서문(Foreword)

정말 주목할만한 책입니다. 하지만, 읽기를 거의 끝냈을 때까지 얼마나 뛰어난 책인지 깨닫지 못했죠. 이미 C++, C++의 모든 것에 친숙한 사람들을 위해서 쓰여진 책들 중 으뜸이 될 수 있으리라 생각합니다. 이 책은 언어 특성에서부터 프로그래밍 기법과 표준 라이브러리의 구성요소까지, 항상 약간의 균형을 유지하며 여러분들의 관심을 끌면서, 그 주제 사이를 넘나들고 있습니다. 마치 실제 C++ 프로그램들처럼 말이죠. 클래스 디자인은 가상 함수들의 행동과 마주치게 되고, 반복자(iterator) 변환은 이름 검색 규칙과 대응하여 만나게 되며, 할당 연산자는 예외 안전성을 스쳐 지나가고 있고, 컴파일 의존성은 밖으로 표출된 템플릿을 가로질러 가기도 합니다. 마치 실제 프로그램들에서처럼 그렇게... 그 결과는, 무질서하고 방대한 상황에서 언어 특성과 라이브러리 컴포넌트, 그리고 프로그래밍 기법들의 어지러운 커다란 소용돌이입니다. 마치 실제 프로그램들처럼...

본인은 *GotW*를 "Gotcha"와 같이 운을 달아 발음합니다. 그리고 아마 그게 적당한 것 같고요. 이 책의 질문들에 대한 Sutter의 답들과 본인의 해결책을 비교해 보면, 스스로 인정하기보다는 그가 (그리고 C++가) 설치해 놓은 함정에 빠져드는 것 같습니다. 본인이 각 오류를 만들 때마다, Herb가 웃음을 지으며 살며시 "Gotcha!"를 외치는 것을 보는 듯 합니다. 어느 항목들을 보면, '내가 C++에 대해 많이 알지 못하는구나'하는 생각이 듭니다. 다른 것들은, C++가 어느 누군가에게 터득하도록 설명하는 것이 무척 복잡하다는 사실을 말하기도 합니다. 여러분이 C++로 작업을 할 때, 무엇을 하고 있는지 신중히 생각해야 한다는 것을 보여줍니다. C++는 큰 노력이 필요한 문제들을 해결하는데 도움을 주도록 설계된 강력한 언어입니다. 따라서, 언어와 그 라이브러리, 프로그래밍 이디엄들의 지식을, 가능한 아주 정밀하게 연마하는 것은 매우 중요한 일입니다. 이 책 속에 있는 주제들의 깊이는 여러분들이 그렇게 하도록 도와줄 것입니다. 그만큼 독특한 퀴즈 기반의 형태를 이루고 있죠.

C++ 뉴스그룹의 경험 많은 독자들은 *Guru of the Week*로 선포되는 것이 얼마나 어려운 것인지 알고 있을 것입니다. 노련한 참가자들은 심지어 경의를 표하기도 합니다. 재미있게도, 물론 각 주에 한 명만 그루(guru, 권위자)가 될 수 있지만, 이 책의 정보를 다루다 보면, 여러분도 당연히 프로그램을 작성하는 모든 시간에 마치 그루가 된 듯이 작성할 수 있습니다.

Scott Meyers

머리말(Preface)

*Exceptional C++*는 예제를 통하여 견고한 소프트웨어 엔지니어링을 어떻게 실현하는가를 보여주고 있습니다. 다른 많은 요소들과 함께 이 책에서는, 완비된 C++ 엔지니어링 문제들과 특정 디자인, 코딩 기법들을 설명하고 있는 해결책들의 연재물로 이루어진, 유명한 Internet C++ feature *Guru of the Week*(혹은, 간단한 형태로 *GotW*)의 처음 30개의 이슈들의 확장된 형태를 포함하고 있습니다.

이 책은 코드 문제들을 아무렇게나 모아놓은 것이 아닙니다. 완전한 실제 세계에서 C++로 거대한 소프트웨어 디자인을 하기 위한 근본적인 지침서입니다. 효율적으로 독자 여러분들을 참여하도록 유도하기 위해서 문제 뒤에 아이디어를 제공하고, 가이드라인 뒤에 그 이유를 설명하는 방식으로 효율성을 높이기 위한 문제/해결책 형태를 사용하였습니다. 항목들이 다양한 주제를 포함하고 있다 하더라도, 예외 안전성, 완전한 클래스와 모듈 디자인, 적절한 최적화, 그리고 이식 가능한 표준에 따르는 코드 작성과 같은 개발 이슈들에 초점을 맞추고 진행되어 가는 것을 알 수 있습니다.

여러분들의 일상 작업에서 유용하게 사용될 수 있는 재료들을 찾아가기를 희망합니다. 또한, 최소한 재치 있는 생각들과 우아한 기법들을 발견하고, 가끔씩 페이지를 넘길 때마다 "아하! 훌륭해"라고 갑자기 외칠 수 있기를 희망합니다. 결국, 소프트웨어 엔지니어링의 매력에 흠뻑 취하게 될 것입니다.

어떻게 이 책을 읽을 것인가?

여러분들이 이미 C++의 기본들은 알고 있으리라 생각합니다. 그렇지 않다면, 좋은 C++ 입문서로 먼저 시작하여 전체 개념을 잡고 (Bjarne Stroustrup의 *The C++ Programming Language, Third Edition*[1]이나 Stan Lippman과 Josee Lajoie의 *C++ Primer, Third Edition*[2]과 같은 고서들이 좋은 선택이 될 수 있습니다), 그 다음에 Scott Meyers의 대표 저서 *Effective C++*와 같은 스타일 지침서를 선택하는 것이 좋습니다(브라우저 기반의 CD 버전이 편리하고 유용합니다)[3]

1) Stroustrup B. *The C++ Programming Language, Third Edition*(Addison Wesley Longman, 1997)
2) Lippman S. and Lajoie J. *C++ Primer, Third Edition*(Addison Wesley Longman, 1998)
3) Meyers S. *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*(Addison Wesley Longman, 1999). 온라인 데모는 http://www.meyerscd.awl.com에서 보실 수 있습니다.

이 책의 각 항목들은 다음과 같은 도입부를 통해 퍼즐 혹은 문제처럼 소개됩니다.

항목 ## : 이번 문제의 주제 (난이도 : X)

주제와 난이도(10까지의 범위 중에서, 보통 3에서 9½까지 분포되어 있음)를 통해 무엇을 하고 있는지 간략하게 짐작할 수 있습니다. 난이도는 필자가 대부분의 사람들이 각 문제들에 대해 얼마나 어렵게 느끼는지 주관적인 관점에서 표시한 것입니다. 따라서 난이도 7의 문제가 난이도 5의 문제보다 더 쉽게 느껴질 수도 있습니다. 9½과 같은 벅찬 주제들은 심호흡을 가다듬고 미리 준비하는 것이 좋겠죠.

