어, 연관 컨테이너 map 등에서 원소의 정렬을 위해 사용되는 데이터 구조가 어떤 것이어야 할지 표준에서는 전혀 규정하지 않았다. 다만 연산이 이루어지는 구간에 비례하는 추상적인 복잡도와 결과값만을 기

술하고 있고, 이에 따라 임플리멘테이션에서는 규정을 만족하는 최소한의 것 이외에 다양한 내부 관리 알고리즘을 사용할 수 있다. 이 때문에 대다수의 임플리멘테이션에서는 원소 관리를 위해 빨강-검정 트리(Red-black Tree)를 사용하지만, 일부에서는 다른 형태의 트리를 사용할 수도 있다.

■ 임플리멘테이션 확장

표준 라이브러리의 정의는 임플리멘테이션 차원에서 확장할 수 있는 여지를 많이 남겨놓고 있다. 따라서 표준에서는 전혀 규정하지 않지만, 해시 등의 새로운 종류의 컨테이너를 제공할 수도 있다. 물론 해시는 표준이 아니기 때문에 모든 임플리멘테이션에서 제공될 것이라는 보장은 없다. 또 다른 중요한 예로는, 기본 인자 등의 방법을 통해 임플리멘테이션에서 추가 인자를 두는 것에 대해 전혀 규제하지 않고 있다. 다음 예를 보자.

```
#include<set>
#include<functional>
#include<algorithm>
#include<string>
#include<iostream>
...
std::set<std::string> s;
std::transform(s.begin(), s.end(),
  std::ostream_iterator<std::string::size_type>(std::cout, " "),
  std::mem_fun_ref(&std::string::size));
```

스콧 마이어스(Scott Meyers)가 문제를 제기하기 위해 제시했던 본 예제는 set에 담겨 있는 모든 string의 크기를 출력하고자 set의 멤버 함수 size의 포인터를 사용하려고 시도하였다.

이 프로그램에서 mem_fun_ref가 요구하는 멤버 함수에 대한 포인터 타입은 R (T::*m)( )로, 인자가 없어야 한다. 그러나 임플리멘테이션에 따라서 size에는 기본 인자가 추가될 수 있기 때문에 mem_fun_ref의 요구 조건을 만족하지 않을 수도 있다. 실제로 이 프로그램은 어떤 환경에서는 번역에 성공하지만, 어떤 환경에서는 실패한다. 좀더 일반화하면 이 문제로 인해 표준 라이브러리에서 제공하는 함수에서 포인터를 얻어오는 행위가 보장되지 않음을 의미한다.

지금까지 살펴본 것처럼 같은 표준 라이브러리라 하여도 임플리멘테이션에 따라 다양한 변종이 존재할 수 있기 때문에 자신이 사용하는 번역 환경에 어떠한 특성이 있는지 알아보는 것은 중요하다. 여기에서는 이와 같은 주제들에 대해 살펴볼 것이다. 다시 강조하지만 지금부터 다룰 내용들은 상당 부분 임플리멘테이션에 의존하므로 섣불리 일반화시켜 이해해서는 곤란하다. '부록 A' 에서는 아래와 같은 사안들에 대해 살펴본다.

- 윈도우 DLL과 표준 템플릿 라이브러리
- 템플릿 라이브러리

❶ 윈도우 DLL과 표준 템플릿 라이브러리

DLL(Dynamic-Linked Library)은 실행 파일의 크기를 줄이기 위해 실행 중에 엔티티에 연결할 수 있게 해주는 라이브러리의 일종이다. DLL을 이용하면 실행 파일의 크기가 줄어들고, 프로그램을 물리적으로 나누어 관리할 수 있다는 장점이 있어 흔히 윈도우 환경의 프로그램에 사용되는 모듈을 개발할 때 이용된다.

하지만 표준 템플릿 라이브러리를 사용하려는 개발자의 입장에서 본다면, DLL은 문제를 일으킬 소지가 있다. 다음 프로그램을 비주얼 C++에서 DLL용으로 컴파일해 보자.

```
#include<vector>

#if !defined(DLLEXPORT_EXPORTS)
#   define DLLEXPORT_API __declspec(dllexport)
#else
#   define DLLEXPORT_API __declspec(dllimport)
#endif

class DLLEXPORT_API CDLLexport
{
public:
    std::vector<char> vs;
};
```

위 프로그램은 클래스 CDLLexport를 DLL에서 사용할 수 있도록 만든 것이다. 여기서 매크로 DLLEXPORT_API는 DLL에서 클래스, 함수 등을 외부에 export시키기 위해 사용되는 것으로, export용으로 컴파일할 때는 __declspec(dllexport)로, import용으로 컴파일할 때는 __declspec(dllimport)로 정의하게 조절하면 된다. 이들은 함수나 클래스를 선언할 때 선두에 적용되어 해당 함수나 개체가 DLL을 통한 외부 인터페이스를 갖는 것을 의미한다.

