

# C++ 11

## FAQs

C++ 11 - Frequently Asked Questions

JAEBUM LEE

The Publisher 

This document is based on C++ 11 FAQ in the website of Bjarne Stroustrup - the creator of C++. This document is translated by Jaebum Lee, with many other contributors under the open-computer-book distribution project. The entire content of the document cannot be modified without the direct permission of original translator - Jaebum Lee, but can be freely distributed through any kind of medium without the permission.

# 제 1 장

## Introduction

이 글은 C++ 언어의 창시자 **Bjarne Stroustrup** 씨의 개인 홈페이지에서 C++ 11 FAQ 라는 제목의 글을 번역한 내용입니다. 원문은 <http://www.stroustrup.com/C++11FAQ.html>에서 볼 수 있고, 이런 훌륭한 글을 제공해주신 Bjarne Stroustrup 씨에게 감사의 말을 전합니다. 양질의 컴퓨터 관련 문서를 한국어로 공급하기 위한 프로젝트의 일환으로 이 문서는 <http://itguru.tistory.com>에서 무료로 배포되고 있습니다. (오픈북 프로젝트에 관련한 자세한 내용은 <http://itguru.tistory.com>을 참조하시기 바랍니다.) 문서 전체 번역은 이재범([kev0960@gmail.com](mailto:kev0960@gmail.com)) 외 수 많은 사람들이 도움을 주셨습니다.

C++ 11은 비교적 최근에 (2011년) 발표되었으며, 기존의 C++ 98과 그 이전 버전들에 비해 많은 기능들이 추가되었습니다. 하지만 애석하게도, 새롭게 추가된 기능들에 대해 간략하게 훑어볼 수 있도록 제공되는 한글 문서가 거의 전무한 실정입니다. 따라서 저는 C++의 원제작자인 Bjarne Stroustrup 씨의 개인 홈페이지에 올라와 있는, 본인 직접 C++ 11에 관련한 여러 질문들에 답변한 내용을 바탕으로 이와 같이 문서로 만들게 되었습니다. 이 문서는 새로운 기술들에 대해 간단하게 짚어가며, 각각의 기술들에 대한 자세한 설명은 ‘참고자료’ 형태로 링크해놓았습니다.

참고로, 저는 번역을 수행함으로써, 널리 영어 그대로 쓰이는 단어(예를 들어 템플릿(template) 등)가 아닐 경우, 최대한 한글로 번역하도록 노력하였습니다. 물론, 이렇게 직접 번역된 단어 옆에는 원 단어를 써놓아서 이해하는데 큰 무리가 없도록 하였습니다.

### 1.1 이 글의 목적

o) C++11 FAQ을 제작한 이유로;

- C++ 11에서 제공하는 새로운 기능(facilities)들 (언어와 표준 라이브러리에 새로 추가된 것들)과 이전 ISO C++ 표준에서 제공되는 것들
- ISO C++ 표준이 지향하는 목표에 대해 설명하기 위해서
- 새로운 기능들에 대해 유저들의 견해를 보여주기 위해
- 여러 기능의 심도 있는 공부를 위한 레퍼런스를 제공하기 위해서
- C++에 기여한 많은 분들의 이름을 언급하기 위해서 (대부분은 C++ 표준 위원회를 위한 보고서를 작성한 분들). 표준은 얼굴 없는 단체에 의해 작성된 것이 아닙니다.

참고로 말하자면 이 FAQ 의 목적은 C++ 의 새로운 기능 개개에 대해서 자세한 설명이나 토론을 하기 위한 것은 아닙니다. 이 FAQ 에서는 C++ 의 새로운 기능들에 대한 간단한 예시와 설명과 함께, 자세한 공부를 싶어하는 분들께 레퍼런스를 제공하고 있습니다. 여기서의 목표는 그 새로운 기능이 얼마나 복잡하냐에 상관 없이 ‘한 기능당 한 페이지’ 원칙을 지키려고 합니다. 자세한 내용은 레퍼런스에서 볼 수 있습니다.

## 1.2 질문 목록들

아래는 먼저 높은 수준의 질문들입니다.

- 당신은 C++ 11 에 대해 어떻게 생각하시나요?
- C++ 0x 가 공식적인 표준이 언제 될까요?
- 컴파일러가 C++ 11 를 언제 구현할까요?
- 새로운 표준 라이브러리를 언제 사용할 수 있을까요?
- C++ 11 에서 제공하는 새로운 언어 기능들은 무엇인가요? (아래 질문들 참조)
- C++ 11 에서 제고공하는 새로운 표준 라이브러리는 무엇인가요? (아래 질문들 참조)
- C++ 0x 가 목표했던 것은 무엇인가요?
- 위원회가 추구했던 디자인 목표는 무엇인가요?
- 위원회 보고서들을 어디서 찾을 수 있나요?
- C++ 11 에 대한 전문적이고 기술적인 문서들을 어디서 찾을 수 있나요?
- C++ 11 에 대해 어디서 읽을 수 있나요?
- C++ 11 에 대한 영상 자료들은 있나요?
- C++ 11 은 배우기 어렵나요?
- 위원회를 어떻게 운영되나요?
- 누가 위원회에 있나요?
- C++1y 가 있을 예정인가요?
- “concepts” 에는 무슨 일이 있었나요?
- 당신이 별로 마음에 들지 않는 기능들이 있나요?

개개의 언어 기능 자체에 대한 질문들은 아래와 같습니다.

- `_cplusplus`
- `alignments`
- `atomic` 연산들
- `auto` (초기화자(initializer) 로 부터 타입 유추)

- C99 기능들
- enum 클래스
- [[carries\_dependency]]
- 예외의 복사와 다시 던지기 (rethrow)
- 상수 표현식 (constexpr)
- decltype
- 디폴트 조정하기 : default 와 delete
- 디폴트 조정하기 : 이동과 복사
- 대표 생성자(delegating constructor)
- 동시성(Concurrency) 를 이용한 동적 초기화와 파괴
- 예외의 전파
- 명시적(explicit) 타입 변환 연산자
- extern 템플릿
- for 문 (range-for 문 참조)
- 후위 리턴 타입 문법 (확장된 함수 선언 문법)
- 클래스 내부 멤버 초기화자
- 상속된 생성자들
- 초기화 리스트(initializer\_list)
- 인라인 네임스페이스(inline namespace)
- 람다(lambda)
- 템플릿 인자로 사용되는 지역 클래스
- long long integer (최소 64비트)
- 메모리 모델
- 이동 연산 (우측값 레퍼런스 참조)
- 줄어듬(narrowing)
- [[noreturn]]
- 널 포인터 (nullptr)
- 오버라이드 컨트롤 : override
- 오버라이드 컨트롤 : final
- PODs
- range-for 문

- raw string 리터럴
- <>
- 우측값 참조
- static\_assert
- 템플릿 별명(alias)
- 템플릿 typedef (템플릿 별명 참조)
- thread\_local
- 유니코드 문자
- 단일화(uniform) 된 초기화 문법
- union
- 사용자 정의 리터럴
- 가변인자(variadic) 템플릿

위 내용들에 대해 설명하면서 일부 예시들은 위 기능들을 예시한 저자들의 문서에서 가져왔습니다. 그 저자 분들께 감사드립니다. 또한, 많은 예시들은 저의 문서들과 설명에서 가져왔습니다.

개개의 표준 라이브러리에 대한 질문들은 다음과 같습니다.

- 표준 알고리즘의 향상된 부분들
- array
- async()
- atomizing 작업들
- condition 변수들
- 표준 컨테이너의 향상된 부분들
- function 과 bind
- forward\_list
- future 와 promise
- 가비지 컬렉션 ABI
- hash\_tables (unordered\_map 참조)
- 메타 프로그래밍(meta programming)과 type traits
- 상호 배제(Mutual exclusion)
- 난수 생성
- <regex> (정규 표현식 라이브러리)
- 범위있는 할당자

- `shared_ptr`
- 스마트 포인터 (`shared_ptr`, `weak_ptr`, `unique_ptr`)
- 쓰래드(thread)
- Time 기능들
- `tuple`
- `unique_ptr`
- `unordered_map`
- `weak_ptr`

아래는 위 질문들에 대한 답변들입니다.



## 제 2 장

# Answers

### 2.1 C++ 11에 대해 어떻게 생각하시나요?

이는 (저에게) 놀랍게도 엄청나게 많이 물어보는 질문이지요. 아나 이 질문에 제가 가장 많이 받았던 질문일 것입니다. 놀랍게도 C++ 11은 새로운 언어 처럼 느껴집니다. C++ 11의 요소들이 이전 언어 보다 훨씬 더 짜임새있고, 이전 버전 보다 높은 수준의 프로그래밍을 좀더 자연스럽고 효율적으로 할 수 있게 되었습니다. 만일 여러분이 C++ 을 단순히 C 의 나아진 버전이나, 단순한 객체 지향 언어라 생각한다면, 이러한 장점들을 놓치게 될 것이지요. 이전 버전 보다 추상화를 좀더 유연하고(flexible), 쉽게 제공할(affordable) 수 있게 되었습니다. 만일 여러분이 어떠한 것에 대해 개별적인 아이디어나 객체로 생각하는 게 있다면, 프로그램으로 직접적으로 표현할 수 있습니다; 즉, 실제 세계의 객체들을 모델링할 수 있으며, 코드로 직접적으로 추상화 시킬 수 있다는 것입니다. 여러분의 아이디어들은 하나의 추상 메커니즘으로 끼워 맞추는 것이 아니란, 열거형(enumration), 객체, 클래스, 그리고 클래스 위계 (e.g 상속된 생성자), 템플릿, 별명(alias), 예외(exceptions), 루프, 쓰레드 등으로 표현할 수 있습니다.

제가 꿈꾸는 이상은, 프로그래밍 언어 기능들이 프로그래머들이 시스템 디자인이나 구현에 다른 방식으로 생각할 수 있도록 사용되는 것입니다. 제 생각에 C++ 11은 이를 할 수 있습니다 - 단순히 C++ 11 프로그래머들에게만 뿐만이 아니라, 다른 여러 현대 프로그래밍 언어들에 익숙한 프로그래머들 에게도, 그리고 많은 범위의 시스템 프로그래밍에도 가능하다는 것입니다.

쉽게 말해, 저는 아직도 낙관론자입니다.

### 2.2 C++ 0x 가 공식적인 표준이 언제 될까요?

지금이요! 첫번째 드래프트가 2008년 9월에 제출되었고, 최종 국제 표준 드래프트(Final International Draft standard FCD) 가 ISO C++ 위원회에서 2011년 3월 25일 만장일치로 승인되었습니다. 이는 2011년 8월에 21 - 0 의 국가간 투표(national vote)로 공식적으로 승인되었습니다. 표준은 그 해 (2011) 년에 출판되었습니다.

관습에 따라, 새 표준은 C++ 11이라 부르기로 하였습니다 (왜냐하면 2011년에 출판되었으므로) 개인적으로 저는 그냥 C++이라 쓰고, 만일 기존 버전의 C++ 언어들과 구별할 때에만 연도를 표시하는 것을 선호합니다. 예를 들어 ARM C++, C++ 98, C++ 03과 같이 말이지요. 참고로, 전환 기간동안에는

저는 C++ 0x 를 사용합니다. x 를 16진수 표현이라 생각하세요.

### 2.3 컴파일러가 C++ 11 를 언제 구현할까요?

현재의 많이 사용되는 컴파일러 (e.g GCC C++, Clang C++, IBM C++, Microsoft C++) 에는 이미 많은 C++ 기능들이 구현되어 있습니다. 예를 들어, 이들 컴파일러에는 대부분의 새 표준 라이브러리들이 들어 있지요. 저는 컴파일러들이 새로 패치될 때마다, C++ 의 새 기능들이 계속 추가될 것이라 생각합니다. 제가 예상하기에 첫 번째 완전한 C++ 11 컴파일러가 2012년 즈음에 나올 것이라 생각하는데, 사실 언제 C++ 11 을 완벽히 지원하는 컴파일러가 나올지, 어떤 컴파일러가 될지 예상하는데 신경을 쓰고 싶지는 않습니다. 사실 모든 C++ 11 기능들이 이미 누군가에 의해 어디선가 구현되었으므로, 컴파일러 개발자들은 이러한 구현 자료를 이용할 수 있을 것이빈다.

아래는 각각의 컴파일러 C++ 11 지원 여부 상태입니다.

- 컴파일러 간 비교
- GCC
- IBM
- Microsoft
- EDG
- Clang

### 2.4 새로운 표준 라이브러리를 언제 사용할 수 있을까요?

새로운 표준 라이브러리의 초기 버전들은 GCC, Clang, Microsoft 구현에서 제공되고 있으며, 특히 이들은 boost 를 통해 사용 가능합니다.

### 2.5 C++ 11 에서 제공하는 새로운 언어 기능들은 무엇인가요?

다른 사람이 좋은 아이디어라 생각하는 기능들을 단순히 언어에 추가한다고 해서 언어가 발전하는 것은 아닙니다. 사실은, 모든 현대 언어들의 기능들은 저에게 다른 누군가에 의해 이미 C++ 에 추가하도록 제안되었습니다. 예를 들어 C99, C#, Java, Haskell, Lisp, Python, Ada 를 모두 포함하는 언어를 한 번 상상해 보세요. 이 문제를 좀 더 복잡하기 위해, 비록 위원회가 나쁜 기능이라고 결정한 것이라도, 이전 언어에서 사용되었던 기능을 쉽게 없앨 수 없는 것입니다. 경험에 따르면, 비록 예전의 사용을 권하지 않고, 나쁜 기능이라 판단된 것들도, 사용자들이 호환성 문제를 위해 삭제하지 말라고 강요하는 경우가 많습니다.

홍수처럼 쏟아지는 제안되는 C++ 기능 중에서 이성적으로 골라내기 위해 우리는 몇 가지 특정한 디자인 목표들을 구축하였습니다. 사실 우리는 이 디자인 목표들을 완벽하게 따를 수 없고, 이 목표들은 위원회로 하여금 어떠한 방향으로 나아가도록 세세하게 가이드 할 수는 없습니다.