순서대로 각 절들과 문제들을 읽을 필요는 없습니다. 하지만 "파트 1", "파트 2" 등과 같이 표시된 관련된 문제에 대한 "연재물"은 하나로 묶어서 읽는 것이 좋습니다.

이 책은 많은 가이드라인을 포함하고 있습니다. 그 안에서 나오는 다음 단어들을 주의해서 보기 바랍니다.

- 항상 ... 하다 = 절대적으로 필요한 것입니다. 이렇게 하지 않으면 안 됩니다.
- 선호하다 = 올바른 방법을 표시합니다. 정당한 이유가 있는 특수한 경우에만 다른 선택을 하는 것이 좋습니다.
- 고려하다 = 적용해도 되고 하지 않아도 됩니다. 하지만 충분히 생각해야 합니다.
- 회피하다 = 보통 가장 좋은 방법은 아니며 심지어 위험할 수도 있습니다. 대안을 찾되, 정당한 이유가 있는 특수한 경우에만 사용합니다.
- 절대 ... 하지 않는다 = 완전히 잘못된 경우입니다. 생각조차 하지 않는 것이 좋습니다.

마지막으로, URL에 대해 말해보면, 웹에서 이동하는 것입니다. 특별히, 이동을 방해할 수도 없는 것이죠. 출판된 책에서 언급한 임의적인 웹 URL이 5년 이상 한 자리에 머물 것이라고 생각할 수는 없습니다. 이 책에서 다른 사람들의 기사 혹은 웹 사이트를 참조하였을 때, 이들 주소를 필자의 웹 사이트 www.gotw.ca의 URL을 통해 실제 웹 페이지로 갈 수 있도록 하였습니다. 이 책에 인쇄된 링크가 더 이상 동작하지 않는다면, 필자에게 E-메일을 보내주세요. (다시 그 페이지를 찾을 수 있다면) 변경된 새 페이지의 주소를 갱신하거나 (찾을 수 없다면) 존재하지 않는다고 표시해 놓겠습니다. 어느 방법으로든, 이 책의 URL을 최신으로 유지할 것입니다. 휴.

*GotW*와 PeerDirect

C++ Guru of the Week 연재물은 오랫동안 지속되어 왔습니다. *GotW*는 원래, 1996년 말에 흥미로운 도전들과 PeerDirect에서 우리 자신들의 개발팀을 위해 진행되는 교육을 제공하려는 목적으로 만들어졌습니다. 상속과 예외 안전성의 적절한 사용법과 같은 것을 퍼뜨리려는 것을 포함하여, 재미있는 교육 도구들을 제공하기 위해 만들었습니다. 시간이 흘러, C++ 표준 회의에서 만들어진 변화들을 가시적으로 팀에 조공하는 수단으로 사용하기도 하였습니다. 그 이후, *GotW*는 정기적으로 새로운 질문들과 대답(그리고 많은 재미있는 논의들)을 찾을 수 있는 인터넷 뉴스그룹 *com.lang.c++.moderated*의 모습들을 활용하여 일반적인 C++ 공공 장소로 만들어졌습니다.

C++를 잘 이용하는 것은 각각 다른 목적을 성취하기 위한 것이겠지만, 여러분들의 회사에서 중요하다고 생각되는 많은 이유들은 PeerDirect에서도 똑같이 중요합니다. 신뢰성, 안전성, 이식성, 효율성, 그리고 많은 다른 이슈들이 만들어지거나 사라지는 것을 염려하여(데이터베이스를 배포하고 복제하기 위하여) 시스템 소프트웨어를 만들게 되었습니다. 이 소프트웨어는 다양한 컴파일러와 운영체제에 이식될 수 있도록 만들어야 합니다. 데이터베이스 트랜잭션과 인터럽트 통신, 프로그래밍 예외가 존재하더라도 안전하고 견고해야 합니다. 또한, 스마트 카드 혹은 PalmOS와 WinCE 장치, Windows NT와 Linux 그리고 Solaris 서버들, 웹 서버들과 같은 소프트웨어를 이용한 데이터 웨어하우스를 위한 대규모의 병렬 Oracle 백-엔드에서 같은 신뢰성, 같은 코드를 이용하여 작은 데이터베이스 착석을 고객이 관리하도록 사용됩니다. 이제, 코드의 주석을 제외한 거대한 50만 라인에서 이식성과 신뢰성에 도전장을 내밀어야 합니다.

지난 몇 년간 인터넷에서 *Guru of the Week*을 읽어온 독자들에게 다음과 같이 말하고자 합니다.

- 여러분의 관심, 지원, E-메일, 칭찬, 정정, 주석, 비평, 질문에 대해 감사드립니다. 그리고 특히 *GotW* 연재물이 이 책으로 녹아 들어가도록 요청해 주신 것에 깊은 감사를 드립니다. 자, 여기에 있습니다. 즐겁게 봐주시기를 희망합니다.
- 이 책은 인터넷에서 보았던 것보다 더 많은 것을 포함하고 있습니다.

*Exceptional C++*은 이미 사이버 공간 어디선가 드러났던 진부한 *GotW* 이슈들을 잘라 붙인 것이 아닙니다. 모든 문제들과 해결책들은 주의 깊게 교정되고 재작업되었습니다. 예를 들어, 원래 하나의 *GotW* 퍼즐로 등장한 것을 항목 8에서 17을 통해 더 깊이, 10개 파트에 걸쳐 살펴보고 있습니다. 각 문제들과 해결책은 변경되고, 이제 공식의 C++ 표준안으로 새롭게 검사하였습니다.

따라서, 전에 *GotW*의 정기 독자였더라도, 새로운 것이 많이 있을 것입니다. 모든 성실한 독자들에게 다시 감사를 드리며, 이 재료들이 여러분의 소프트웨어 엔지니어링과 C++ 프로그래밍 기술들을 연마하는데 지속적인 도움이 되었으면 합니다.

감사의 글

먼저, 당연히 모든 *GotW* 독자들과 *comp.lang.c++.moderated*의 열성 팬들, 특히 이 책의 이름을 선정하는 대회에 참여해 준 많은 사람들에게 감사를 드립니다. 특별히 마지막 제목으로 선정하는데 도움이 되었던 두 사람, *Enlightened C++*이라는 이름을 제안한 Marco Dalla Gasperina씨와 *Practical C++ Problems and Solutions*를 제안한 Rob Stewart씨에게 감사를 드립니다. 자연스럽고 재미를 가미한 *exceptional*은 지속적으로 이 글의 예외 안전성을 강조하고 있습니다.

연재물 편집자인 Bjarne Stroustrup과 Marina Lang, Debbie Lafferty, 그리고 Addison Wesley Longman 편집자 여러분들의 이 프로젝트에 대한 지속적인 관심과 열정, 그리고 1998년 Santa Cruz의 C++ 표준안 회의에서의 정말로 유쾌한 허락에 많은 감사를 드립니다.