일반적으로 DLL을 사용하는 방식대로 작성한 프로그램이지만, 위 프로그램을 컴파일하면 다음과 같은 경고를 내는 것을 볼 수 있다.

```
warning C4251: 'vs' : class 'std::vector< char,class
std::allocator<char> >' needs to have dll-interface to be used by
clients of class 'CDLLexport'
warning C4231: nonstandard extension used : 'extern' before template
explicit instantiation
```

경고는 에러가 아니라고 생각하고 무시하고 넘어가기로 하고, 만들어진 DLL를 사용하는 실행 파일을 만들어 CDllexport::vs를 직접 사용해 보자. 하지만 사용 중에 'Access Violation'과 같은 에러를 내면서 프로그램이 종료되는 등의 문제를 일으킨다. 무언가 잘못된 것이다. 그렇다면 번역 도중 산출되었

던 경고 메시지를 살펴보고, 해결해 보자.

먼저 첫번째 경고를 살펴보자. 이 경고는 클래스 CDLLexport에서 사용된 vs라는 개체의 클래스가 DLL 인터페이스를 가져야 한다고 말해주고 있다. 즉 이 경고에 따르면 CDLLexport::vs는 DLL에 해당하는 인터페이스가 없기 때문에 외부에서 사용할 수 없다는 뜻이다. 분명 클래스를 DLL 인터페이스로 지정하였는데 어째서 CDllexport::vs는 문제가 발생하는 것일까? 일반적인 클래스나 함수, 개체를 DLL 인터페이스로 지정하는 것은 위와 같은 선언이면 충분하지만, 템플릿을 DLL 인터페이스로 지정하기 위해서는 좀더 복잡한 과정이 필요하기 때문이다.

템플릿은 DLL 인터페이스로 지정되기 전에 먼저 템플릿을 인스턴스화하고, 이 인스턴스가 DLL 인터페이스를 갖는다고 지정해야 한다. 앞의 예를 다음과 같이 수정해 보자.

```
#include<vector>

#if !defined(DLLEXPORT_EXPORTS)
#   define DLLEXPORT_API __declspec(dllexport)
#   define DLLEXPORT_TEMPLATE
#else
#   define DLLEXPORT_API __declspec(dllimport)
#   define DLLEXPORT_TEMPLATE extern
#endif

DLLEXPORT_TEMPLATE template class DLLEXPORT_API std::vector<char>;

class DLLEXPORT_API CDLLexport
{
public:
    std::vector<char> vs;
};
```

다시 컴파일해 보자. 자, C4251은 사라졌다. 먼저 사용할 컨테이너 vector의 완전한 인터페이스를 인스턴스화하고, DLL 인터페이스를 지정함으로써 더 이상 경고가 발생하지 않는다. 여기서 사용된 키워드 extern(DLLEXPORT_TEMPLATE)은 이 파일이 DLL용으로 컴파일될 때 템플릿 클래스를 명시적으로 인스턴스화할 것을 명령하는 역할을 한다.

이런 방법을 사용한다고 해도 DLL에서 컨테이너를 export하는 것은 vector를 제외하면 불가능하다. 인스턴스화와 동시에 인터페이스를 지정하는 도중 체인이 발생하여 더 이상 인스턴스화를 진행할 수 없기 때문이다. 다시 아래 예를 보자. 이번에는 vector가 아닌 deque을 export하고자 한다.

```
#include<deque>

#if !defined(DLLEXPORT_EXPORTS)
#   define DLLEXPORT_API __declspec(dllexport)
```

```
#   define DLLEXPORT_TEMPLATE
#else
#   define DLLEXPORT_API __declspec(dllimport)
#   define DLLEXPORT_TEMPLATE extern
#endif

DLLEXPORT_TEMPLATE template class DLLEXPORT_API std::deque<char>;

class DLLEXPORT_API CDLLexport
{
public:
    std::deque<char> vs;
};
```

같은 방법을 사용했음에도 불구하고 여전히 다음과 같이 C4251 경고가 나온다.

```
warning C4251: '_First' : class 'std::deque<char,class
std::allocator<char> >::iterator' needs to have dll-interface to be
used by clients of class 'std::deque<char,class std::allocator<char> >'
```

위 경고는 deque의 내부에서 정의되는 iterator가 DLL 인터페이스를 가져야 한다고 말해주고 있다. 이는 deque을 DLL 인터페이스로 지정하면서 deque 내부에 정의되어 있는 iterator 클래스 역시 DLL 인터페이스를 가지기를 요구하기 때문이다. 하지만 iterator를 먼저 선언한다고 해도 문제는 생긴다. 다시 컴파일을 시도하면 이번엔 deque이 인터페이스를 가져야 한다고 경고할 것이다.

이 문제는 비주얼 C++에서 DLL 인터페이스를 지원하는 능력의 한계 때문에 발생한다. 템플릿을 DLL 인터페이스로 지정할 때는 export할 템플릿과 관계된 모든 템플릿의 export 지정 역시 필요하기 때문이다. 그리고 이 경우에 deque은 iterator를 필요로 하고, iterator는 deque을 필요로 하기 때문에 인터페이스 지정 시 체인이 발생되어 더 이상 진행할 수 없게 된다. 결과적으로 모든 종류의 내장된 클래스를 사용하는 템플릿은 export될 수 없게 된다. 그럼에도 불구하고 vector가 export할 수 있는 이유는 비주얼 C++가 vector를 가장 많이 사용되고 있어서 vector에 대해서만 가능하도록 특별히 만들었기 때문이다.