결과적으로 언어적 차원에서 비약적으로 향상된 부분은 바로 추상화 메커니즘입니다. 새로운 C++에서 제공하는 추상화 기능들은 기존의 소위 ‘노가다’ 뒤던 코드들에 비해 우아하고, 유연하고, 그리고 비용이 적게 듭니다. 우리가 ‘추상화’를 이야기 하면, 많은 사람들은 단순히 ‘클래스’나 ‘객체’를 더올립니다. 하지만 C++ 11은 이 보다 더 나아가서, 깨끗하고 안전하게 정의할 수 있는 사용자 정의 타입의 범주가 크게 늘어났고, 특히 새롭게 추가된 초기화자 리스트(initializer-list), 단일화 된 초기화(uniform initialization), 템플릿 별명(alias), 우측값 레퍼런스, 디폴트/삭제된 함수(default and deleted function), 가변인자 템플릿 들을 사용할 수 있게 되었습니다. 뿐만 아니라, auto, 생성자 상속, decltype 를 이용해서 이들의 구현 역시 편리해 졌고요. 이러한 향상된 기능들은 C++ 11이 마치 새로운 언어 느낌이 나게 하였습니다.

새롭게 추가된 언어 기능 목록을 보려면 the feature list 를 참조하세요

## 2.6 C++ 11에서 제공하는 새로운 표준 라이브러리는 무엇인가요?

더 많은 표준 라이브러리를 제공하였으면 좋겠지만, 사실 표준 문서의 70% 이상이 이미 표준 라이브러리에 대한 정의들입니다. (심지어 C 표준 라이브러리에 대한 설명을 빼고도 말이지요) 비록 여러분들 중 일부는 더 많은 표준 라이브러리를 보았으길 원해였지만, 아무도 라이브러리 제작 그룹이 게을러서 라고는 말할 수 없을 것입니다. 왜냐하면, 초기화 리스트, 우측값 참조, 가변인자 템플릿, noexcept, constexpr 을 이용해서 기존의 언어에서 제공되었던 C++ 98 라이브러리가 크게 향상되었기 때문이지요.

추가된 라이브러리의 목록을 보려면 라이브러리 요소 목록을 참조하세요

## 2.7 C++ 11 가 노력한 목표는 무엇인가요?

C++은 시스템 프로그래밍 쪽으로 살짝 특화되어 있는 범용 프로그래밍 언어로,

- C 보다 낫고
- 데이터 추상화를 지원하고
- 객체 지향 프로그래밍을 지원하고
- 일반화(generic) 프로그래밍을 지원

합니다.

C++ 11을 만들기 위한 노력의 목표로 다음을 강화하는 것이었습니다.

- C++을 시스템 프로그래밍과 라이브러리 제작에 더 좋은 언어로 만든다.
- C++을 좀더 쉽게 배우고 이해하게 만든다.
- 이러한 과정은 이전 버전과 호환성을 염밀히 고려하며 수행되었습니다. 위원회가 이러한 방침을 깨뜨릴 경우는 오직 새로운 키워드 (static\_assert, nullptr, constexpr) 을 도입할 때 뿐입니다.

자세한 설명을 원하는 분들은

- B. Stroustrup: **What is C++11?**. CVu. Vol 21, Issues 4 and 5. 2009.
- B. Stroustrup: **Evolving a language in and for the real world: C++ 1991-2006**. ACM HOPL-III. June 2007.
- B. Stroustrup: **A History of C++**: 1979-1991. Proc ACM History of Programming Languages conference (HOPL-2). March 1993.
- B. Stroustrup: **C and C++: Siblings**. The C/C++ Users Journal. July 2002.

## 2.8 위원회가 추구한 디자인 목표는 무엇인가요?

당연하게도, 표준 과정에서는 저마다 목표가 다른 사람들과 그룹들이 참여를 하였고, 세세한 목표역시 시간에 따라 변화해갔습니다. 하지만 일단 여기서 C++ 11에 적합한 라이브러리와 기능들에 대한 기준을 소개하고자 합니다.

- 유지 보수 안정성(maintain stability) 과 호환성 - 옛날 코드를 파괴 하지 말자. 반드시 그래야만 하는 경우에는 언급해줘야 함
- 언어 확장 보다는 라이브러리를 선호한다 - 이상적인 것으로, 사실 위원회는 이에 완전히 성공적이지는 않았다. 위원회의 너무 많은 사람들이 ‘실제 언어 기능’들을 선호하였다.
- 일반성(generality) 를 특수화 보다 선호한다 - 추상화 메커니즘(클래스, 템플릿 등)을 항상시키는데 집중한다.
- 초보자와 전문가 모두 지원한다 - 초보자들은 라이브러리와 일반화 된 규칙을 이용해서 도움 받을 수 있고, 전문가들은 일반화 되고 효율적인 기능을 이용하면 된다.
- 타입 안정성 증대 - 프로그래머로 하여금 타입 안전하지 않는 기능들을 피하도록 하는 방식으로 지원
- 성능 향상과 직접적인 하드웨어 제어 기능 - C++ 을 임베디드 시스템 프로그래밍과 고성능 연산에 적합하도록 한다.
- 실제 세계를 고려 - 툴 체인, 구현 비용, 전환 문제, ABI 문제, 언어 교육 문제 등

사실 새로운 기능과 옛날 기능들을 통합하는 문제야 말로 우리의 작업의 대부분을 차지했습니다. 전체 언어 자체를 보면 각 부분들을 단순히 합쳐놓은 것에 불과하지요.

위원회가 목표하였던 것을 다른 관점에서 본다면 다음과 같습니다.

- 머신 모델과 동시성(concurrency) - 현대 하드웨어를 사용하는데 좀더 나은 방식을 제공합니다 (e.g 멀티 코어나 약하게 coherent 한 메모리 모델 등). 예를 들어 쓰레드 ABI, future, thread-local 저장, atomics ABI 등이 있습니다.
- 일반화 프로그래밍 (generic programming, GP) - GP 는 C++ 98 의 가장 성공적인 부분이 아닐 수 없습니다. 우리는 경험을 토대로 이 것에 대한 지원을 향상 시킬 필요성이 있었습니다. 그 예로 auto 와 템플릿 별명이 있습니다.

- 시스템 프로그래밍 - 하드웨어에 가까운 프로그래밍 (예를 들어 저수준의 임베디드 시스템 프로그래밍) 과 효율성 증가. 예를 들어 `constexpr`, `std::array`, 그리고 일반화 된 POD 들이 있습니다.
- 라이브러리 설계 - 추상화 메커니즘에서의 한계, 비효율성, 그리고 비규칙성을 없앴습니다. 예를 들어 인라인 네임스페이스, 생성자 상속, 우측값 참조 등이 있습니다.

## 2.9 위원회 보고서들을 어디서 찾을 수 있나요?

위원회 웹사이트의 문서 섹션으로 가시면 됩니다. 아마 거기서 여러분들은 엄청난 정보에 파묻혀 버릴 텐데, 목록에서 “issues list” 와 “State of” 를 참고하시면 편합니다. (예를 들어 [State of Evolution](#) (July 2008)) 주요 그룹들로

- 코어 (CWG) - 언어 기술적인 문제에 대해 다룹니다
- 진화 (EWG) - 언어의 새로운 기능에 대한 제안이나, 언어/라이브러리 범주를 벗어나는 것들에 대해 다룹니다
- 라이브러리 (LWG) - 라이브러리 관련한 것들을 다룹니다.

여기에 가장 최근의 [C++ 11 표준 드래프트](#)가 있습니다.

## 2.10 C++ 11 에 대한 전문적이고 기술적인 문서들을 어디서 찾을 수 있나요?

- Bjarne Stroustrup: [Software Development for Infrastructure](#). Computer, vol. 45, no. 1, pp. 47-58, Jan. 2012, doi:10.1109/MC.2011.353. A video interview about that paper and [video of a talk](#) on a very similar topic (That's a 90 minute talk incl. Q&A).
- Saeed Amrollahi: [Modern Programming in the New Millennium: A Technical Survey on Outstanding features of C++0x](#). Computer Report (Gozarsh-e Computer), No.199, November 2011 (Mehr and Aban 1390), pages 60-82. (in Persian)
- Mark Batty et al's: [Mathematizing C++ concurrency](#), POPL 2012. // thorough, precise, and mathematical.
- Gabriel Dos Reis and Bjarne Stroustrup: [General Constant Expressions for System Programming Languages](#). SAC-2010. The 25th ACM Symposium On Applied Computing.
- Hans-J. Boehm and Sarita V. Adve: [Foundations of the C++ concurrency memory model](#). ACM PLDI'08.
- Hans-J. Boehm: [Threads Basic](#). HPL technical report 2009-259 // “what every programmer should know about memory model issues”

- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: [Concepts: Linguistic Support for Generic Programming in C++](#). OOPSLA'06, October 2006. // The concept design and implementation as it stood in 2006; it has improved since, though not sufficiently to save it.
- Douglas Gregor and Jaakko Jarvi: [Variadic templates for C++0x](#). Journal of Object Technology, 7(2):31-51, February 2008.
- Jaakko Jarvi and John Freeman: [Lambda functions for C++0x](#). ACM SAC '08.
- Jaakko Jarvi, Mat Marcus, and Jacob N. Smith: Programming with C++ Concepts. Science of Computer Programming, 2008. To appear.
- M. Paterno and W. E. Brown : Improving Standard C++ for the Physics Community. CHEP'04. // Much have been improved since then!
- Michael Spertus and Hans J. Boehm: [The Status of Garbage Collection in C++0X](#). ACM ISMM'09.
- Verity Stob: [An unthinking programmer's guide to the new C++ – Raising the standard](#). The Register. May 2009. (Humor (I hope)).
- [N1781 = 05 – 0041] Bjarne Stroustrup: [Rules of thumb for the design of C++0x](#).
- Bjarne Stroustrup: [Evolving a language in and for the real world: C++ 1991-2006](#). ACM HOPL-III. June 2007. (incl. slides and videos). // Covers the design aims of C++0x, the standards process, and the progress up until 2007.
- B. Stroustrup: [What is C++0x?](#). CVu. Vol 21, Issues 4 and 5. 2009.
- Anthony Williams: [Simpler Multithreading in C++0x](#). devx.com.

이 목록은 아직 미완성이고, 사람들이 새로운 문서를 냄에 따라 구식의(out-of-date)의 것이 될 것입니다. 만일 여러분이 이 목록에 포함시키고자 하는 문서가 있다면 꼭 보내주시기 바랍니다. 또한, 표준의 가장 최근의 항상 내용을 위 문서들이 모두 포함하고 있지는 않을 것입니다. 아무튼 저는 위 내용을 가장 최신으로 유지하도록 노력하겠습니다.

## 2.11 C++ 11 에 대해 어디서 읽을 수 있나요?

C++ 11에 대한 내용이 표준안이 거의 완성됨에 따라 증가하고 있고, C++ 구현들이 새로운 언어 기능과 라이브러리를 제공하기 시작하고 있습니다. 아래는 읽어볼 자료 리스트입니다.

- B. Stroustrup: [The C++ Programming Language \(Fourth Edition\)](#).
- the papers section of the committee's website.
- [C++11 draft](#).
- the [C++11 Wikipedia entry](#). (비록 위원회 멤버들이 하는 것 같지는 않지만 지속적으로 관리되고 있는 것처럼 보임)
- A list of support for C++11 features.

## 2.12 C++ 11에 대한 영상 자료들은 있나요?

(저를 잘 아시는 분들이라는 알겠지만, 이 질문이 제가 직접 자문 자답하는 것이 아닌 진짜 FAQ 임을 알 수 있습니다. 저는 기술적인 문제에 대한 영상을 보는 것에 대해 좋게 생각하지 않습니다. 제가 느끼기에 비디오는 집중하기에 힘들고, 구술로 설명하기 때문에 세세한 기술적인 오류들을 포함하고 있을 때가 많습니다)

물론 있지요.

- B. Stroustrup, H. Sutter, H-J. Boehm, A. Alexandrescu, S.T.Lavavej, Chandler Carruth, and Andrew Sutter: [several talks and panels from the Going Native 2012 conference](#).
- Herb Sutter: [Writing modern C++ code: how C++ has evolved over the years](#). September 2011.
- Herb Sutter: [C++ and Beyond 2011: Herb Sutter - Why C++?](#). August 2011.
- Try Google videos.
- Lawrence Crowl: [Lawrence Crowl on C++ Threads](#). in Sophia Antipolis, June 2008.
- Bjarne Stroustrup: [The design of C++0x](#) at U of Waterloo in 2007.
- Bjarne Stroustrup: [Initialization](#) at Google in 2007.
- Bjarne Stroustrup: [C++0x – An overview](#). in Sophia Antipolis, June 2008.
- Lawrence Crowl: [Threads](#).
- Roger Orr: [C++0x](#). January 2008.
- Hans-Jurgen Boehm: [Getting C++ Threads Right](#). December 2007.

## 2.13 C++ 11은 배우기 어렵나요?

C++의 이전의 중요한 기능들은 계속 계승해 왔기 때문에, C++ 11은 C++ 98 보다 크기가 더 큽니다. 따라서, 세세하게 모든 것을 배우려고 한다면 C++ 11을 배우는 것이 C++ 98 보다 더 어렵겠지요. 이 때문에 배우는 사람 입장에서 다음의 두 가지 방법으로 배울 수 있습니다.

- 일반화 : 한 개의 좀더 일반적인 규칙을 배운다. (예를 들어 일원화 된 초기화, 생성자 상속, 쓰레드)
- 간단한 대응책들 : 이전의 방법보다 보다 더 새로운 방법을 사용합니다. (예를 들어 array, auto, range-for 문, <regex>)

명백하게도, 밑바닥부터 위로 훑어가는(bottom - up) 방식으로 공부를 한다면 위와 같은 방법은 사용할 수 없겠고, 위와 같은 방법 말고 다른 방식으로 가르치는 자료들은 현재 까지는 별로 없습니다.