또한, 검토해 주신 많은 사람들(여러분들이 읽을 문서들을 개선하는데 도움을 준 깊은 통찰력과 예리한 의견을 제공한 표준 위원회 멤버들)에게도 감사를 드리고 싶습니다. 특별히 정말로 가치있는 통찰력과 평론들을 해준 Bjarne Stroustrup과 Scott Meyers, 그리고 Andrei Alexandrescu, Steve Clamage, Steve Dewhurst, Cay Horstmann, Jim Hyslop, Brendan Kehoe, 그리고 Dennis Mancl에게 감사를 드립니다.

마지막으로, 항상 다른 많은 방식으로 지켜봐 주었던 나의 가족들과 친구들에게 감사를 드립니다.

Herb Sutter

일반 프로그래밍과 C++ 표준 라이브러리

항목 1 : 반복자(Iterator)
항목 2 : 대·소문자 구분 없는 문자열 - 파트 1
항목 3 : 대·소문자 구분 없는 문자열 - 파트 2
항목 4 : 최대 재사용 가능한 일반 컨테이너 - 파트 1
항목 5 : 최대 재사용 가능한 일반 컨테이너 - 파트 2
항목 6 : 임시 개체들
항목 7 : 표준 라이브러리의 사용(혹은, 다시 보는 임시 개체)

시작하기 전에, 일반적인 프로그래밍 영역에서 몇 가지 주제를 선택하여 생각해 봅시다. 이런 문제들은 템플릿(template), 반복자(iterator), 그리고 알고리즘의 효과적인 사용법과 표준 라이브러리의 기능을 확장하여 사용하는 방법에 초점이 맞추어져 있습니다. 그리고, 예외에 안전한 템플릿을 작성해야 하므로, 자신도 모르게 안전한 예외 처리를 분석하는 다음 절로 관심이 가게 될 것입니다.

항목 1 : 반복자(Iterator) (난이도 7)

표준 라이브러리를 사용하는 모든 프로그래머들은 이런 일반적이며 동시에 그렇게 일반적이지 않은 반복자에 관한 실수를 주의해야 합니다. 여러분은 아래에서 몇 개나 실수를 찾을 수 있나요?

다음의 프로그램은 적어도 4개의 반복자와 관련된 문제를 가지고 있습니다. 여러분의 눈에는 얼마나 보이나요?

```
int main()
{
    vector<Date> e;
    copy( istream_iterator<Date>( cin ),
          istream_iterator<Date>(),
          back_inserter( e ) );
    vector<Date>::iterator first =
           find( e.begin(), e.end(), "01/01/95" );
    vector<Date>::iterator last =
         find( e.begin(), e.end(), "12/31/95" );
    *last = "12/30/95";
    copy( first,
          last,
          ostream_iterator<Date>( cout, "\n" ) );
    e.insert( --e.end(), TodaysDate() );
    copy( first,
          last,
          ostream_iterator<Date>( cout, "\n" ) );
}
```

해결책

```
int main()
{
    vector<Date> e;
```

```
copy( istream_iterator<Date>( cin ),
      istream_iterator<Date>(),
      back_inserter( e ) );
```

여기까지는 괜찮습니다. Date 클래스 작성자는 cin 스트림에서 Date를 읽어들이는데 사용하는 인쇄 operator >>(iostream&, Data&)를 추출할 수 있는 함수를 제공합니다. copy( ) 알고리즘은 단지 Date를 벡터(vector)에 넣는 역할을 합니다.

```
vector<Date>::iterator first =
     find( e.begin(), e.end(), "01/01/95" );
vector<Date>::iterator last =
     find( e.begin(), e.end(), "12/31/95" );
*last = "12/30/95";
```

오류 : 문제가 좀 있는 듯 보이죠. last는 e.end( )가 될 수도 있는데, 역참조(dereference)가 가능한 반복자가 아니기 때문입니다.

find( ) 알고리즘은 값을 찾지 못했을 경우, 두 번째 매개변수(범위 내의 마지막 반복자)를 반환합니다. 이 경우에, "12/31/95"가 e에 없다면, last는 컨테이너(container)의 끝을 하나 지난 위치인, 유효하지 않은 e.end( )를 가리키게 됩니다.

```
copy( first,
      last,
      ostream_iterator<Date>( cout, "\n" ) );
```

오류 : 이것 또한 [first,last)가 올바르지 않은 범위가 될 수 있기 때문에 이상한 것 같습니다. 게다가, first가 실제로 last의 뒤가 될 수도 있는 것 같습니다.

예를 들어, e에서 "01/01/95"는 발견하지 못하고 "12/31/95"는 발견했다면, last 반복자는 집합 내의 어떤 항목("12/31/95"를 가리키는 Date 개체)을 가리키게 되는 반면, first 반복자는 끝의 다음 항목을 가리키게 됩니다. 하지만, copy( )는 같은 집합에서 last보다 first가 이전 위치를 가리켜야 합니다. 다시 말해서, [first, last)가 유효한 범위가 되어야 합니다.

이런 문제들의 일부를 탐지하는 표준 라이브러리 버전을 사용하고 있지 않다면, copy( ) 이후에 문제를 일으키는 이런 종류의 증상들을 발견하는 것은 어렵습니다.

```
e.insert( --e.end(), TodaysDate() );
```

첫 번째 오류 : "--e.end( )" 표현식은 잘못된 듯 보이네요.

이유는 정말 간단한데, 약간 불분명하기 때문이죠. 표준 라이브러리의 일반적인 구현 방법에서 vector<Date>::iterator는 종종 간단히 Date*로 바꿀 수 있으며, C++ 언어에서는 내장 형식의 일시적인 수정은 허락하지 않습니다. 예를 들어, 다음과 같은 간단한 코드 역시 잘못되었습니다.

```
Date* f();          // Date*를 반환하는 함수
p = --f();          // 오류, 하지만 "f() -1"은 가능하다.
```

운 좋게도, vector<Date>::iterator가 임의 접근(random-access) 반복자라는 것은 이미 알고 있습니다. 그래서, 다음과 같이 효율을 떨어뜨리지 않고 올바르게 작성하면 됩니다.

```
e.insert( e.end() -1, TodaysDate() );
```

두 번째 오류 : 아직 다른 오류가 숨어 있습니다. e가 비어 있다면, "e.end( ) 이전의 반복자"를 가져오는 어떤 것들도("--e.end( )" 또는 "e.end( )-1", 어떤 것을 사용하던지 간에) 유효한 반복자가 될 수 없습니다.

```
    copy( first,
          last,
          ostream_iterator<Date>( cout, "\n" ) );
}
```

오류 : first와 last는 더 이상 유효한 반복자가 아닙니다.

벡터(vector)는 무엇을 삽입할 때마다 버퍼를 다시 할당하는 것을 피하기 위해 "덩어리"로 커져 갑니다. 하지만, 가끔 벡터가 꽉 차게 되어 어떤 것을 추가하려면 다시 할당을 해야 합니다.