이제 두 번째 경고를 살펴보자. 두 번째 경고는 먼저 살펴보았던 경고보다는 한층 수준이 낮은 경고라고 할 수 있다. 이 경고는 단지 extern을 템플릿에 지정한 것이 올바른 사용인지 확인하는 것이다. extern으로 템플릿을 명시적으로 인스턴스화하는 것은 비주얼 C++에서만 지원되는 확장 기능이기에 정말로 사용자가 의도적으로 지정한 것인지 확인하는 차원의 경고 문구다. 하지만 이는 원래 의도한 것이므로 무시하자. 만일 경고를 아예 끄고 싶다면 프로그램의 최상단에 다음과 같이 적어주면 된다.

```
#pragma warning(disable: 4231)
```

비록 DLL에서 외부로 컨테이너를 export할 수 없다 하더라도 DLL 내부에서만 운용되는 용도로는 얼마든지 사용할 수 있기 때문에 DLL에서 표준 템플릿 라이브러리가 전혀 쓸모 없는 것은 아니다. 즉 다음과 같은 경우에는 경고가 발생하지만, 문제가 되지는 않는다.

```
#include<deque>
#include<algorithm>

#if !defined(DLLEXPORT_EXPORTS)
#   define DLLEXPORT_API __declspec(dllexport)
#else
#   define DLLEXPORT_API __declspec(dllimport)
#endif

void print_char(char c);

class DLLEXPORT_API CDLLexport
{
public:
    void add(char c) {  vs.push_back(c); }
    void print()
    {
      std::for_each( vs.begin(), vs.end(), print_char );
    }
private:
    std::deque<char> vs;
};
```

이 때문에 비록 DLL의 인터페이스에서 컨테이너를 직접 사용할 수 없다 해도 래퍼(wrapper) 인터페이스를 제공하는 형식으로 여전히 유용하게 사용될 수 있다.

❷ 템플릿 라이브러리

표준 템플릿 라이브러리에도 여러 종류가 있다. 대표적인 것을 살펴보면 [표 A-1]과 같다.

[표 A-1] 다양한 표준 라이브러리

라이브러리	위치	특징
Dinkum Standard Template Library	http://www.dinkumware.com	비주얼 C++에 탑재되는 표준 템플릿 라이브러리
SGI STL	http://www.sgi.com/tech/stl	표준 템플릿 라이브러리를 최초로 구현한 HP의 버전 계승
STLport	http://www.stlport.org/	널리 사용되는 안정성이 높은 STL 라이브러리

RogueWave Standard C++ Library	http://www.roguewave.com/	UNIX 기반의 컴파일러 및 볼랜드 C++에 표준 템플릿 라이브러리 제공
ObjectSpace	http://www.recursionsw.com	
Safe STL	http://www.horstmann.com/safestl.html	안정성이 강화된 표준 템플릿 라이브러리

이외에도 템플릿 기반으로 이루어져 있는 강력한 라이브러리들이 있다. 이들은 특히 과학 계산에 초점을 맞추고 있으며, 일부는 개체 패턴과 메타 프로그래밍, 함수형 라이브러리 등 다양한 주제로 설계되었다.

[표 A-2] 다양한 템플릿 기반 라이브러리

라이브러리	위치	특징
Blitz++	http://www.oonumerics.org/blitz/	수치 계산, 과학 계산용 템플릿 라이브러리
FC++	http://www.cc.gatech.edu/~yannis/fc++/	함수형 프로그래밍 라이브러리
Boost	http://www.boost.org/	표준화가 목적인, 혹은 표준에서 제외되었지만 표준에 가장 가까운 라이브러리. 실제로 Boost는 TR1에 새로운 기능을 가장 많이 제안하였고, 받아들여졌다.
Loki	http://sourceforge.net/projects/loki-lib/	디자인 패턴 라이브러리. 표준 템플릿 라이브러리의 제안자 알렉산더 알렉산드레스쿠가 제작
The Matrix Template Library	http://www.osl.iu.edu/research/mtl/	다양한 형태의 매트릭스 연산을 지원하는 선형 대수 라이브러리
NTL	http://www.shoup.net/ntl/	수치 계산 라이브러리
STL soft	http://synesis.com.au/stlsoft/	

이 밖에도 많은 수의 수치 전문 라이브러리, 병렬화를 이용한 고속 과학 계산 라이브러리, 신경망, 진화, 기계 학습, 데이터 마이닝 등을 위한 인공지능 라이브러리, 물리, 양자 라이브러리, 미적분 전문 라이브러리, 그래프, 조합 라이브러리 등 C++ 템플릿 기술을 응용해 제작된 라이브러리는 수 없이 많다.