## 2.14 위원회는 어떻게 운영되나요?

ISO 표준 위원회, SC22 WG21은 이러한 위원회를 위한 ISO 규칙을 바탕으로 운영됩니다. 흥미로운 사실은 이 규칙들은 아직 표준화 되지 않았을 뿐더러 오랫동안 바뀌지도 않았습니다.

많은 나라들은 활동적인 C++ 그룹을 가진 표준화 그룹들이 있습니다. 이 그룹들 기리, 회의를 가지며, 웹을 통해 조율을 하고, ISO 회의에 몇몇의 대표를 보내기도 합니다. 캐나다, 프랑스, 독일, 스위스, 영국, 그리고 미국의 대표들이 주로 회의에 참석하지요. 덴마크, 네덜란드, 일본, 노르웨이, 스페인 등의 나라들에서는 가끔 대표를 보내는 편입니다.

대부분의 작업들은 인터넷을 통한 회의를 통해 진행되며, 그 결과들은 위원회 보고서에 기록되어 보관되게 됩니다. 이들은 [WG21 웹사이트](#)에서 확인할 수 있습니다.

위원회는 일년에 일주일 정도 두 세번씩 실제로 만남을 가집니다. 이 회의에서 대부분의 작업들은 이 서브 그룹들 (Core, Library, Evolution, Concurrency 등)에서 진행이 됩니다. 필요에 따라 특별히 빠르게 처리해야 할 주제(예를 들어 concepts나 메모리 모델 등)가 있다면 즉석으로 작업 그룹(working group)들을 만들 때도 있습니다. 투표는 주 회의에서 진행됩니다. 먼저 작업 그룹이 비공식 여론 조사(straw vote)를 통해 프레젠테이션이 위원회 전체에 제공될 준비가 되어 있는지 확인한 다음에, 위원회 전체 회원들이 투표를 한 다음, 무언가가 통과되었다면 국가들이 투표를 합니다. 우리는 여기서 매우 신중을 기해 위원회 대다수는 찬성하는 반면 국가들이 모두 반대하는 식의 문제는 발생하지 않도록 합니다. 왜냐하면 이러한 일이 이러나기 이전에, 오랜 논의를 통해서 대부분 해결하기 때문이지요. 공식적인 드래프트의 최종 투표는 각 국가의 표준 그룹에서 메일을 보내는 식으로 이루어집니다.

위원회는 공식적으로 C 표준 그룹(SC22 WG14)과 POSIX 연락을 주고 받고 있으며 몇 개의 다른 그룹들과도 공식적인 만남을 진행하고 있습니다.

## 2.15 위원회에는 누가 있나요?

위원회는 200명의 사람들로 구성되어 있으며, 그 중 60명 가량이 일년에 2~3차례 열리는 일주일 동안 지속되는 정기 회의에 참석합니다. 게다가, 일부 국가에서 열리는 국가 표준 그룹 간의 회의들도 있습니다. 대부분의 위원회 멤버들은 회의에 참석하던지, 이메일 토론에 참석하던지, 혹은 위원회에 보고서를 보내는 방식으로써 기여를 합니다. 대부분의 멤버들은 그들을 도와주는 동료나 친구들이 있습니다. 위원회가 만들어진 첫번째 날부터, 위원회는 세계 각국으로부터 회원을 받았습니다. 최종 투표는 20개의 국가 표준 그룹에서 진행됩니다. 따라서 ISO C++ 표준은 꽤나 거대한 노력의 집합체로, ‘자기 자신과 같은 사람들을 위한’ 프로그래밍 언어를 만드는 소규모의 폐쇄적인 그룹이 아니라는 의미입니다. 위원회는 다시 말해 모두가 쉽게 사용할 수 있는 것들을 만들고자 최선을 다하는 자원 봉사자들의 모임이라 할 수 있습니다.

당연히도, 많은 (하지만 전부는 아닌) 자원 봉사자들이 C++에 관련된 직업을 가지고 있습니다. 우리의 그룹에는 컴파일러 제작가, 툴 제작가, 라이브러리 제작가, 응용 프로그램 제작가, 연구자, 자문가 등이 있습니다.

다음은 위원회에 포함된 단체들에 목록입니다 : *Adobe, Apple, Boost, Bloomberg, EDG, Google, HP, IBM, Intel, Microsoft, Red Hat, Sun*.

여기에 여러분이 인터넷에서 한 번쯤은 보았을 멤버 회원들의 목록입니다 : *Dave Abrahams, Matt Austern, Pete Becker, Hans Boehm, Steve Clamage, Lawrence Crowl, Beman Dawes, Francis Glassborow, Doug Gregor, Pablo Halpern, Howard Hinnant, Jaakko Jarvi, John Lakos, Alisdair Meredith, Jens Maurer, Jason Merrill, Sean Parent, P.J. Plauger, Tom Plum, Gabriel Dos Reis, Bjarne Stroustrup, Herb Sutter, David Vandevoorde, Michael Wong.* 여기에 포함되지 않는 200+ 의 현재 및 과거의 멤버들에 사과의 말을 전합니다. 또한 여러 보고서들의 저자들의 이름들도 기억해주시기 바랍니다. 표준안은 익명의 위원회에서 만들어진 것이 아니라 많은 수의 개개인이 함께 모여 작성된 것입니다.

여러분들은 [WG21 문서들](#)을 살펴보는 것을 통해 이 일에 관련된 많은 전문가들의 이름을 볼 수 있을 것입니다. 다만, 표준에 기여한 많은 사람들 중 일부는 그다지 많은 문서들을 제출하지 않는 경우도 있습니다.

## 2.16 C++1y 가 있을 것인가요?

거의 확실합니다. 이는 물론 위원회가 C++0x 의 데드라인을 맞추지 못해서 그런 것이 아닙니다. C++ 14 은 마이너한 수정과 약간의 발전(이 기능들의 초안들은 투표에 부쳐졌고, 구현 되었습니다) 을 도모할 것이고, 2017 년의 C++ 17 에서 대폭적인 변화가 있을 것입니다.

## 2.17 “concepts” 는 어떻게 되었나요?

“concepts” 는 원래 템플릿 인자들이 정확히 어떠한 요구 조건을 충족하도록 하게 하는 것으로 만들어졌습니다. 하지만 불행이도, 위원회는 concept 들을 개발하는 과정이 표준안을 만드는 것을 심각하게 지연 시킬 수 있다고 결정해서, 이러한 기능을 현재의 표준안에서 지우기로 하였습니다. 저의 [The C++0x “Remove Concepts” Decision](#) 노트와 [A DevX interview on concepts and the implications for C++0x](#) 를 참조하시기 바랍니다.

비약적으로 간단화된 버전인 ‘concepts lite’ 는 C++ 14 의 일부분이 될 것입니다.

저는 이 문서로 부터 concept 섹션을 지우지 않고, 뒤에 다음과 같은 것들을 남겨 놓았습니다.

- axioms (semantic assumptions)
- concepts
- concept maps

## 2.18 혹시 원하지 않는 기능이 있나요?

네 있습니다. 또한 C++ 98 에도 제가 좋아 하지 않는 기능들 (예를 들어 매크로) 이 있습니다. 이 문제는 제가 좋아하느냐, 아니면 제가 필요성을 느끼느냐에 관련한 문제가 아닙니다. 이 문제는 만일 어떤 사람이 이러한 기능을 충분히 필요하다고 느껴서 다른 사람들로 하여금 이 기능을 유지시키도록 설득시키거나, 혹은 이러한 기능의이 특정 유저 커뮤니티에 꼭 필요해서 유지 시킬 이유가 있는지의 문제입니다.

## 2.19 \_\_cplusplus

C++ 11 에는 매크로 `__cplusplus` 가 현재의 값인 199711L 과 다른 (이 보다 큰) 값으로 설정될 예정입니다.

## 2.20 auto – 초기화자 부터 타입을 유추

예를 들어

---

```
auto x = 7;
```

---

위 코드에서 x 는 int 타입이 될 것입니다. 왜냐하면 초기화자(initializer) 의 타입이 int 이기 때문이죠. 일반적으로 우리는

---

```
auto x = expression;
```

---

와 같이 한다면 x 는 “expression 의 타입이 될 것입니다. auto 를 이용하는 가장 주된 이유는 어떠한식의 타입이 알기 어렵거나 쓰기에 매우 길 경우입니다. 예를 들어

---

```
template<class T> void printall(const vector<T>& v)
{
    for (auto p = v.begin(); p!=v.end(); ++p)
        cout << *p << "\n";
}
```

---

와 같은 코드를 생각해봅시다. C++ 98 에서는 위 코드를

---

```
template<class T> void printall(const vector<T>& v)
{
    for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
        cout << *p << "\n";
}
```

---

으로 써야만 했습니다. 또한 변수의 타입이 템플릿 인자에 의해 좌우 될 때, 이를 auto 를 사용하지 않고 코드로 쓰려면 매우 까다롭습니다. 예를 들어 아래와 같은 코드를 생각해봅시다.

---

```
template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu)
{
    // ...
    auto tmp = vt[i]*vu[i];
    // ...
}
```

---

tmp 의 타입은 T 와 U 를 곱한 것의 타입이 되어야만 하며, 이는 사람이 읽었을 때 무엇인지 생각하기에 까다롭습니다. 하지만 당연하게도 컴파일러는 어떠한 T 와 U 에 대해 처리하는지 알기 때문에 컴파일러는 무슨 타입이 될 지 알겠지요.

`auto` 는 다른 기능들 중에서도 가장 먼저 제안되었고 구현되었습니다. 저는 1984년 초반에 이를 구현하였지만, C와 호환성 문제 때문에 사용할 수는 없었습니다. 이러한 호환성 문제는 C++98 과 C99에서 암시적 `int(implicit int)` 라는 것을 삭제함으로써 해결되었습니다. 이제 두 언어 모두 모든 변수와 함수가 반드시 명확한 타입으로 정의되어야만 합니다<sup>1</sup>. `auto` 의 옛날 의미(이 변수는 지역 변수이다)는 이제 사용할 수 없습니다. 몇몇의 위원회 회원들이 수백만 줄에 달하는 코드를 검토한 끝에 오직 몇개의 오류(즉 `auto` 의 의미가 바뀜으로서 생기는 오류)만을 발견할 수 있었습니다 - 그리고 개들 중 대부분은 테스트버전의 코드나 버그로 판단되었습니다.

`auto` 키워드의 역할 자체는 코드에서 길게 써야 하는 것을 간단히 줄이는 것이기 때문에 표준 라이브러리 사용법에는 영향은 없습니다.

다음 글들도 참고해보세요.

- the C++ draft section 7.1.6.2, 7.1.6.4, 8.3.5 (for return types)
- [N1984 = 06 – 0054] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: [Deducing the type of variable from its initializer expression](#) (revision 4).

## 2.21 Range-for 문

range for 문은 여러분이 “범위(range) 안에서 마치 STL 의 `begin()` 과 `end()` 에서 하는 것처럼 반복(iterate) 할 수 있도록 해줍니다. 모든 표준 컨테이너들은 range 로 사용될 수 있고, `std::string`, 초기화자 리스트, 배열, 그리고 여러분이 `begin()` 과 `end()` 를 정의할 수 있는 모든 것(예를 들어 `istream`)들도 가능합니다.

---

```
void f(vector<double>& v)
{
    for (auto x : v) cout << x << '\n';
    for (auto& x : v) ++x; // 레퍼런스를 이용해서 값을 바꾼다.
}
```

---

여러분은 위 코드를 “`v` 안에 있는 모든 `x` 에 대해 `v.begin()` 부터 `v.end()` 까지 반복하는 것으로 생각할 수 있습니다. 다른 예시로:

---

```
for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';
```

---

`begin()` (과 `end()`) 는 `x.begin()` 형태로 멤버 함수로써 호출되거나, 자유 함수인 `begin(x)` 로 호출될 수 있습니다. 다만, 멤버 함수 버전이 우선순위가 높습니다.

다음 글들도 참고해보세요.

- the C++ draft section 6.5.4 (note: changed not to use concepts)
- [N2243 = 07 – 0103] Thorsten Ottosen: [Wording for range-based for-loop](#) (revision 2).

---

<sup>1</sup>기존에는 타입이 명시적으로 정의되지 않았다면 암시적으로 `int` 라 가정됨

- [N3257 = 11 – 0027] Jonathan Wakely and Bjarne Stroustrup: Range-based for statements and ADL (Option 5 was chosen).

## 2.22 괄호

다음과 같은 코드를 생각해봅시다.

---

```
list<vector<string>> lvs;
```

---

C++ 98에서 위는 문법 오류를 발생 시켰습니다. 왜냐하면 두 개의 > 사이에 공간이 없었기 때문이지요(즉 쉬프트 연산자 >>로 해석되었음). C++ 11은 두 개의 템플릿 사이에서 발생하는 위와 같은 것을 올바르게 해석합니다. 사실 이와 같은 문제가 발생하였던 이유는, 컴파일러의 가장 최전선에서의 과정은 parse 와 stage 라는 두 단계로 이루어 졌기 때문입니다. 아래는 가장 단순한 모델입니다.

- 어휘 분석 (lexical analysis) (문자들로 부터 토큰을 생성한다)
- 문법 해석 (문법을 체크한다)
- 타입 체크 (타입들의 이름들과 문장들을 찾아낸다)

이 단계들은 이론적으로, 그리고 실제로 엄밀하게 분리되어 있으므로, 어휘 분석기가 >>를 하나의 토큰으로 인식하여 이게 실제로 무슨 의미인지, 특히 여러개의 템플릿을 사용할 때 발생하는 것인지 전혀 알 수 없게 되었습니다. 하지만, 위와 같은 문제를 올바르게 처리하기 위해서는 각 단계에서 서로 정보를 주고받아야만 합니다. 이러한 문제를 해결 할 수 있게된 중요한 발견은 사실 모든 C++ 컴파일러들이 이미 위 문제를 인식하고 있었고 좋은 오류 메세지를 출력하고 있다는 점이었습니다.