여기에 e.insert( )의 결과로, 벡터가 커지거나 커지지 않는 것은 메모리가 이동하거나 이동하지 않는다는 것을 의미합니다. 이런 불확실성 때문에, 반드시 어떤 존재하는 반복자가 무효화(역자 주 invalidate, 어떤 변화에 의해서 데이터가 갱신되는 것을 뜻함)된 컨테이너로 들어갔는지 주의해야 합니다. 이 경우, 메모리가 실제로 이동했다면, 버그를 내재한 copy( )는 다시 알 수 없는 미궁 속으로 여러분들을 안내하게 됩니다.

가이드라인

유효하지 않은 반복자를 역참조(dereference)하지 말자.

요약해 보면, 반복자를 이용할 때는 다음 4개의 주요 논점을 주의해야 합니다.

1. 유효한 값 : 반복자를 역참조했는가? 예를 들어, “*e.end( )”를 작성한 것은 항상 프로그래밍 오류이다.

2. 유효한 수명 : 반복자를 사용한 후에도 여전히 유효한가? 혹은, 가져온 후로부터 어떤 연산이 수행된 후에 무효화되었는가?

3. 유효한 범위 : 반복자의 쌍이 유효한 범위인가? 실제로, first가 last보다 같거나 이전의 데이터인가? 모두 같은 컨테이너에 있는 것이 확실한가?

4. 불법적인 내장 형식 처리 : 예를 들어, 코드가 앞의 “--e.end( )”에서처럼 내장된 형식의 일시적인 수정을 시도하였는가(다행히도, 컴파일러는 이런 종류의 실수는 발견하기도 합니다. 가끔씩 라이브러리의 작성자가 문법상의 편리함을 위해 이런 종류의 기능들을 선택할 수 있게 합니다)?

항목 2 : 대·소문자 구분 없는 문자열 - 파트 1 (난이도 7)

대·소문자 구분이 없는 문자열 클래스가 필요했던 적은 없나요? 이번 기회에 하나 작성할 필요가 있겠죠.

이번 항목은 세 가지 부분으로 나누어 볼 수 있습니다.

1. “*대·소문자 구분 없는*” 것이 무슨 뜻인가요?

2. 표준 std::string 클래스와 동일하지만, 확장된 stricmp( )처럼 일반적인 방법으로 대·소문자 구분이 없는 ci_string 클래스를 작성해 봅시다.[1] ci_string은 다음과 같이 사용될 수 있어야 합니다.

```
ci_string s( "AbCdE" );
// 대·소문자를 구분하지 않음
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// 물론, 아직까지 구분 가능하다.
//
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

1) 대·소문자를 구분하지 않는 stricmp 문자열 비교 함수는 C 혹은 C++ 표준의 일부가 아니지만, 많은 C와 C++ 컴파일러들에서 일반적으로 확장되어 있습니다.

3. 개체의 속성으로 대·소문자를 구분할 수 있도록 만드는 것이 과연 필요한가요?

해결책

세 가지 질문에 대한 답을 말해 보자면 다음과 같습니다.

1. "*대·소문자 구분 없는*" 것이 무슨 뜻인가요?

실제로, "*대·소문자 구분 없는*" 것은 전반적으로 여러분의 응용 프로그램과 언어에 달려 있습니다. 한 예로, 많은 언어들은 대·소문자를 구분하는 격(역자 주 일반적인 '격'의 의미와는 다소 다를 수가 있어서, 이후에는 의미상 적절히 풀어서 '대·소문자 구분이 없는'으로 설명하겠다)을 가지고 있지 않습니다. 이런 언어들에서는 강조된 문자를 강조되지 않은 문자와 비교할지 말지, 기타 등등을 결정해야 합니다. 이번 항목에서는 여러분들의 상황에 적용할 수 있도록, 표준 문자열에서 대·소문자를 구분하지 방법에 대한 지침서를 살펴보도록 하겠습니다.

2. 표준 std::string 클래스와 동일하지만, 확장된 stricmp( )처럼 일반적인 방법으로 대·소문자 구분 없는 ci_string 클래스를 작성해 봅시다.

"어떻게 대·소문자를 구분하지 않는 문자열을 만들어야 하지?"라는 질문은 매우 일반적이어서, 아마도 자신의 FAQ(이번 항목부터)에 보관할 가치가 있을 정도입니다.

다음은 완성하고자 하는 모습입니다.

```
ci_string s( "AbCdE" );
// 대 소문자를 구분하지 않음
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// 물론, 아직까지 구분 가능하다.
//
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

핵심은, string이 실제로 표준 C++에 있다는 것을 이해하는 것이죠. 정말로 string 헤더를 보면, 다음과 같은 부분을 확인할 수 있습니다.

```
typedef basic_string<char> string;
```

앞에서 보듯이, string은 실제 클래스가 아니라 템플릿을 typedef한 것입니다. 뒤집어 보면, basic_string<> 템플릿은 구현에 따른 템플릿 매개변수가 추가될 수 있지만, 기본적으로 다음과 같은 형식으로 선언되어 있습니다.

```
template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
class basic_string;
```

그러므로 실제 "string"은, 여러분이 사용하는 구현에 대한 특정적인 템플릿 매개변수를 제외하면 "basic_string <char, char_traits<char>, allocator<char> >"으로 나타낼 수 있게 되는 것이죠. allocator 항목에 대해서는 별로 신경 쓰지 않아도 되지만, char_traits 항목은 핵심이 되는 내용입니다. char_traits에서 문자들간에 서로 영향을 끼치고 비교하는 방법을 정의해 놓기 때문입니다.

그럼, 문자열을 비교해 봅시다. basic_string은 하나의 문자열이 다른 것과 같은지, 혹은 작은지 등을 판단할 수 있도록 유용한 비교 함수들을 제공합니다. 이런 문자열 비교 함수들은 char_traits 템플릿에서 제공하는 문자 비교 함수들 중 위쪽(top)에 만들어져 있습니다. 특별히, char_traits 템플릿은 같은지를 판단하는 ep( )와 작은지를 판단하는 lt( )라는 문자 비교 함수들을 제공하며, 문자열을 비교하거나 찾는데 사용하는 compare( )와 find( ) 함수들을 제공합니다.

이 함수들의 기능을 변형시키려면, char_traits 템플릿과는 다른 손수 작성한 템플릿을 제공하면 됩니다. 여기에 쉬운 방법이 있으니 참고해 볼까요?

```
struct ci_char_traits : public char_traits<char>
          // 대체할 필요가 없는
          // 다른 모든 함수들을 상속받기 위해
{
     static bool eq( char c1, char c2 )
          { return toupper(c1) == toupper(c2); }
     static bool lt( char c1, char c2 )
          { return toupper(c1) < toupper(c2); }
     static int compare( const char* s1,
                         const char* s2,
                         size_t n )
          { return memicmp( s1, s2, n ); }
               // 여러분의 플랫폼에서 가능하지 않다면,
               // 적절히 수정하기 바람
     static const char*
```

```
      find( const char* s, int n, char a )
      {
          while( n-- > 0 && toupper(*s) != toupper(a) )
          {
              ++s;
          }
          return n >= 0 ? s : 0;
      }
  };
```

그리고 마지막으로, 이를 적용하는 포인트는 다음과 같습니다.

```
  typedef basic_string<char, ci_char_traits> ci_string;
```

문자 비교 규칙을 얻어오기 위해, char_traits<char> 대신에 ci_char_traits를 사용하는 것을 제외하고는 표준 string과 유사하게 동작하는(아무튼, 대부분 표준 string과 유사하죠), ci_string이라고 이름 붙인 typedef를 만들었습니다. 별다른 노력도 들이지 않고, ci_char_traits를 대・소문자 구분하지 않도록 만듦으로써 ci_string 자체를 대・소문자를 구분하지 않도록 하였습니다. 즉, basic_string을 전혀 건드리지 않고 대・소문자 구분 없는 string을 가지게 된 것이죠. 이것이 바로 *확장성*입니다.

3. 개체의 속성으로 대・소문자를 구분할 수 있도록 만드는 것이 과연 필요한가요?

여기에서 보여진 것처럼 개체의 속성 대신에 비교 함수의 속성으로 대・소문자를 구분하는 것은 종종 유용할 경우가 있습니다. 예를 들어, 다음 코드를 생각해 봅시다.

```
  string      a = "aaa";
  ci_string   b = "aAa";
  if( a == b ) /* ... */
```

적절하게 사용된 operator ==()는 “a == b”를 true 또는 false로 판단할 수 있을까요? 간단히 생각해 보면, 둘 중 하나가 대・소문자를 가지지 않으면 대・소문자를 구분하여 비교할 수 없습니다. 하지만, 예제를 조금 고쳐서, 제 3 의 방법으로 비교를 수행하는 basic_string의 다른 예제를 넣어봅시다.

```
  typedef basic_string<char, yz_char_traits> yz_string;

  ci_string b = "aAa";
  yz_string c = "AAa";
  if( b == c ) /* ... */
```

이제, 문제를 다시 생각해 보도록 하죠. “a == b”를 true 또는 false로 판단할 수 있을까요? 이번에는, 마음대로 한 개체의 순서를 다른 개체보다 앞에 둘 수 있는지 약간 망설여 할 것입니다.

대신에, 다음과 같이 작성하면 더 명확해지지 않을까요?

```
string a = "aaa";
string b = "aAa";
if( stricmp( a.c_str(), b.c_str() ) == 0 ) /* ... */
string c = "AAa";
if( EqualUsingYZComparison( b, c ) ) /* ... */
```

많은 경우에, 비교 연산의 특성으로 대·소문자를 구분하는 능력은 유용합니다. 하지만, 실제로 개체(특히, 대부분 또는 모든 비교들이 C-형식의 char* string일 경우)의 특성을 만드는 경우, 매번 대·소문자를 구분하지 않는 비교를 사용하고 있다는 것을 기억할 필요도 없이, “자연스럽게”(예를 들어, “if(a == “text”) ...”) 값을 간단히 비교할 수 있으므로 더욱 유용합니다.

이번 항목에서는 basic_string 템플릿이 어떻게 동작하고 실제로 얼마나 유연성을 갖는지 맛을 봤습니다. 그리고 만약에, memicmp()와 toupper()가 제공한 것과 다른 방식으로 비교하고 싶다면, 단지 여기에서 보여준 5개의 함수들을 여러분의 응용 프로그램에 맞게, 문자 비교를 수행하는 코드로 바꿔주면 됩니다.

항목 3 : 대·소문자 구분 없는 문자열 - 파트 2 (난이도 5)

항목 2에서 만든 ci_string이 정말 사용할 만한가? 이제 유용성에 초점을 맞춰서 디자인 문제이나 직면하게 될 손익을 따져 보고, 남아있는 몇 가지 결함을 메꿔 보기로 하자.

항목 2의 해결책에 대해 다시 생각해 봅시다(함수 몸체는 무시).

```
struct ci_char_traits : public char_traits<char>
{
    static bool eq( char c1, char c2 )  { /*...*/ }
    static bool lt( char c1, char c2 )  { /*...*/ }
    static int compare( const char* s1,
                        const char* s2,
                        size_t n )      { /*...*/ }
    static const char*
    find( const char* s, int n, char a ) { /*...*/ }
};
```

이번 항목에서, 다음과 관련된 질문에 가능한 정확하게 답해 보도록 합시다.

1. char_traits<char>에서 ci_char_traits를 상속받는 방식이 안전한가요?

2. 왜 다음과 같은 코드가 컴파일에 실패하나요?

```
ci_string s = "abc";
cout << s << endl;
```

3. 다른 연산자(예를 들어, +, +=, =)를 사용하는 것은 어떤가요? 그리고, 다음 예제처럼 strings와 ci_strings를 매개변수로 섞어 쓰는 것은 어떤가요?

```
string       a = "aaa";
ci_string    b = "bbb";
string       c = a + b;
```

해결책

세 가지 질문에 대한 답을 생각해 보셨나요? 하나씩 살펴보도록 합시다.

1. char_traits<char>에서 ci_char_traits를 상속받는 방식이 안전한가요?

public 상속은 일반적으로 Liskov Substitution Principle(LSP)에 따라 동작하도록 설계합니다(항목 22와 28 참조). 하지만, 이번은 LSP의 드문 예외 중 하나입니다. ci_char_traits는 기반 클래스 char_traits<char>의 포인터나 참조를 통해 다형적으로 사용되도록 만들어지지 않았기 때문이죠. 표준 라이브러리는 traits 개체를 다형적으로 사용하지 않습니다. 이 경우에, 상속은 단지 편리함("게으름"이라고 말할 수도 있겠네요)을 위해서만 사용됩니다. 그리고 상속은 개체-지향적 방식으로도 사용되지 않습니다.

다른 한편으로는, 여전히 다른 방식으로 LSP를 적용할 수 있습니다. 파생된 개체가 반드시 기반 개체와 같은 방식(WORK_LIKE_A)으로 즉, basic_string 템플릿의 요구사항과 같은 조건을 가지고 동작해야 할 때, 이를 컴파일 시점에 적용할 수 있습니다. 뉴스그룹에서 Nathan Myers는[2] 이를 다음과 같이 표현하였습니다.

2) Nathan씨는 C++ 표준 위원회의 원로 회원이며, 표준의 로케일(locale) 기능들의 주요 저자입니다.

"다시 말해서, LSP는 단지 컴파일 시점에 적용합니다. 그리고 간편히 "요구사항들의 목록"을 말하면 됩니다. 나는 이 경우를 구분하였으면 합니다. Generic Liskov Substitution Principle (GLSP): 템플릿의 매개변수로 전달되는 어떤 형식(또는 템플릿)이든지, 그 매개변수에 대한 요구사항들의 목록을 수용해야 합니다.