다음 글들을 읽어보세요.

- the C++ draft section ???
- [N1757 == 05 – 0017] Daveed Vandevoorde: revised right angle brackets proposal (revision 2).

## 2.23 디폴트 제어하기: default 와 delete

“복사를 막는다”를 가장 쉽게 수행하는 방법은

---

```
class X {
    // ...
    X& operator=(const X&) = delete; // 복사를 불허(disallow) 한다
    X(const X&) = delete;
};
```

---

역으로, 우리는 디폴트 복사 방법을 아래와 같이 명시적으로 지정할 수 있습니다.

---

```

class Y {
    // ...
    Y& operator=(const Y&) = default; // 복사를 디폴트로 한다
    Y(const Y&) = default;
}

```

---

사실 디폴트를 명시적으로 나타내는 것은 불필요합니다. 하지만, 복사 연산에 대한 주석이나, 사용자가 (비록 디폴트 복사 연산과 동일할지라도) 복사 연산을 직접 정의하는 일은 드물지 않습니다. 컴파일러로 하여금 디폴트 연산을 알아서 구현하라고 놔두는 것이 더 간단하고, 오류도 적고, 더 좋은 목적 코드를 생성할 수 있습니다. “default” 메커니즘은 default 형태를 가지고 있는 어떤 함수에 대해서라도 사용할 수 있습니다. 그리고 “delete” 메커니즘은 아무 함수에나 사용할 수 있습니다. 예를 들어, 우리는 다음과 같은 원하지 않는 타입 변환을 막을 수 있습니다.

---

```

struct Z {
    // ...

    Z(long long); // long long 으로 초기화 할 수 있다
    Z(long) = delete; // 그 외의 것들은 막는다
}

```

---

다음 글들도 참고해보세요.

- the C++ draft section ???
- [N1717 = 04 – 0157] Francis Glassborow and Lois Goldthwaite: [explicit class and default definitions \(an early proposal\)](#).
- Bjarne Stroustrup: [Control of class defaults \(a dead end\)](#).
- [N2326 = 07 – 0186] Lawrence Crowl: [Defaulted and Deleted Functions](#).
- [N3174 = 100164] B. Stroustrup: [To move or not to move](#). An analysis of problems related to generated copy and move operations. Approved.

## 2.24 디폴트 제어하기 : move 와 copy

디폴트로, 클래스에는 5 개의 연산자 정의되어 있습니다 :

- 복사 대입(copy assignment)
- 복사 생성(copy constructor)
- 이동 대입(move assignment)
- 이동 생성(move constructor)

- 소멸(destructor)

만일 위들 중 어떤 것을 선언 하더라도 여러분은 전부를 명시적으로 정의하거나, 여러분이 원하는 것을 디폴트 처리하도록 고려해야 합니다. 복사, 이동, 그리고 소멸 연산이 모두 인접하게 연관된 작업이라는 사실을 고려한다면, 각 작업을 자유롭게 섞고 대응 시키기 보다는 일부 조합 만이 의미론상으로(semantically) 말이 되게 됩니다. 만일 이동, 복사, 소멸 작업 중 어느 하나라도 사용자에 의해 명시적으로 정의(선언, =default 혹은 =delete) 되었다면 이동 작업은 디폴트로 생성되지 않습니다. 또한 이동, 복사, 소멸 작업 중 어느 하나라도 사용자에 의해 명시적으로 정의되었다면, 선언 되지 않은 복사 연산은 디폴트로 정의되지만 이 기능은 사용되지 않을 예정 (deprecated)<sup>2</sup> 이므로 권장하지 않습니다. 예를 들어서;

---

```
class X1 {
    X1& operator=(const X1&) = delete; // 복사를 불허한다
};
```

---

위 문장은 암시적(implicitly) 으로 X1 들의 이동 연산을 불허합니다. 하지만 복사 생성은 사용할 수 있고, 이 기능은 사용되지 않을 예정입니다.

---

```
class X2 {
    X2& operator=(const X2&) = default;
};
```

---

위 코드 또한 X2 들의 이동 연산을 암시적으로 불허합니다. 마찬가지로 복사 생성은 사용할 수 있고, 이 기능은 사용되지 않을 예정입니다.

---

```
class X3 {
    X3& operator=(X3&&) = delete; // 이동을 불허한다
};
```

---

위는 X3 의 복사를 암시적으로 막습니다.

---

```
class X4 {
    ~X4() = delete; // 소멸을 불허한다
};
```

---

위는 X4 들의 이동 연산을 암시적으로 막습니다. 복사는 사용할 수 있지만 이 기능은 사용되지 않을 예정입니다. 저는 여러분들이 위 5 개의 함수 중 하나라도 정의한다면, 모두 명시적으로 정의하기를 권합니다. 예를 들어서:

---

```
template<class T>
class Handle {
    T* p;
public:
    Handle(T* pp) : p{pp} {}
    ~Handle() { delete p; } // 사용자 정의 소멸자 : 암시적 복사 혹은 이동은 없다
```

---

<sup>2</sup>deprecated 를 번역할 만한 적당한 좋은 단어가 없으므로 ‘사용되지 않을 예정이다’ 라 번역하겠습니다

```

Handle(Handle&& h) :p{h.p} { h.p=nullptr; } // 소유권을 전달한다

// 소유권을 전달한다
Handle& operator=(Handle&& h) { delete p; p=h.p; h.p=nullptr; return *this; }

Handle(const Handle&) = delete; // 복사 없음
Handle& operator=(const Handle&) = delete;

// ...
};

```

---

### 참고 자료

- the C++ draft section ???
- [N2326 == 07 – 0186] Lawrence Crowl: [Defaulted and Deleted Functions](#).
- [N3174 = 100164] B. Stroustrup: [To move or not to move](#). An analysis of problems related to generated copy and move operations. Approved.

## 2.25 enum 클래스

enum 클래스는 기존의 C++ enum에서의 3 가지 문제점을 보여줍니다.

- 기존의 enum은 암시적으로 int로 타입 변환이 되어서, enum이 정수로써 작동되기를 원하지 않을 때 오류를 발생시켰습니다.
- 기존의 enum은 허용되는 범위(scope)<sup>3</sup> 밖에서도 그 열거 이름들을 사용할 수 있었기 때문에 이름 간의 충돌이 발생하였습니다.
- 열거형의 실제 타입을 명시할 수 없기 때문에 혼란을 발생시키고, 호환성 문제도 만들고, 전진 선언(foward declaration)도 불가능 하였습니다.

enum 클래스는 타입과 그 정의 범위가 명확 합니다.

---

```

enum Alert { green, yellow, election, red }; // 이전의 enum

enum class Color { red, blue }; // 범위가 정해져있고, 타입이 잘 정해진 enum
                                // 이를 포함하고 있는 범위 바깥에서는 열거형을 사용 불가
                                // int로 암시적으로 변환되지 않는다.

enum class TrafficLight { red, yellow, green };

```

<sup>3</sup>scoped 를 적절하게 옮길 단어(– 간단히 말해 C++에서 같은 {} 블록 안에 있는 것들)가 없으므로 ‘범위의’의 뜻으로 옮기도록 하겠습니다. 예를 들어 scoped member라 하면 특정 범위 안의 멤버, scoped variable이라 하면 특정 범위 안의 변수라는 의미입니다.

```
Alert a = 7; // 오류 (모든 C++ 버전에서)
Color c = 7; // 오류 (int에서 Color로 변환 불가)
```

```
int a2 = red; // 가능 (Alert에서 int로 변환 가능)
int a3 = Alert::red; // C++ 98에서는 오류. C++ 11에서는 가능
int a4 = blue; // 오류 : blue가 범위에 없다
int a5 = Color::blue; // 오류 : Color에서 int로 변환 불가능
```

---

```
Color a6 = Color::blue; // 가능
```

위에서 나타나듯이 기존의 enum 역시 동일하게 작동하지만 이제 여러분은 enum 이름에 특별한 키워드들을 넣을 수 있게 되었습니다. 새롭게 도입한 열거형의 이름이 “enum class”인 이유는 전통적인 enum의 특징들(값을 가진 이름들)과 클래스의 특징(범위를 가진 멤버들과, 암시적 변환의 부재)을 합친 형태입니다.

enum의 타입 자체를 명확하게 정의하게 된다면, 열거형의 크기를 보장하고, 다른 것들과 쉽게 같이 사용(interoperability)할 수 있습니다.

---

```
enum class Color : char { red, blue }; // 함축된 표현
```

```
enum class TrafficLight { red, yellow, green }; // 디폴트로 내장 타입은 int로 처리된다.
```

```
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFF0U }; // E의 크기가 얼마나 클까요?
// 이전의 규칙에 따르면 -
// 구현에 따라 다르다(implementaion defined)
```

---

enum EE : unsigned long { EE1 = 1, EE2 = 2, Ebig = 0xFFFFFFFF0U }; // 이제 구체적으로 정할 수 있습니다

---

이 또한 enum을 전진 선언하는 것도 가능하게 합니다.

---

```
enum class Color_code : char; // (전진) 선언
void foobar(Color_code* p); // 전진 선언을 사용
// ...
enum class Color_code : char { red, yellow, green, blue }; // 정의
```

enum의 내장 타입은 반드시 부호 있는/없는 정수 타입이어야만 합니다. 디폴트는 int입니다. 표준 라이브러리에서 enum class는 다음과 같은 경우로 사용됩니다.

- 시스템의 특별한 오류 코드들을 매핑하기 위해서 (enum class errc)
- 포인터 안정성 표시자(indicator) (enum class pointer\_safety {relaxed, preferred, strict}; )
- I/O 스트림 오류들 (enum class io\_errc {stream = 1 }; )

- 비동기화 통신에서의 오류 처리를 위해 (enum class future\_errc {broken\_promise, future\_already\_retrieved, promise\_already\_satisfied}; )

이들 중 많은 것들이 == 와 같은 연산자들을 가지고 있습니다.

다음 글들도 읽어보세요

- the C++ draft section 7.2
- [N1513 = 03 – 0096] David E. Miller: Improving Enumeration Types (original enum proposal).
- [N2347 = J16/07 – 0207] David E. Miller, Herb Sutter, and Bjarne Stroustrup: Strongly Typed Enums (revision 3).
- [N2499 = 08 – 0009] Alberto Ganesh Barbati: Forward declaration of enumerations.

## 2.26 constexpr – 일반화(generalized) 된, 그리고 보장된(guaranteed) 상수 표현식

constexpr 메커니즘은

- 좀 더 일반적인 상수 표현식을 제공합니다.
- 사용자 정의 타입을 포함한 상수 표현식을 사용할 수 있게 됩니다.
- 초기화가 컴파일 시간(compile time)<sup>4</sup>에 수행되도록 보장합니다.

아래와 같은 코드를 고려해봅시다.

---

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2) { return Flags(int(f1)|int(f2)); }

void f(Flags x)
{
    switch (x) {
        case bad: /* ... */ break;
        case eof: /* ... */ break;
        case bad|eof: /* ... */ break;
        default: /* ... */ break;
    }
}
```

---

여기서 constexpr 은 함수로 하여금 간단한 형태여서 컴파일 시에 인자로 상수 식이 들어가게 된다면 그 값이 계산될 수 있도록 합니다. 게다가, 컴파일 시에 식이 계산 될 수 있게 하는 것 뿐만이 아니라, 우리는

<sup>4</sup>컴파일 시간이라 하는 것은, 컴파일 되는 도중을 의미합니다

식들이 컴파일 시에 계산될 수 있도록 **요구** 할 수 도 있습니다. 변수 앞에 **constexpr** 키워드가 그 역할을 하게 됩니다 (또한 상수라는 의미도 부여) :

---

```
constexpr int x1 = bad|eof; // 가능

void f(Flags f3)
{
    constexpr int x2 = bad|f3; // 오류 : 컴파일 시간에 계산할 수 없다
    int x3 = bad|f3; // 가능
}
```

---

통상적으로 우리는 컴파일 시간 계산이 오직 전역 혹은 네임 스페이스 객체 - 즉 우리가 읽기 전용 저장 공간에 위치시키길 원하는 것들에만 적용되기를 원합니다. 이는 또한 생성자가 **constexpr** 가 될 수 있을 정도로 간단한 객체들에 대해서도 작동합니다.

---

```
struct Point {
    int x,y;
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point origo(0,0);
constexpr int z = origo.x;

constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2)};
constexpr int x = a[1].x; // x becomes 1
```

---

참고로 **constexpr** 은 **const** 를 일반적으로 대체할 수 있는 것은 아닙니다 (그 역도 마찬가지 이고요) :

- **const** 의 원래 역할은 어떠한 객체가 인터페이스를 통해서 값이 바뀌는 것을 원치 않는다는 것을 알려주는 역할을 합니다 (비록 그 객체가 다른 인터페이스를 통해 수정될 수 있을 수도 있겠지만). 따라서 어떠한 객체가 **const** 라고 알려주는 것은 컴파일러에게 좋은 최적화 할 수 있는 기회를 제공해주는 것입니다. 특히, 만일 객체가 **const** 로 선언되었고, 그 주소를 사용하지 않는다면, 컴파일러는 흔히 그 객체의 초기화자를 컴파일 시간에 계산하고 (비록 이 것이 보장되어 있지 않다고 해도) 객체를 생성된 코드에 넣기 보다는 테이블에 보관합니다.
- **constexpr** 의 주요 목적은 컴파일 시에 계산될 수 있는 것들의 범위를 더 늘려서 이와 같은 계산들이 타입-안전(type safe) 할 수 있도록 하기 위함입니다. **constexpr** 로 선언된 객체들의 초기화자들은 컴파일 시간에 계산됩니다. 이들은 단순히 컴파일러의 테이블에 보관되는 값들에 불과하며, 필요할 때만 생성된 코드에 들어가게 됩니다.

### 참고 자료

- the C++ draft 3.6.2 Initialization of non-local objects, 3.9 Types [12], 5.19 Constant expressions, 7.1.5 The **constexpr** specifier

- [N1521 = 03 – 0104] Gabriel Dos Reis: Generalized Constant Expressions (original proposal).
- [N2235 = 07 – 0095] Gabriel Dos Reis, Bjarne Stroustrup, and Jens Maurer: Generalized Constant Expressions – Revision 5.

## 2.27 decltype – 식(expression) 의 타입

`decltype(E)` 는 (선언된 타입 - declared type) 식 E 의 타입을 의미하며, 어떠한 것을 선언할 때 사용될 수 있습니다. 예를 들어서

---