반복자 형식에서 파생된 클래스와 traits 클래스들은 GLSP의 대상이지만, 전형적인 LSP의 고려사항(예를 들어, 가상 소멸자 기타 등등)은 실행 시점의 다형성이 요구사항들의 목록에 명시되어 있는지에 따라 적용되거나 적용되지 않습니다."

다시 간단히 말해, 이 상속은 GLSP(LSP가 아니라면)를 수용하기 때문에 안전합니다. 하지만, 이를 여기에서 사용한 가장 큰 이유는 편리함(다른 char_traits<char> 기능들을 쓰는 것을 피하기 위해서) 때문이 아니라, *무엇이 다른가*를 설명하기 위해서입니다. 즉, 우리가 원하는 효과를 얻기 위해 단지 4개의 연산자만 바꾸면 된다는 것을 설명하기 위한 것이죠.

첫 번째 질문을 통해 다음 항목들에 관해서 더 깊이 있게 생각해 보았으면 합니다. ❶ 상속의 적절한 사용(과 부적절한 사용), ❷ 정적 멤버들만 있다는 사실을 암시, ❸ char_traits 개체가 다형적으로 사용되지 않는다는 사실들에 대해서요.

2. 왜 다음과 같은 코드가 컴파일에 실패하나요?

```
ci_string s = "abc";
cout << s << endl;
```

힌트 : C++ 표준안의 21.3.7.9 [lib.string.io]에서, basic_string에 대한 operator <<의 선언은 다음과 같이 되어 있습니다.

```
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const basic_string<charT,traits,Allocator>& str);
```

답 : 먼저, cout이 실제로 basic_ostream<char, char_traits<char> >인 것에 주목합시다. 이제, 문제가 좀 더 명확해질 것입니다. basic_string에 대한 operator<<는 템플릿이고, string으로 같은 "char 형"과 "traits 형"으로 basic_ostream에 삽입한다는 것만 기술하고 있습니다. 즉, 표준 operator<<는, ci_string이 앞의 해결책처럼 char_traits<char>로부터 상속받았음에도 불구하고, ci_string을 cout이 이해할 수 없는 basic_ostream<char, ci_char_traits>로 보여주게 됩니다.

이 문제를 해결하는데는 두 가지 방법이 있습니다. ci_string에 대한 operator<<( )와 operator>>( )를 여러분 스스로 정의하거나, 또는 여러분의 응용 프로그램에서 문자열이 내부적으로 널(null)을 갖지 않는다면, operator<<(const char*)를 사용할 때 ".c_str( )"를 붙여서 사용하면 됩니다.

```
cout << s.c_str() << endl;
```

3. 다른 연산자(예를 들어, +, +=, =)를 사용하는 것은 어떤가요? 그리고, 다음 예제처럼 strings와 ci_strings를 매개변수로 섞어 쓰는 것은 어떤가요?

```
string    a = "aaa";
ci_string b = "bbb";
string    c = a + b;
```

역시, 이 문제를 처리하는 두 가지 방법이 있습니다. operator+( ) 함수를 다시 정의하거나, operator+(const char*)를 사용할 때마다 ".c_str()"를 붙여서 사용하면 됩니다.

```
    string  c = a + b.c_str()
```

항목 4 : 최대 재사용 가능한 일반 컨테이너 - 파트 1 (난이도 8)

간단한 컨테이너 클래스를 어떻게 유동적으로 만들 수 있을까요? 힌트: 진행하면서 멤버 템플릿에 대해 좀 더 배우게 될 것입니다.

어떻게 하면 다음과 같은 고정 길이 벡터(vector) 클래스의 복사 생성자와 복사 할당자를 가장 잘 구현할 수 있을까요? 어떻게 하면 생성자와 할당자를 최대로 사용할 수 있을까? 힌트 : 클라이언트 코드에서는 어떻게 사용할지를 생각해 보면 됩니다.

```
template<typename T, size_t size>
class fixed_vector
{
public:
    typedef T*           iterator;
    typedef const T* const_iterator;
    iterator             begin()          { return v_; }
    iterator             end()            { return v_+size; }
    const_iterator       begin()   const  { return v_; }
    const_iterator       end()     const  { return v_+size; }
private:
     T v_[size];
};
```

노트 : 다른 것들은 고치지 않아야 합니다. 이 컨테이너는 완전한 STL-호환은 아니고, 단지 하나의 민감한 문제점만 가지고 있을 뿐입니다. 중요한 문제를 설명하기 위해 간단히 설정한 것입니다.

해결책

이 항목에 대한 해결책은 좀 다르게 진행해 볼까요? 이번에는 먼저 해결책을 보고, 여러분 스스로 제시한 해결책을 설명하고 비평해 보도록 합시다. 생각해 보았다면, 바로 항목 5로 갑시다.

항목 5 : 최대 재사용 가능한 일반 컨테이너 - 파트 2 (난이도 6)

진행 노트 : 이 항목에서 사용된 예제는 영국의 C++ 매거진 Overload의 이슈 12와 20번에서 Kevlin Henney가 제공하고 Jon Jagger가 분석한 곳에서 개조한 것입니다(영국 독자들은 주의할 것: 이 항목에 대한 답은 Overload #20에서 제공한 것과는 다릅니다. 사실, 거기서 제공한 효율적인 최적화는 이 문제의 해결책에는 동작하지 않을 것입니다).

다음 해결책은 무슨 일을 합니까? 그리고 이유는? 각 생성자와 연산자를 설명해 봅시다. 이 디자인이나 코드에 어떤 흠이 있나요?

```
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*        iterator;
  typedef const T* const_iterator;
  fixed_vector() { }

  template<typename O, size_t osize>
  fixed_vector( const fixed_vector<O,osize>& other )
  {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
  }
  template<typename O, size_t osize>
  fixed_vector<T,size>&
```

```
    operator=( const fixed_vector<O,osize>& other )
    {
      copy( other.begin(),
            other.begin()+min(size,osize),
            begin() );
      return *this;
    }

    iterator        begin()         { return v_; }
    iterator        end()           { return v_+size; }
    const_iterator  begin()  const  { return v_; }
    const_iterator  end()    const  { return v_+size; }

  private:
    T v_[size];
  };
```

해결책

앞에서 제공한 해결책을 분석해 보고 어떻게 질문에 대한 답변을 했는지 살펴보도록 합시다. 먼저, 원래의 질문을 다시 기억해 보도록 하죠. “어떻게 하면 다음과 같은 고정 길이 벡터(vector) 클래스의 복사 생성자와 복사 할당자를 가장 잘 구현할 수 있을까요? 어떻게 최대로 생성자와 할당자의 사용성을 끌어올릴 수 있을까요? 힌트 : 클라이언트 코드가 원하는 것들을 생각해 보면 됩니다.”

복사 생성자와 복사 할당자

먼저, 첫 번째 질문이 애매 모호합니다. 고맙게도, 원래 코드는 이미 잘 동작하는 복사 생성자와 복사 할당 연산자를 가지고 있습니다. 제시한 해결책은 생성자와 할당자를 좀 더 융통성 있게 만들기 위해 템플릿 생성자와 템플릿 할당 연산자를 추가하는 것에 목적이 있습니다.

```
  template<typename O, size_t osize>
  fixed_vector( const fixed_vector<O,osize>& other )
  {
      copy( other.begin(),
            other.begin()+min(size,osize),
            begin() );
  }
  template<typename O, size_t osize>
  fixed_vector<T,size>&
```

```
operator=( const fixed_vector<O,osize>& other )
{
    copy( other.begin() ,
          other.begin()+min(size ,osize) ,
          begin() ) ;
    return *this ;
}
```

앞의 함수들은 복사 생성자와 복사 할당 연산자가 아님을 유의합시다. 그 이유는, 복사 생성자와 복사 할당 연산자는 특별히, 정확히 같은 형식 즉, 같은 템플릿 매개변수를 가진 다른 개체에서 생성/할당이 이루어지는 것이기 때문입니다.

```
struct X
{
    template<typename T>
    X( const T& ) ;     // 복사 생성자가 아니다, T가 X가 될 수 없다.

    template<typename T>
    operator=( const T& ) ;
    // 복사 할당 연산자가 아니다, T가 X가 될 수 없다.
};
```

하지만, 이 두 개의 템플릿 멤버 함수들은 복사 생성자와 복사 할당자의 형식과 정확히 일치한다고 생각할 수 있습니다. 그래도, 실제로는 두 클래스에서 T는 X가 될 수 없으므로, 일치할 수 없는 것이죠. 표준 문서(12.8/2, note 4)를 인용해 보면,

> 템플릿 생성자는 절대로 복사 생성자가 될 수 없기 때문에, *이러한 템플릿의 존재가 복사 생성자의 암시적인 선언을 제한하지 않는다. 템플릿 생성자는 복사 생성자를 포함하여 다른 템플릿 생성자의 오버로드(overload)에 포함되고, 아마도 다른 생성자보다 더 잘 어울린다면 개체를 복사하는데 사용될 것이다.*

복사 할당 연산자에 대한 유사한 문구(12.8/9, note 7 참조)도 볼 수 있습니다. 컴파일러가 암시적 버전을 생성하므로, 제안된 해결책은 사실 여전히 같은 복사 생성자와 복사 할당 연산자를 원래 코드에서 가지고 있습니다. 우리가 한 것은 예전 버전을 대체한 것이 아니라, 생성과 할당을 동적으로 확장한 것입니다.

다른 예제로, 다음 프로그램을 생각해 봅시다.

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w;
fixed_vector<int,4>  w2(w);
          // 암묵적으로 복사 생성자를 호출한다.
fixed_vector<int,4>  w3(v);
          // 템플릿 변환 생성자를 호출한다.
w = w2;   // 암묵적으로 복사 할당 연산자를 호출한다.
w = v;    // 템플릿 할당 연산자를 호출한다.
```

그러므로, 질문에서 정말로 원한 것은, 엄밀히 말하면 융통성 있는 "복사 생성자와 복사 할당자"가 아니라 "다른 fixed_vectors에서 생성과 할당"을 융통성 있게 제공하는 것이 됩니다.

생성과 할당의 유용성에 관하여

유용성에 관련해서 두 가지 중요 문제가 있습니다.

1. 다양한 데이터형의 지원 (상속 포함)

확실히 fixed_vector가 동종의 컨테이너를 가지고 있는 동시에, 실제로 다른 개체를 가지고 있는 또 하나의 fixed_vector로부터 생성과 할당을 할 수 있어야 합니다. 원본 개체가 우리 개체의 데이터형에 할당이 가능하면, 항상 이 등식도 성립되어야 합니다.

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w(v);      // 템플릿 생성
w = v;                          // 템플릿 할당

class B           { /*...*/ };
class D : public B { /*...*/ };

fixed_vector<D*,4> x;
fixed_vector<B*,4> y(x);        // 템플릿 생성
y = x;                          // 템플릿 할당
```

이 예제에서 D*가 B*에 할당할 수 있으므로, 정확히 기대했던 대로 동작합니다.

2. 다양한 크기의 지원

마찬가지로, 클라이언트는 다른 크기의 fixed_vector에서 생성하거나 할당하고 싶어합니다. 다시 말해서, 이 기능을 지원해야 한다는 것이죠. 예를 들어,

```
fixed_vector<char,6>  v;
fixed_vector<int,4>   w(v);     // 4의 값을 이용해서 초기화
w = v;                          // 4의 값을 이용해서 할당
class B             { /*...*/ };
class D : public B  { /*...*/ };
fixed_vector<D*,16> x;
fixed_vector<B*,42> y(x);       // 16의 값을 이용하여 초기화
y = x;                          // 16의 값을 이용하여 할당
```

대안 : 표준 라이브러리로의 접근

종전에 보았던 함수의 재사용성과 문법 형태가 그런 대로 맘에 들지만, 아직 사용하기엔 뭔가 모자란 듯 보이네요. 한 번, 표준 라이브러리 스타일을 따르는 다음 접근 방식을 살펴보도록 합시다.

1. **복사**

```
template<class RAIter>
fixed_vector( RAIter first, RAIter last )
{
  copy( first,
        first+min(size,(size_t)last-first),
        begin() );
}
```

이제 복사할 때, 다음과 같이 작성하지 말고,

```
fixed_vector<char,6>  v;
fixed_vector<int,4>   w(v);     // 4의 값을 이용하여 초기화
```

다음과 같이 작성하면 됩니다.

```
fixed_vector<char,6>  v;
fixed_vector<int,4>   w(v.begin(), v.end());
                                // 4의 값을 이용하여 초기화
```

잠시 다음을 읽기 전에 곰곰이 생각해 봅시다. 어떤 것 같나요? 생성할 때, 먼저 제시한 해결책과 이 표준 라이브러리와 비슷한 스타일 둘 중에 어떤 형식이 더 좋아 보이나요?

이 경우, 처음에 제시한 해결책이 더 사용하기 편한 반면, 표준 라이브러리 형식은 좀 더 융통성을 갖고 있습니다(예를 들어, 사용자가 다른 종류의 컨테이너에서 문자열의 일부를 선택하여 복사할 수 있습니다).

2. **할당**

operator=( )는 단지 하나의 매개변수만을 취하므로, 템플릿에서 반복자의 범위를 정하여 할당하게 할 수 없습니다. 대신에, 이름을 가진 함수로 제공할 수는 있겠죠.

```
template<class Iter>
fixed_vector<T,size>&
assign( Iter first, Iter last )
{
    copy( first,
          first+min(size,(size_t)last-first),
          begin() );
    return *this;
}
```

이제 할당할 경우 이렇게 작성하지 말고,

```
w = v;                      // 4의 값을 이용하여 할당
```

다음과 같이 작성하면 됩니다.

```
w.assign(v.begin(), v.end());
                            // 4의 값을 이용하여 할당
```

기술적으로, assign( )은 굳이 필요하지 않습니다. 이것 없이도 같은 융통성을 얻을 수 있지만, 좀 보기 흉하고 덜 효율적입니다.

```
w = fixed_vector<int,4>(v.begin(), v.end());
    // 4의 값을 이용하여 초기화하고 할당
```

다시 한번 더 대안을 생각해 봅시다. 할당의 경우는 제시한 해결책과 이 표준 라이브러리 스타일 중에 어떤 것이 더 좋을까요?