```
void f(const vector<int>& a, vector<float>& b)
{
    typedef decltype(a[0]*b[0]) Tmp;
    for (int i=0; i<b.size(); ++i) {
        Tmp* p = new Tmp(a[i]*b[i]);
        // ...
    }
    // ...
}
```

---

이 표현은 `typeof` 라는 라벨 이름에서 오랫 동안 일반화 프로그램에서 매우 많이 사용되었는데, 사실 `typeof`로 이를 구현하는 일은 불완전하고, 호환성이 낮았기에, 표준 버전은 `decltype`로 이름 붙였습니다. 만일 여러분이 단순히 어떠한 변수의 타입만을 필요로 한다면 `auto`를 사용하는 것이 더 쉬운 방법입니다. 여러분이 진짜 `decltype`를 필요로 하는 경우는, 변수가 아닌 것의 타입(예를 들어 리턴 타입) 등의 경우입니다.

참고 자료

- the C++ draft 7.1.6.2 Simple type specifiers
- [Str02] Bjarne Stroustrup. Draft proposal for “`typeof`”. C++ reflector message c++std-ext-5364, October 2002. (original suggestion).
- [N1478 = 03 – 0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto (original proposal).
- [N2343 = 07 – 0203] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: Decltype (revision 7): proposed wording.

## 2.28 초기화자 리스트 - Initializer lists

아래와 같은 코드를 고려합시다.

---

```
vector<double> v = { 1, 2, 3.456, 99.99 };
list<pair<string,string>> languages = {
    {"Nygaard", "Simula"}, {"Richards", "BCPL"}, {"Ritchie", "C"}
```

```
};

map<vector<string>,vector<int>> years = {
    { {"Maurice", "Vincent", "Wilkes"}, {1913, 1945, 1951, 1967, 2000} },
    { {"Martin", "Ritchards"}, {1982, 2003, 2007} },
    { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

---

초기화자 리스트는 단순히 배열만을 위해 사용되는 것은 아닙니다. {} 리스트를 받는 메커니즘은 std::initializer\_list<T> 를 인자로 받는 함수(많은 경우 생성자)와 같습니다. 예를 들어서

```
void f(initializer_list<int>);

f({1,2});
f({23,345,4567,56789});
f({}); // 비어 있는 리스트
f{1,2}; // 오류 : 함수 호출 () 가 없습니다

years.insert({{"Bjarne", "Stroustrup"}, {1950, 1975, 1985}});
```

---

초기화자 리스트는 임의의 길이가 될 수 있지만 모두 같은 타입이어야만 합니다. (즉 모든 원소들이 템플릿 인자 타입 T 이거나, T 로 변환될 수 있어야만 합니다)

컨테이너는 초기화자 리스트 생성자를 다음과 같이 구현할 수 있습니다.

```
template<class E> class vector {
public:
    vector (std::initializer_list<E> s) // 초기화자 리스트 생성자
    {
        reserve(s.size()); // 올바른 크기의 공간을 얻는다

        // 원소들을 초기화 한다. (elem[0:s.size()) 안에 있는 것들을)
        uninitialized_copy(s.begin(), s.end(), elem);

        sz = s.size(); // vector 크기를 설정한다
    }

    // ... as before ...
};
```

---

직접적으로 초기화 하는 것과 복사 초기화의 차이는 {}를 이용한 초기화에서도 마찬가지로 적용됩니다. 예를 들어서 std::vector 에는 int 로 부터 생성하는 명시적 생성자와, 초기화자 리스트 생성자가 있습니다.

```
vector<double> v1(7); // 가능 : v1 에는 7 개의 원소가 있다
v1 = 9; // 오류 : int 를 vector 로 변환할 수 없다
vector<double> v2 = 9; // 오류 : int 를 vector 로 변환할 수 없다
```

```

void f(const vector<double>&);

f(9); // 오류 : int 를 vector 로 변환할 수 없다

vector<double> v1{7}; // 가능 : v1 에는 1 개의 원소가 있다 (값이 7 인)
v1 = {9}; // v1 에는 1 개의 원소가 있다 (값이 9인)
vector<double> v2 = {9}; // v2 에는 1 개의 원소가 있다 (값이 9 인)
f({9}); // f 는 리스트 {9} 로 호출되었다

vector<vector<double>> vs = {
    vector<double>(10), // 가능 : 명시적 생성 (10 개의 원소)
    vector<double>{10}, // 가능 : 명시적 생성 (값이 10 인 1 개의 원소)
    10 // 오류 : vector 의 생성자는 명시적
};

```

---

함수는 `initializer_list` 를 인자로 접근할 수 있습니다. 예를 들어서:

```

void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p) cout << *p << "\n";
}

```

---

`std::initializer_list` 만 인자로 취하는 생성자를 초기화자 리스트 생성자라고 부릅니다. 표준 라이브러리 컨테이너들, 문자열, 그리고 regex 는 모두 초기화자 리스트 생성자, 대입 연산자들이 정의되어 있습니다. 초기화자 리스트는 단일화되고 일반적인 초기화를 위한 조각 중 하나입니다.

참고 자료

- the C++ draft 8.5.4 List-initialization [dcl.init.list]
- [*N1890 = 05 – 0150*] Bjarne Stroustrup and Gabriel Dos Reis: [Initialization and initializers](#) (an overview of initialization-related problems with suggested solutions).
- [*N1919 = 05 – 0179*] Bjarne Stroustrup and Gabriel Dos Reis: [Initializer lists](#).
- [*N2215 = 07 – 0075*] Bjarne Stroustrup and Gabriel Dos Reis :[Initializer lists](#) (Rev. 3) .
- [*N2640 = 08 – 0150*] Jason Merrill and Daveed Vandevoorde: [Initializer Lists – Alternative Mechanism and Rationale \(v. 2\)](#) (final proposal).

## 2.29 줄어듬(narrowing) 방지하기

C 와 C++ 의 문제점으로 아래와 같은 것을 암시적으로 잘라내게 됩니다.

```

int x = 7.3; // Ouch!
void f(int);
f(7.3); // Ouch!

```

---

하지만 C++ 11 에서는 {} 초기화는 이와 같은 줄어듬(narrowing)<sup>5</sup>을 허용하지 않습니다.

---

```
int x0 {7.3}; // 오류 : 줄어듬
int x1 = {7.3}; // 오류 : 줄어듬
double d = 7;
int x2{d}; // 오류 : 줄어듬 (double에서 int로)
char x3{7}; // 가능 : 7이 비록 int이지만 이는 줄어듬이 아니다
vector<int> vi = { 1, 2.3, 4, 5.6 }; // 오류 : double에서 int로 줄어듬
```

---

C++ 11 이 많은 수의 호환성 문제를 방지할 수 있는 이유는 바로 가능한 초기화자의 타입이 아닌 실제 값을 비교해서 (예를 들어 위 코드에서 7과 같은 경우) 줄어듬인지 아닌지 판단하기 때문입니다. 만일 어떠한 값이 목표한 타입과 정확히 동일하게 같은 값으로 표현될 수 있다면, 그 타입 변환은 줄어듬이 아닙니다.

---

```
char c1{7}; // 7은 int이지만 char에 포함되므로 가능하다
char c2{77777}; // 오류 : 줄어듬
```

---

참고로 부동 소수점 - 정수 간 변환은 언제나 줄어듬으로 판단됩니다. 심지어 7.0에서 7로 변환하는 것도 말이지요.

#### 참고 자료

- the C++ draft section 8.5.4.
- [N18907 = 05 – 0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N2215 = 07 – 0075] Bjarne Stroustrup and Gabriel Dos Reis : Initializer lists (Rev. 3) .
- [N2640 = 08 – 0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists - Alternative Mechanism and Rationale (v. 2) (primarily on "explicit").

## 2.30 대표 생성자(Delegating constructors)

C++ 98에서 만일 두 생성자가 같은 일을 하도록 원했다면, 여러분은 어떤 “init()” 함수를 만들어서 이 함수를 호출하도록 하였을 것입니다. 예를 들어서

---

```
class X {
    int a;
    validate(int x) { if (0 < x && x <= max) a=x; else throw bad_X(x); }
public:
    X(int x) { validate(x); }
    X() { validate(42); }
```

---

<sup>5</sup>narrow의 원래 뜻은 좁아짐 이란 뜻이지만, 한글로 번역할 때 줄어듬이 좀 더 어감이 좋아 이를 사용하게 되었다. 원래 C++에서 narrow은 wide character (2 바이트 문자)를 그냥 character로 변환하는 것을 narrow라 불렀는데 (즉 wide를 줄이는 것이었으니까) 이를 더 넓은 의미로 사용한다.

---

```
X(string s) { int x = lexical_cast<int>(s); validate(x); }
// ...
};
```

---

이와 같이 한다면, 불필요하게 코드를 더 쓰게 되므로 오류가 잘 발생하며, 유지 보수에도 불편합니다. 따라서, C++ 11 에서는 우리는 한 개의 생성자를 정의하여 다른 생성자에 사용할 수 있게 하였습니다.

---

```
class X {
    int a;
public:
    X(int x) { if (0 < x && x <= max) a = x; else throw bad_X(x); }
    X() :X{42} { }
    X(string s) :X{lexical_cast<int>(s)} { }
    // ...
};
```

---

### 참고 자료

- the C++ draft section 12.6.2
- [N1986 == 06 – 0056] Herb Sutter and Francis Glassborow: [Delegating Constructors \(revision 3\)](#).

## 2.31 클래스 내부 멤버 초기화자

C++ 98 에서는 오직 정수 타입의 정적 상수 멤버들만이 클래스 내부에서 초기화 될 수 있었고, 초기화자는 반드시 상수 식이여야만 했습니다. 이러한, 제약 조건은 컴파일 타입에 초기화가 될 수 있게 보장을 하였지요. 예를 들어서

---

```
int var = 7;

class X {
    static const int m1 = 7; // 가능
    const int m2 = 7; // 오류 : static 아님
    static int m3 = 7; // 오류 : const 아님
    static const int m4 = var; // 오류 : 초기화자가 상수 식이 아니다
    static const string m5 = "odd"; // 오류 : 정수 타입이 아닙니다
    // ...
};
```

---

C++ 11 에서는, 기본적으로 비 정적 데이터 멤버들도 클래스 안에 선언된 부분에서 바로 초기화 할 수 있게 하였습니다. 그 다음에 만일 런타임 초기화가 필요할 때 생성자 초기화자를 사용할 수 있게 말이지요. 예를 들어:

---

```
class A {
public:
    int a = 7;
};
```

---

는 아래 코드와 동일합니다.

---

```
class A {
public:
    int a;
    A() : a(7) {}
};
```

---

이는 많은 타이핑을 줄여줍니다. 뿐만 아니라, 여러개의 생성자를 가지고 있는 클래스들을 사용할 때에도 도움이 되지요. 많은 경우 모든 생성자가 특정 멤버에 대해 공통적인 초기화자를 가지는 경우가 있습니다.

---

```
class A {
public:
    A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor run") {}
    A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor run") {}
    A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor run") {}
    int a, b;
private:
    HashingFunction hash_algorithm; // 모든 A 인스턴스들에 대해 암호화 된 해시가 적용된다
    std::string s; // 객체 생존사이클에 대한 상태를 나타내기 위한 문자열
};
```

---

hash\_algorithm 과 s 모두 하나의 디폴트 값이 있다는 사실은 위의 혼잡한 코드에서 쉽게 눈치채기 어렵고, 유지 보수시에 문제가 될 것입니다. 그 대신, 우리는 데이터 멤버들의 초기화를 다음과 같이 수행할 수 있습니다.

---

```
class A {
public:
    A(): a(7), b(5) {}
    A(int a_val) : a(a_val), b(5) {}
    A(D d) : a(7), b(g(d)) {}
    int a, b;
private:
    HashingFunction hash_algorithm{"MD5"}; // 모든 A 인스턴스들에 대해 암호화 된 해시가 적용된다
    std::string s{"Constructor run"}; // 객체 생존사이클에 대한 상태를 나타내기 위한 문자열
};
```

---

만일 멤버가 클래스 내부와 생성자 모두에 의해 초기화 된다면, 생성자에 의한 초기화만 수행됩니다. (즉, 디폴트를 오버라이드 합니다). 따라서 우리는 아래와 같이 더 간단하게 나타낼 수 있습니다.

---

```

class A {
public:
    A() {}
    A(int a_val) : a(a_val) {}
    A(D d) : b(g(d)) {}
    int a = 7;
    int b = 5;
private:
    HashingFunction hash_algorithm{"MD5"}; // 모든 A 인스턴스들에 대해 암호화 된 해시가 적용된다
    std::string s{"Constructor run"}; // 객체 생존사이클에 대한 상태를 나타내기 위한 문자열
};

```

---

### 참고 자료

- the C++ draft section "one or two words all over the place"; see proposal.
- [N2628 = 08 – 0138] Michael Spertus and Bill Seymour: [Non-static data member initializers](#).

## 2.32 상속된 생성자

사람들은 종종 클래스 멤버들에도 보통의 범위 규칙(scope rule) 이 적용된다는 사실을 혼동합니다. 특히, 부모 클래스(base class) 의 멤버들이 자식 클래스(derived class) 의 멤버들과 다른 범위(scope) 안에 있을 때 그렇지요.