이번에는, 융통성이 판단할 수 있는 잣대가 될 수는 없는 것 같네요. 사용자가 쉽게 (그리고 혹은 더 융통성 있게) 복사할 수 있습니다. 이렇게 사용하는 대신에,

```
w.assign( v.begin(), v.end() );
```

이런 식으로 사용하기만 하면 됩니다.

```
copy( v.begin(), v.begin()+4, w.begin() );
```

따라서, 이 경우엔 범위를 지정하는 assign()을 사용할 이유가 없습니다. 그리고 할당할 때, 제시한 방법을 사용하고 범위를 지정해서 할당할 경우에는 클라이언트가 직접 copy()를 사용하게 하는 것이 가장 좋은 것 같습니다.

왜 기본 생성자를 작성하는가?

마지막으로, 왜 제시한 해결책에서, 단지 컴파일러가 생성하는 기본 생성자처럼 보이는, 비어 있는 기본 생성자를 만들었을까요? 대답은 아주 간단하고도 명확합니다. 어떤 종류의 생성자를 정의하기만 하면, 컴파일러는 기본 생성자를 생성하지 않게 됩니다. 확실히, 앞과 같은 클라이언트 코드는 이런 작업이 필요합니다.

망설이는 문제

이번 항목의 문제에서 이런 말도 있었죠, "이 디자인이나 코드에 어떤 흠이 있나요?"

아마도, 이 책의 다음 내용에서는 다양한 예외 안전 보증(항목 8부터 항목 11) 처리들을 구분해 볼 것입니다. 컴파일러가 생성한 복사 할당 연산자처럼, 우리의 템플릿 할당 연산자는 완벽하게 좋은 기본적인 보증을 해줍니다. 잠시 생각해 보면, 좀 더 확실한 보증을 제공하기 위해, 모든 예외를 안전하게 처리하려면, 어떻게 해야 하는지 살펴보도록 합시다. 템플릿 할당 연산자가 다음과 같이 정의되었던 것을 상기해 보죠.

```
template<typename O, size_t osize>
fixed_vector<T,size>&
operator=( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
  return *this;
}
```

copy( ) 연산 동안, T 할당 중에 하나라도 실패한다면 개체는 무질서한 상태가 됩니다. fixed_vector 개체의 일부 내용은 할당이 실패하기 전의 생태로 남아있고, 다른 일부 내용은 새로 갱신된 상태로 남아있게 됩니다.

안타깝게도, 현재의 디자인으로는 fixed_vector에서 *할당에 대해 강한 예외 안정성을 만들 수 없습니다*. 왜 그럴까요? 다음과 같은 이유에서입니다.

- 일반적으로, 이 문제를 해결하기 위한 (그리고 가장 쉬운) 정석은 두 개의 fixed_vector 개체의 내용을 서로 바꿔주며 예외를 일으키지 않는 아주 작은 Swap( ) 함수를 제공하는 것입니다. 그리고, 간단히 임시 개체를 만들고 바꿔주는 관용구를 사용하는, operator=( )를 사용하면 됩니다.
- 두 개의 fixed_vector 개체의 내부 자료를 교환할 예외를 던지지 않는 아주 작은 Swap( ) 함수를 작성할 방법이 없는 것이 문제입니다. fixed_vector의 내부 자료는 값의 대입과 같은 간단한 작업으로 복사할 수 없는 배열로 이루어져 있기 때문이죠. 결국, 제시한 operator=( )에는 부가 작업이 필요하게 되는 문제가 발생합니다.

하지만 아직 실망할 필요는 없습니다. 실제로, 좋은 해결책이 있지만, fixed_vector의 디자인을(정말 약간이지만) 멤버 배열 대신 동적으로 할당 가능한 배열로 바꿔야 합니다. 참으로, 이것만 제거하면 fixed_vector는 커다란 효율성을 가져오게 됩니다. 소멸자를 만들어야 하지만, 이런 것쯤은 강력한 예외 안정성에 비하면 헐값이지요.

```
// 강력한 예외 버전
//
template<typename T, size_t size>
class fixed_vector
{
public:
    typedef T*              iterator;
    typedef const T* const_iterator;

    fixed_vector() : v_( new T[size] ) { }

    ~fixed_vector() { delete[] v_; }
    template<typename O, size_t osize>
    fixed_vector( const fixed_vector<O,osize>& other )
      : v_( new T[size] )
      { try {copy(other.begin() ,other.begin()+min(size ,osize) ,
                    begin()) ;}
            catch(...) { delete[] v_; throw; } }
    fixed_vector( const fixed_vector<T, size>& other )
        : v_( new T[size] )
        { try {copy(other.begin(), other.end(), begin()) ;}
            catch(...) { delete[] v_; throw; } }

    void Swap( fixed_vector<T,size>& other ) throw()
    {
        swap( v_, other.v_ );
    }

    template<typename O, size_t osize>
    fixed_vector<T,size>& operator=(
        const fixed_vector<O,osize>& other )
    {
        fixed_vector<T,size> temp( other ); // 모든 작업을 함
        Swap( temp ); return *this;         // 예외를 일으키지 않음
    }
    fixed_vector<T,size>& operator=(
        const fixed_vector<T,size>& other ) {
        fixed_vector<T,size> temp( other ); // 모든 작업을 함
        Swap( temp ); return *this;         // 예외를 일으키지 않음
    }

    iterator        begin()       { return v_; }
    iterator        end()         { return v_+size; }
    const_iterator  begin() const    { return v_; }
    const_iterator  end()   const    { return v_+size; }

private:
    T* v_;
};
```

일반적인 실수

절대로 생각을 다 마친 다음에 예외 안정성을 만들지 말자. 예외 안정성은 클래스의 디자인에 영향을 준다. 결코, "단지 세부 구현 과정"이 아니다.

요약하자면, 이번 항목을 통해 멤버 변수 템플릿을 주물럭거릴 수 있기를 기대합니다. 또한, 왜 표준 라이브러리가 전반적으로 사용되고 있는지도 알았으면 좋겠습니다. 아직 익숙하지 않더라도 너무 실망하지는 마십시오. 현재 모든 컴파일러가 멤버 템플릿을 지원하지는 않고 있지만, 표준이므로 조만간 모든 컴파일러가 지원하게 될 것입니다.

여러분 자신만의 클래스를 만들 때, 더 큰 효율을 얻기 위해서 멤버 템플릿을 사용하는 것이 좋습니다. 단지, 사용자에게 기쁨을 주기 위해서가 아니라 그 보다 재사용에 가장 적절한 코드를 다시 사용하기 위해서입니다.