---

```

struct B {
    void f(double);
};

struct D : B {
    void f(int);
};

B b; b.f(4.5); // 가능
D d; d.f(4.5); // 놀랍게도 f(int) 의 인자로 4 가 전달되며 호출

```

---

C++ 98 에서는, 우리는 부모 클래스로 부터 자식 클래스로 오버로딩 된 함수들을 옮길 수 있었습니다.

```

struct B {
    void f(double);
};

struct D : B {
    using B::f; // 모든 f() 들을 B 의 범위로 가져온다
};

```

```

void f(int); // 새로운 f() 를 추가
};

B b; b.f(4.5); // 가능
D d; d.f(4.5); // 가능 : D::f(double) (이는 실제로 B::f(double)) 을 호출한다.

```

---

이제 이러한 기능은 보통의 멤버 함수 뿐만이 아니라 생성자들에게도 사용할 수 있게 됩니다. C++ 11에서는:

```

class Derived : public Base {
public:
using Base::f; // 부모의 f 를 자식의 범위로 가져온다 – C++ 98 에서 작동함
void f(char); // 새로운 f 를 제공한다
void f(int); // 이 f 를 Base::f(int) 보다 선호한다

using Base::Base; // Base 의 생성자를 자식의 범위로 가져온다 – C++ 11 에서만 가능
Derived(char); // 새로운 생성자를 제공한다
Derived(int); // 이 생성자를 Base::Base(int) 보다 선호한다.
// ...
};

```

---

한편, 자식클래스에서 부모 클래서의 생성자를 상속할 때, 자식 클래스에서 새로운 멤버 변수를 초기화해야 한다면, 아래와 같은 실수를 할 수도 있습니다.

```

struct B1 {
    B1(int) { }
};

struct D1 : B1 {
    using B1::B1; // 암시적으로 D1(int) 를 선언한다
    int x;
};

void test()
{
    D1 d(6); // d.x 가 초기화되지 않았습니다!
    D1 e; // 오류 : D1 은 디폴트 생성자가 없습니다
}

```

---

이와 같은 문제는 멤버 초기화자(member initializer) 을 도입함으로써 해결할 수 있습니다.

```

struct D1 : B1 {
    using B1::B1; // 암시적으로 D1(int) 를 선언한다
    int x{0}; // x 는 초기화되었습니다
}

```

```

};

void test()
{
    D1 d(6); // d.x 는 0 입니다
}

```

---

## 참고 자료

- the C++ draft section 12.9.
- [N1890 = 05 – 0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N1898 = 05 – 0158] Michel Michaud and Michael Wong: Forwarding and inherited constructors .
- [N2512 = 08 – 0022] Alisdair Meredith, Michael Wong, Jens Maurer: Inheriting Constructors (revision 4).

## 2.33 정적(컴파일 타임) assertion – static\_assert

컴파일 타임 assertion 은 상수식과 문자열 리터럴로 구성되어 있습니다.

---

```
static_assert(expression,string);
```

---

컴파일러는 식을 계산한 다음에, expression 이 거짓이라면 string 에 오류 메세지를 적습니다. (예를 들어 assertion 실패시에) 예를 들어서:

```

static_assert(sizeof(long)>=8, "64-bit code generation required for this library.");
struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof(Y),"unexpected padding in S");

```

---

static\_assert 는 프로그램의 어떠한 것을 것을 기대하고(assumption) 이를 처리하는 것을 컴파일러 명시적으로 할 때 유용합니다. 참고로 static\_assert 는 컴파일 시에 계산되고, 런타임에 의존하는 값들을 확인할 때에는 사용될 수 없음을 주의하셔야 합니다. 예를 들어

```

int f(int* p, int n)
{
    static_assert(p==0,"p is not null"); // 오류 : static_assert() 식이 상수 식이 아닙니다
    // ...
}

```

---

(그 대신에 실패시 예외를 던지는 것으로 해결할 수 있습니다)

## 참고 자료

- the C++ draft 7 [4].

- [N1381 = 02 – 0039] Robert Klarer and John Maddock: [Proposal to Add Static Assertions to the Core Language](#).
- [N1720 = 04 – 0160] Robert Klarer, John Maddock, Beman Dawes, Howard Hinnant: [Proposal to Add Static Assertions to the Core Language \(Revision 3\)](#).

## 2.34 long long – 더 길어진 정수형

최소 64 비트의 길이를 가지는 정수형. 예를 들어

---

```
long long x = 9223372036854775807LL;
```

---

참고로 long long long이나 long 을 short long long 으로 쓸 수 없습니다.

참고 자료

- the C++ draft ???.
- [05 – 0071 = N1811] J. Stephen Adamczyk: [Adding the long long type to C++ \(Revision 3\)](#).

## 2.35 nullptr – 널 포인터 리터럴

nullptr 은 널 포인터를 나타내는 리터럴(literal)입니다. 이는 정수가 아닙니다.

---

```
char* p = nullptr;
int* q = nullptr;
char* p2 = 0; // 0 을 넣어도 잘 작동하고, p == p2 가 된다
```

```
void f(int);
void f(char*);
```

```
f(0); // f(int) 호출
f(nullptr); // f(char*) 호출
```

```
void g(int);
g(nullptr); // 오류 : nullptr 는 int 가 아닙니다
int i = nullptr; // 오류 : nullptr 은 int 가 아닙니다
```

---

참고 자료

- the C++ draft section ???
- [N1488 = /03 – 0071] Herb Sutter and Bjarne Stroustrup: [A name for the null pointer: nullptr](#).
- [N2214 = 07 – 0074] Herb Sutter and Bjarne Stroustrup: [A name for the null pointer: nullptr \(revision 4\)](#).

## 2.36 후위 리턴 타입 문법

아래와 같은 코드를 살펴봅시다.

---

```
template<class T, class U>
??? mul(T x, U y)
{
    return x*y;
}
```

---

우리는 위 함수의 리턴 타입을 뭐라고 써야 될까요? 물론, “ $x * y$ ”의 타입이겠지요. 하지만 이를 어떻게 쓸까요? 첫 번째 아이디어는 decltype를 사용하는 것입니다.

---

```
template<class T, class U>
decltype(x*y) mul(T x, U y) // 범위 문제 !
{
    return x*y;
}
```

---

하지만 이는  $x$  와  $y$  가 범위 안에 있지 않기 때문에 오류가 발생합니다. 하지만 우리는

---

```
template<class T, class U>
decltype(*(T*)(0)**(U*)(0)) mul(T x, U y) // 보기 좋지 않고, 오류 발생 가능성 높음
{
    return x*y;
}
```

---

와 같이 쓸수 있겠지요. 하지만 위는 상당히 보기 좋지 않습니다. 위 해결책으로 리턴 타입을 인자 뒤에 서주면 됩니다.

---

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y)
{
    return x*y;
}
```

---

여기서 auto 의미는 “리턴 타입은 후에 유추될 것이다”라는 것을 내포하고 있습니다. 후위 문법은 단순히 템플릿과 타입 유추에 관련한 것만은 아닙니다. 이는 실제로 범위에 관련한 내용입니다.

---

```
struct List {
    struct Link { /* ... */ };
    Link* erase(Link* p); // p 를 제거하고, p 이전의 링크를 리턴
    // ...
};
```

---

```
List::Link* List::erase(Link* p) { /* ... */ }
```

---

먼저 첫번째 `List::` 는 `List` 의 범위가 두 번째 `List::` 를 만나기 전까지 범위 안에 포함되지 않으므로 필수적으로 써야 합니다. 더 나은 버전은

---

```
auto List::erase(Link* p) -> Link* { /* ... */ }
```

---

### 참고 자료

- the C++ draft section ???
- [Str02] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002.
- [N1478 = 03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto.
- [N2445 = 07 - 0315] Jason Merrill: New Function Declarator Syntax Wording.
- [N2825 = 09 - 0015] Lawrence Crowl and Alisdair Meredith: Unified Function Syntax.

## 2.37 템플릿 별명(alias) - 이전에는 “template typedef” 라 알려짐

어떻게 우리는 그냥 다른 평범한 템플릿 같이 보이지만, 일부 템플릿 인자들이 정해진(specified) 템플릿을 어떻게 만들까요? 아래의 코드를 살펴봅시다.

---

```
template<class T>
using Vec = std::vector<T,My_alloc<T>>; // 사용자의 할당자를 사용하는 표준 vector
```

```
Vec<int> fib = { 1, 2, 3, 5, 8, 13 }; // My_alloc 을 이용해서 할당한다
```

---

```
vector<int,My_alloc<int>> verbose = fib; // verbose 와 fib 은 같은 타입이다.
```

`using` 키워드는 “이 이름이 다음에 오는 것들을 의미한다” 라는 뜻입니다. 우리는 이를 고전적인 `typedef` 키워드를 사용해서 해결하려고 했으나, 문법적으로 모호하고 일관적인 해결책을 찾을 수 없었습니다.

물론 템플릿 특수화 역시 작동합니다 예를 들어

---

```
template<int>
struct int_exact_traits { // 아이디어 : int_exact_trait<N>::type 은 정확히 N 개의 비트를 가지는 타입
    typedef int type;
};

template<>
struct int_exact_traits<8> {
    typedef char type;
};
```

```

template<>
struct int_exact_traits<16> {
    typedef char[2] type;
};

// ...

template<int N>
using int_exact = typename int_exact_traits<N>::type; // 편리한 방법으로 별명을 정의한다

int_exact<8> a = 7; // int_exact<8> 은 8 비트의 int 이다

```

---

템플릿과의 중요한 연동성 말고도, 타입 별명은 다른 평범한 타입에 별명을 붙여 주는 것으로도 사용될 수 있습니다.

---

```

typedef void (*PFD)(double); // C 스타일
using PF = void (*)(double); // using 과 C 스타일 혼용
using P = [](double) -> void; // using 과 후위 리턴 타입

```

---

### 참고 자료

- the C++ draft: 14.6.7 Template aliases; 7.1.3 The **typedef** specifier
- [N1489 = 03 – 0072] Bjarne Stroustrup and Gabriel Dos Reis: [Templates aliases for C++.](#)
- [N2258 = 07 – 0118] Gabriel Dos Reis and Bjarne Stroustrup: [Templates Aliases \(Revision 3\) \(final proposal\).](#)

## 2.38 가변인자 템플릿(variadic template)

다음과 같이 해결해야 할 문제들이 있습니다 :

- 1, 2, 3, 4, 5, 6, 7, 8, 9, .. 개의 초기화자를 가지는 클래스를 어떻게 생성할까요?
- tuple 을 어떻게 생성할까요?

그 마지막 질문이 핵심입니다. tuple 을 생각해보세요! 아래는 그 예제로, 범용적이고 타입 안전한 printf() 를 만드는 예시입니다. 사실 boost::format 을 사용했다면 더 나았겠지만 아래와 같은 코드를 봅시다.

---

```

const string pi = "pi";
const char* m = "The value of %s is about %g (unless you live in %s).\n";
printf(m, pi, 3.14159, "Indiana");

```

---

printf() 의 가장 간단한 케이스로, 형식 문자열(format string) 외에 인자가 없는 경우입니다. 따라서 우리는 첫번째 경우 부터 처리할 것입니다.

---

```

void printf(const char* s)
{
    while (s && *s) {
        if (*s=='%' && *++s!=='%') // 더 이상 인자를 가지지 않도록 확실히함
            throw runtime_error("invalid format: missing arguments");
        std::cout << *s++;
    }
}

```

---

이제, 우리는 더 많은 인자를 가지는 printf() 를 처리해야 합니다.

---

```

template<typename T, typename... Args> // note the "..."
void printf(const char* s, T value, Args... args) // note the "..."
{
    while (s && *s) {
        if (*s=='%' && *++s!=='%') { // 형식 지정자6 (사실 이를 무시한다)
            std::cout << value; // 첫번째 비-형식 인자를 사용
            return printf(++s, args...); // 첫번째 인자를 벗겨낸다
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}

```

---

위 코드는 간단히 말해서, 맨 처음 비-형식 인자를 ‘벗겨 내면서(peel off)’ 재귀적으로 호출 됩니다. 만일 더이상 비 형식 인자들이 없게 된다면, 이 함수는 맨 처음에 printf() 을 호출하게 되지요. 이는 마치 컴파일 타임에 표준 함수 프로그래밍(functional programming) 을 수행하는 것과 같습니다. 참고로, << 를 오버로딩 하는 것이 형식 지정자(format specifier) 에서 발생할 수 있었던 오류를 없애줄을 알 수 있습니다. Args.. 는 “인자 모음(parameter pack)” 이라고 부르는 것입니다. 이는 간단히 (타입(type)/값(value)) 쌍으로 구성되며, 여러분은 각각의 인자를 맨 처음부터 벗겨낼 수 있습니다. 만일 printf() 함수가 1 개의 인자를 가질 때 호출되었다면, printf() 의 첫번째 정의인 printf(const char\*) 가 호출됩니다. 만일 printf() 가 두 개 이상의 인자를 가지는 형태로 호출되었다면, 두 번째 정의인 printf(const char\*, T value, Args... args) 가 선택되며, 첫번째 인자는 s, 두번째는 value, 나머지는 인자 모음에 args 로 나중에 사용될 형태로 들어가게 됩니다. 아래

---

```
printf(++s, args...);
```

---

를 호출 시에, 인자 모음 args 는 확장되어서, 다음 인자가 value 로 선택될 수 있도록 합니다. 이와 같은 과정은 args 가 빌 때 까지 수행됩니다 (즉 printf() 가 호출될 때 까지) 여러분이 함수 프로그래밍에 익숙하다면, 여러분은 이런 식으로 사용하는 것이 꽤나 표준적인 테크닉이라는 사실을 알 것입니다. 그렇지

---

<sup>6</sup>형식 지정자란, 이전에 printf 문에서 %d, %f 등으로 어떤 타입이 오고 어떤 방식으로 출력하는지 나타냈던것들

않다면, 아래 약간의 도움이 될만한 기술적인 예제를 써놓았습니다. 먼저, 우리는 간단한 가변인자 템플릿 함수를 선언할 것입니다. (위 `printf()` 처럼)

---

```
template<class ... Types>
void f(Types ... args); // 가변인자 템플릿 함수
                           // (임의의 개수의 임의의 타입의 인자를 가지는 함수
f(); // 가능 : args 는 인자를 가지고 있지 않다
f(1); // 가능 : args 는 1 개의 인자 int 를 가진다
f(2, 1.0); // 가능 : args 는 2 개의 인자 int 와 double 을 가지고 있다
```

---

우리는 가변인자 타입을 아래와 같이 구성할 수 있습니다.

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
    : private tuple<Tail...> { // 재귀를 사용한다.
        // 기본적으로 tuple 은 head 를 첫번째 (타입/값)쌍에 저장을 한다음
        // head 뒤에 오는 (타입/값) 쌍들로 부터 상속을 받는다.
        // 참고로 타입은 타입안에 인코드 되어 있으며, 데이터로 저장되지 않는다.
typedef tuple<Tail...> inherited;
public:
tuple() { } // 디폴트 : 비어있는 tuple

// 다른 인자를 가지는 tuple 을 생성한다.
tuple(typename add_const_reference<Head>::type v, typename add_const_reference<Tail>::type... vtail)
    : m_head(v), inherited(vtail...) { }

// 다른 tuple 로 부터 tuple 을 생성한다.
template<typename... VValues>
tuple(const tuple<VValues...>& other)
    : m_head(other.head()), inherited(other.tail()) { }

template<typename... VValues>
tuple& operator=(const tuple<VValues...>& other) // 대입
{
    m_head = other.head();
    tail() = other.tail();
    return *this;
}

typename add_reference<Head>::type head() { return m_head; }
typename add_reference<const Head>::type head() const { return m_head; }

inherited& tail() { return *this; }
const inherited& tail() const { return *this; }
```

**protected:**

```
    Head m_head;
}
```

---

정의에 따라 우리는 tuple 을 만들 수 있게 됩니다 (그리고 복사와 수정도 가능)

```
tuple<string,vector,double> tt("hello",{1,2,3,4},1.2);
string h = tt.head(); // "hello"
tuple<vector<int>,double> t2 = tt.tail(); // {{1,2,3,4},1.2};
```

---

모든 타입들에 대해 언급하는 것은 꽤나 지루하니, 많은 경우 우리는 인자 타입으로 부터 유추할 수 있습니다. 예를 들어 표준 라이브러리 make\_tuple() :

```
template<class... Types>
tuple<Types...> make_tuple(Types&&... t) // 이 정의는 실제보다 단순하게 표현한것 (표준 20.5.2.2 참조)
{
    return tuple<Types...>(t...);
}

string s = "Hello";
vector<int> v = {1,2,3,4,5};
auto x = make_tuple(s,v,1.2);
```

---

## 참고 자료

- Standard 14.6.3 Variadic templates
- [N2151 = 07 – 0011] D. Gregor, J. Jarvi: [Variadic Templates for the C++0x Standard Library](#).
- [N2080 = 06 – 0150] D. Gregor, J. Jarvi, G. Powell: [Variadic Templates \(Revision 3\)](#).
- [N2087 = 06 – 0157] Douglas Gregor: [A Brief Introduction to Variadic Templates](#).
- [N2772 = 08 – 0282] L. Joly, R. Klarer: [Variadic functions: Variadic templates or initializer lists?](#)  
– Revision 1.
- [N2551 = 08 – 0061] Sylvain Pion: [A Variadic std::min\(T, ...\) for the C++ Standard Library \(Revision 2\)](#) .
- Anthony Williams: [An Introduction to Variadic Templates in C++0x](#). DevX.com, May 2009.

## 2.39 단일화된 초기화 문법

C++ 에는 객체를 초기화 할 때 타입과 어떠한 방식으로 초기화 하는지에 따라 여러가지 방법을 제공합니다. 하지만 이를 잘못 사용하게 될 때 발생하는 오류 메세지는 의아할 수 도 있고 모호 한 경우가 많습니다. 예를 들어

---

```
string a[] = { "foo", "bar" }; // 가능 : 배열 변수를 초기화 한다.
vector<string> v = { "foo", "bar" }; // 오류 : non-aggregate vector에 대한 초기화자 리스트
void f(string a[]);
f( { "foo", "bar" } ); // 문법 오류
```

---

**와**


---

```
int a = 2; // 한 가지 대입 형태
int aa[] = { 2, 3 }; // 리스트를 사용한 한 가지 대입 형태
complex z(1,2); // 함수 형태의 초기화
x = Ptr(y); // 함수 형태의 변환/캐스트/생성
```

---

**와**


---

```
int a(1); // 변수 정의
int b(); // 함수 선언
int b(foo); // 변수 정의 혹은 함수 선언
```

---

초기화 규칙을 외우고 가장 좋은 것을 택하는 일은 매우 어려운 것입니다. C++ 11에서는 모든 초기화 방식으로 {}-초기화자를 사용하도록 하였습니다.

---

```
X x1 = X{1,2};
X x2 = {1,2}; // =는 안써도 된다
X x3{1,2};
X* p = new X{1,2};

struct D : X {
    D(int x, int y) :X{x,y} { /* ... */ };
}
```

```
struct S {
    int a[3];
    S(int x, int y, int z) :a{x,y,z} { /* ... */ }; // 이전 문제 해결
};
```

---

중요한 점은 X{a}는 모든 코드에서 동일한 값으로 초기화가 수행되므로, {}-초기화는 이에 사용 가능한 모든 곳에서 올바른 결과를냅니다. 예를 들어서

---

```
X x{a};
X* p = new X{a};
z = X{a}; // 캐스팅으로 사용
f({a}); // 타입 X의 함수 인자
return {a}; // 함수 리턴 값으로 사용 (X를 리턴하는 함수)
```

---

참고 자료

- the C++ draft section ???
- [N2215 = 07 – 0075] Bjarne Stroustrup and Gabriel Dos Reis: [Initializer lists \(Rev. 3\)](#) .
- [N2640 = 08 – 0150] Jason Merrill and Daveed Vandevoorde: [Initializer Lists – Alternative Mechanism and Rationale \(v. 2\) \(final proposal\)](#).

## 2.40 우측값 참조

좌측값(대입 연산에서 좌변에 올 수 있는 값들)과 우측값(대입 연산에서 우측에 올 수 있는 값)들의 차이를 구별하는 것은 Christopher Strachey (C++ 의 면 조상인 CPL 의 창시자)로 거슬러 올라갑니다. C++에서 비상수 레퍼런스는 좌측값에 결합(bind) 할 수 있고, 상수 레퍼런스는 좌/우측값 모두에 결합 할 수 있지만, 비 상수 레퍼런스가 우측값에 결합 할 수는 없었습니다. 이는 사람들로 하여금 임시 메모리에 들어있는 값을 바꾸는 것을 막기 위함이였습니다. 예를 들어서

---

```
void incr(int& a) { ++a; }
int i = 0;
incr(i); // i 는 1 이 된다
incr(0); // 오류 : 0 은 좌측값이 아닙니다.
```

---

만일 `incr(0)` 이 허용된다면, 어떤 사람도 볼 수 없는 임시 메모리가 증가되거나, 더 심각한 경우 0의 값이 1이 될 것입니다. 사실 후자의 경우 말이 안된다고 생각하지만 사실 이는 초기 포트란(Fortran) 컴파일러에서 있었던 버그 종류 중 하나입니다.

하지만 아래와 같은 코드를 생각해봅시다.

---

```
template<class T> swap(T& a, T& b) // 옛날 스타일의 swap
{
    T tmp(a); // a 의 복사본 2 개가 존재한다.
    a = b; // b 의 복사본 2개가 존재한다.
    b = tmp; // tmp 의 복사본 2 개가 존재한다.
}
```

---

만일 T가 원소를 복사하기에 매우 비용이 비싼 타입이라면 (예를 들어 `string`이나 `vector`), swap 작업은 매우 비싼 작업이 될 것입니다 (표준 라이브러리의 경우 우리는 `string`과 `vector`에 대해 이와 같은 swap 을 처리하는 특수화들이 있습니다). 하지만 여기서 한 가지 흥미로운 점이 있습니다. 우리는 어떠한 것도 복사하기를 원치 않습니다. 우리는 단순히 a 와 b 그리고 tmp 의 값을 단순히 이동 시키기만을 원합니다.

C++ 11에서는 우리는 “이동 생성자(move constructor)”와 “이동 대입 연산자”를 정의해서 복사가 아닌 이동을 수행할 수 있게 되었습니다.

---

```
template<class T> class vector {
    // ...
    vector(const vector&); // 복사 생성자
    vector(vector&&); // 이동 생성자
}
```

---

```

vector& operator=(const vector&); // 복사 대입 연산자
vector& operator=(vector&&); // 이동 대입 연산자
}; // 참고로 이동 생성자와 이동 대입 연산자는 non const && 를 인자로 가진다.
// 쉽게 말해 인자에 복사하는 것이 아닌 인자 그 자체가 되는 것이다.

```

---

&& 는 우측값 레퍼런스를 의미합니다. 우측값 레퍼런스는 우측값에 결합 할 수 있습니다 (하지만 좌측값에는 불가능) :

```

X a;
X f();
X& r1 = a; // r1 이 a 에 (좌측값) 결합한다
X& r2 = f(); // 오류 : f() 는 우측값이다. 결합할 수 없다.

X&& rr1 = f(); // 가능 : rr1 이 임시값에 결합한다.
X&& rr2 = a; // 오류 : 좌측값이 결합할 수 없습니다.

```

---

이동 대입 연산자를 만드는 아이디어는, 복사를 하기 보다는 이는 단순히 source에서 간단한 디폴트로 단순히 치환해버리는 것입니다. 예를 들어서 이동 대입 연산자를 사용하는 `s1 = s2` 는, `s2` 의 문자들을 `s1`에 복사되는 것이 아닙니다. 그 대신에, 단순히 `s1` 이 `s2` 의 문자들을 자신의 것처럼 가진 다음에, `s1`의 이전의 문자들을 지워버리게 되는 것입니다.

그렇다면 우리는 단순히 원본을 이동시키는 것이 가능한 것인지를 어떻게 알 수 있을까요? 간단히 컴파일러에게 말해주면 됩니다.

```

template<class T>
void swap(T& a, T& b) // 완벽한 swap
{
    T tmp = move(a); // a 를 못쓰게 된다
    a = move(b); // b 를 못쓰게 된다
    b = move(tmp); // tmp 를 못쓰게 된다
}

```

---

`move(x)` 는 “`x` 를 우측값 처럼 취급해라” 라는 의미입니다. 물론 `move()` 대신에 `rval()` 이런 이름을 붙였다면 더 알기 쉬웠겠지만 이미 오랫동안 `move` 라는 이름으로 사용되어왔습니다. `move()` 템플릿 함수는 C++ 11로 쓰여졌고, 우측값 참조를 사용합니다.

우측값 레퍼런스는 또한 완벽한 전달(perfect forwarding)을 제공하기 위해 이용될 수 있습니다.

C++ 11 표준 라이브러리에서 모든 컨테이너들에는 이동 생성자와 이동 대입 연산자들, 그리고 새로운 원소를 넣는 연산들 (`insert()`, `push_back()` 등) 은 모두 우측값 레퍼런스를 취하는 버전들이 있습니다. 결과적으로, 컨테이너들과 알고리즘의 성능이 유저 개입 없이 더 향상될 수 있었는데 이는 복사를 전보다 적게 수행하기 때문입니다.

## 참고 자료

- the C++ draft section ???
- N1385 N1690 N1770 N1855 N1952

- [N2027 = 06 – 0097] Howard Hinnant, Bjarne Stroustrup, and Bronek Kozicki: [A brief introduction to rvalue references](#)
- [N1377 = 02 – 0035] Howard E. Hinnant, Peter Dimov, and Dave Abrahams: [A Proposal to Add Move Semantics Support to the C++ Language \(original proposal\)](#).
- [N2118 = 06 – 0188] Howard Hinnant: [A Proposal to Add an Rvalue Reference to the C++ Language](#)
- Proposed Wording (Revision 3) (final proposal).

## 2.41 공용체 (일반화된)

C++ 98에서 (그리고 C++의 이전 버전들에서), 사용자 정의 생성자, 소멸자 혹은 대입 연산자를 가지고 있는 멤버들은 공용체의 멤버들이 될 수 없습니다.

---

```
union U {
    int m1;
    complex<double> m2; // 오류 : complex 는 생성자가 있습니다.
    string m3; // 오류 string 에는 생성, 복사, 소멸자에 의해 관리되는
               // 중요한 불변값이 있습니다
};
```

---

특히

---

```
U u; // 어떤 생성자가 호출되어야 한가요?
u.m1 = 1; // int 멤버에 값을 대입
string s = u.m3; // string 멤버로 부터 값을 읽음!! (큰 문제)
```

---

명백하게도, 위와 같이 한 멤버에 씀으로써 다른 멤버의 값까지 바뀌게 된다면 명백한 오류이겠지만 사람들은 그럼에도 불구하고 위와 같은 코드를 씁니다 (대부분 실수이겠지만)

C++ 11은 union들의 제한 조건을 바꾸어서, 적절한 멤버 타입들만 사용하도록 만들었습니다. 특히 멤버들의 타입이 생성자와 소멸자를 가질 수 있게 허용합니다. 또한 제약 조건을 추가해서 좀 더 유연해진 union이 좀더 오류가 적고 안전하도록 만들었습니다.

공용체의 멤버 타입들은 다음과 같이 제한됩니다.

- 가상 함수 안됨 (원래 안됨)
- 레퍼런스 안됨 (원래 안됨)
- 부모 클래스 안됨 (원래 안됨)
- 만일 공용체가 사용자 정의 생성자, 복사, 소멸자를 가지고 있는 멤버가 있다면, 이러한 함수들은 삭제됩니다. 다시 말해, 공용체 타입의 객체에서 이들을 사용할 수 없다는 점입니다. (새로 추가됨)

예를 들어

---

```
union U1 {
    int m1;
    complex<double> m2; // 가능
};

union U2 {
    int m1;
    string m3; // 가능
};
```

---

위 코드는 오류 발생 가능성성이 높아 보이지만, 새로운 제한 규칙이 이 때 도움을 줍니다. 특히

```
U1 u; // 가능
u.m2 = {1,2}; // 가능 : complex 멤버에 대입된다.
U2 u2; // 오류 : string 의 소멸자가 U 소멸자가 작동하지 못하게 함
U2 u3 = u2; // 오류 : string 복사 생성자가 U 의 복사 생성자를 작동하지 못하게 함
```

---

기본적으로 U2 는 어떤 멤버들이 사용되는지 추적할 수 있는 구조체에 넣지 않는 한 쓸모가 없습니다. 따라서, ‘구별 공용체(discriminate union)’를 제작해봅시다.

```
class Widget { // 3 개의 대안 구현들이 공용체로 표현된다.

private:

    enum class Tag { point, number, text } type; // 구별하기 위한 지시자

    union { // 표현 방식들
        point p; // point 는 생성자가 있다.

        int i;

        string s; // string 은 디폴트 생성자, 복사 연산들, 소멸자가 있다.
    };
    // ...
    widget& operator=(const widget& w) // string 때문에 필요로 함
    {
        if (type==Tag::text && w.type==Tag::text) {
            s = w.s; // 보통의 string 대입
            return *this;
        }

        if (type==Tag::text) s.^string(); // (명시적으로) 소멸

        switch (w.type) {
            case Tag::point: p = w.p; break; // 평범한 복사
            case Tag::number: i = w.i; break;
            case Tag::text: new(&s)(w.s); break; // new
        }
    }
}
```

---

```

    type = w.type;
    return *this;
}
};

```

---

참고 자료

- the C++ draft section 9.5
- [N2544 = 08 – 0054] Alan Talbot, Lois Goldthwaite, Lawrence Crowl, and Jens Maurer: **Unrestricted unions (Revision 2)**

## 2.42 PODs (일반화 된)

POD (“Plain Old Data” 의 약자) 는 C 구조체처럼 조작될 수 있는 것들을 의미합니다. 즉, `memcpy()` 를 통해 복사되고, `memset()` 을 통해 초기화 될 수 있습니다. C++ 98 에서 POD 의 실제 정의는, 구조체를 정의할 때 사용되었던 언어의 기능들을 바탕으로 이루어져 있습니다.

---

```

struct S { int a; };// S 는 POD 이다
struct SS { int a; SS(int aa) : a(aa) { } }; // SS 는 POD 가 아니다
struct SSS { virtual void f(); /* ... */ };

```

---

C++ 11 에서 S 와 SS 는 표준 레이아웃 타입(standard layout type, 다른 말로 POD) 라고 할 수 있습니다. 왜냐하면 SS 에는 어떠한 특이한 것들도 없기 때문입니다. 왜냐하면 생성자는 레이아웃에 영향을 주지 않으므로 (따라서 `memcpy()` 도 사용 가능) 하고, 오로지 문제가 되는 것은 초기화 (`memset()`) 을 사용할 수 없다 - 왜냐하면 불변을 보장하지 않기 때문) 뿐입니다. 하지만 SSS 는 가상 포인터(vptr)가 있기 때문에 POD 라고 할 수 없습니다. C++ 11 에서는 POD 를 자명하게 복사 가능한 타입들, 자명한 타입들, 그리고 POD 들의 기술적인 측면을 처리하기 위한 표준 레이아웃 타입들로 정의됩니다. 즉 POD 는 재귀적으로 정의되어 있습니다.

- 어떠한 것의 멤버들과 부모 클래스들이 모두 POD 라면, 그 클래스는 POD 입니다.
- 또한 자명하게도
  - 가상 함수를 가지면 안되고
  - 부모 클래스를 가져도 안되고
  - 레퍼런스를 가져도 안되고
  - 다중 접근 지시자를 가져도 안됩니다

C++ 11 의 가장 중요한 특징 중 하나로, 생성자를 넣거나 빼는 것은 전체 레이아웃이나 성능에 영향을 주지 않습니다.

참고 자료

- the C++ draft section 3.9 and 9 [10]
- [N2294 = 07 – 0154] Beman Dawes: **POD's Revisited; Resolving Core Issue 568 (Revision 4).**

### 2.43 Raw 문자열 리터럴

영어 단어 Raw 의 원래 의미는 날것, 가공되지 않은 것이라는 의미입니다. C++ 에서 사용되는 의미로 특수문자를 입력하기 위해서 특수문자 앞에 백슬래시를 붙이지 않았다는 뜻입니다. 많은 경우, 예를 들어서 여러분이 표준 정규 표현식(<regex>) 라이브러리를 사용할 때와 같이, \가 탈출 문자로 사용 되는 경우 상당히 골치아픕니다. (왜냐하면 정규 표현식에서 \는 특별한 문자를 사용할 때 쓰이기 때문입니다) 예를 들어서 두 개의 단어를 \로 분리하는 패턴을 어떻게 쓰냐면

```
string s = "\w\\\w"; // 제가 맞게 썼기를 바랍니다
```

참고로 정규 표현식에서 백슬래시 문자는 두 개의 백슬래시 문자로 표현됩니다. 기본적으로, “raw string literal”에서 백슬래시는 단순히 진짜 백슬래시 하나로 처리 됩니다. 따라서 우리의 코드는 아래와 같이 바꿀 수 있습니다.

```
string s = R"(\w\\w)"; // 맞게 썼을 것입니다.
```

사실 raw string의 필요성을 설명하기 위해 원래 아래의 예제 코드를 사용하였습니다.

```
"'(?:[^\\\\\\']|\\\\\\.)*'|\\"(?:[^\\\\\\\"]]|\\\\\\.)*\\")'" // 5개의 백슬래시를 사용하는게 맞는가요?  
// 심지어 전문가들도 헤각립니다.
```

R”(...)” 기호는 단순히 “문자열”로 사용 하는 것 보다 좀더 복잡하지만, 백슬래시 하나만 안쓰는 것 보다 더 쿠 장점이 있습니다. 여기서 따옴표를 어떻게 넣을까요? 쉽습니다.

`R"("quoted string")` //이 문자열은 “quoted string” (큰따옴표도 포함해서)

그렇다면 마지막으로 ”)” 를 어떻게 raw 문자열에 넣을까요? 다행이도 이는 거의 들들게 나타나는 문제이지만, “( ... )” 가 제한 쌍이기 때문에 (delimiter pair) 우리는 아래와 같은 방법을 도입하여야 합니다.

```
R"***("quoted string containing the usual terminator (")")***"  
// 이 문자열은 “quoted string containing the usual terminator (”)” 입니다.
```

) 뒤에 오는 문자 시퀀스는 반드시 ( 앞에 오는 문자 시퀀스와 동일해야 합니다. 이 방법으로, 우리는 임의의 복잡한 패턴 문제를 해결할 수 있게 되었습니다. raw 문자열 앞의 R 문자 앞에, 어떠한 문자열로 문자를 인코딩 할지 표시할 수 있습니다 (u8, u, U, L) 예를 들어서 u8R"(fdfdfa)" 는 UTF-8 문자열 리터럴입니다.

참고 자료

- Standard 2.13.4
  - [N2053 = 06 – 0123] Beman Dawes: **Raw string literals.** (original proposal)
  - [N2442 = 07 – 0312] Lawrence Crowl and Beman Dawes: **Raw and Unicode String Literals; Unified Proposal (Rev. 2).** (final proposal combined with the User-defined literals proposal).
  - [N3077 = 10 – 0067] Jason Merrill: **Alternative approach to Raw String issues.** (replacing [ with () ;

## 2.44 사용자 정의 리터럴

C++ 은 여러가지 내장 타입들에 대한 리터럴들을 제공하고 있습니다. :

---

```
123 // int
1.2 // double
1.2F // float
'a' // char
1ULL // unsigned long long
0xD0 // 16진수 unsigned
"as" // string
```

---

하지만 C++ 98 에는 사용자 정의 타입들에 대한 리터럴들을 만들 수 없습니다. 이는 내장 타입들처럼 사용자 정의 타입이 동일하게 지원되어야만 한다는 원칙에 위반되는 것이므로 상당히 골치아팠습니다. 특히, 사람들은 다음과 같은 것들을 요구하였습니다.

---

```
"Hi!"s // 'char 의 널 문자 종료 배열' 아니 아닌 문자열
1.2i // 허수
123.4567891234df // 정수 부동 소수점(IBM)
101010111000101b // 이진수
123s // 초
123.56km // 킬로미터
123456789012345678901234567890x // 확장 정밀도
```

---

C++ 11 은 사용자 정의 리터럴들을 사용할 수 있도록 지원합니다. 리터럴 연산자는 특정한 접미사를 가지고 있는 리터럴들을 원하는 타입으로 맵핑(mapping) 시켜 줍니다. 예를 들어서

---

```
constexpr complex<double> operator "" i(long double d) // 허수 리터럴
{
    return {0,d}; // complex 는 리터럴 타입이다
}

std::string operator "" s (const char* p, size_t n) // std::string 리터럴
{
    return string(p,n); // 자유 저장 공간을 필요로 한다.
}
```

---

참고로 `constexpr` 를 사용하게 된다면 컴파일 시간에 계산을 할 수 있게 됩니다. 이를 통해 우리는

---

```
template<class T> void f(const T&);
f("Hello"); // char* 에 포인터로 전달
f("Hello"s); // (5 문자)문자열 객체를 전달
f("Hello\n"s); // (6 문자) 문자열 객체를 전달

auto z = 2+1i; // complex(2,1)
```

---

이 아이디어를 구현하는 방법은, 리터럴 일 수 있는 것들을 파싱한 다음, 컴파일러가 항상 접미사를 확인하는 것입니다. 사용자 정의 리터럴 메커니즘은 단순히 사용자가 특별한 접미사를 추가할 수 있게 됩니다. 내장된 리터럴 접미사를 재정의하거나, 리터럴 문법을 추가하는 것은 불가능하지 않습니다. 리터럴 연산자는 접미사 앞에 오는 리터럴을 수정된 (cooked - 즉 접미사가 정의되지 않았을 때 이 리터럴이 가졌을 값) 혹은 수정되지 않은 (문자열 그대로) 형태로 가져올 수 있게 됩니다.

수정되지 않는 문자열을 얻기 위해서는 단순히 `const char*` 인자를 요구하면 됩니다.

---

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);
f(1234567890123456789012345678901234567890x);
```

---

여기서 C 형식의 문자열 “123456789012345678901234567890”이 연산자 `operator"" x()`로 전달됩니다. 여기서 우리는 숫자들을 문자열로 명시적으로 바꾸지 않았습니다.

사용자 정의 리터럴로 만들 수 있는 (즉 접미사를 붙일 수 있는) 리터럴로 4 가지 종류가 있습니다.

- 정수 리터럴 : 하나의 `unsigned long long`이나 `const char*`을 인자로 가지는 리터럴 연산자가 받는다.
- 부동 소수점 리터럴 : 하나의 `long double`이나 `const char*`을 인자로 가지는 리터럴 연산자가 받는다.
- 문자열 리터럴 : (`const char*`, `size_t`)를 인자로 가지는 리터럴 연산자가 받는다.
- 문자 리터럴 : 하나의 문자를 받는 리터럴 연산자가 받는다.

참고로 여러분은 `const char*`만 인자로 받고 `size`는 받지 않는 문자열 리터럴 연산자는 생성할 수 없습니다. 예를 들어서

---

```
string operator"" S(const char* p); // 경고 : 예상했던 대로 작동하지 않을 수 있습니다

"one two" S; // 오류 : 사용 가능한 리터럴 연산자가 없음
```

---

는 예상한 대로 작동하지 않습니다. 상식적으로 우리가 ‘다른 종류의 문자열’을 원한다면, 아마 우리는 언제나 문자열 내의 문자 개수를 필요로 할 것입니다. 접미사들은 짧은 경우가 많으므로 (예를 들어 문자열은 s, 헤수는 i, 미터는 m, 그리고 확장(extended) 되었음을 나타내기 위한 것은 x) 이 때문에 접미사 간에 충돌이 발생하는 경우가 많습니다. 따라서 충돌을 막기 위해 네임스페이스를 사용하도록 하세요.

---

```
namespace Numerics {
    // ...
    class Bignum { /* ... */ };
    namespace literals {
```

```

operator"" X(char const*);  

}  

}  
  

using namespace Numerics::literals;

```

---

### 참고 자료

- Standard 2.14.8 User-defined literals
- [N2378 = 07 – 0238] Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Mauer, Alisdair Meredith, Bjarne Stroustrup, David Vandevoorde: **User-defined Literals** (aka. Extensible Literals (revision 3)).

## 2.45 속성(Attributes)

속성(attribute)은 새로운 표준 문법으로, 부가적인 혹은 장치(vendor) 특이적인 정보들(`_attribute`, `_declspec`, `#pragma`)을 소스코드에 집어넣을 수 있는 방법입니다. C++ 11, 속성은 기존의 문법들과는 다르게 코드 어디에서든지 적용 가능하고, 언제나 다른 코드 보다 우선시 됩니다. 예를 들어서

```

void f [[ noreturn ]] () // f() 는 절대 리턴하지 않음  

{  

    throw "error"; // OK  

}

```

```

struct foo* f [[carries_dependency]] (int i); // 최적화기에 알려줌  

int* g(int* x, int* y [[carries_dependency]]);

```

위 코드에서 볼 수 있듯이 속성은 두 개의 대괄호 안에 들어가게 됩니다 ( `[[ ... ]]` ). `noreturn` 과 `carries_dependency` 는 표준에 정의되어 있는 두 개의 속성입니다. 사실 이러한 속성을 통해 언어의 사투리(dialects)<sup>7</sup> 들을 만들 염려가 처음에 있습니다. 따라서, 속성을 제대로 사용하는 방법으로 오직 프로그램 자체의 의미에 영향을 주지는 않지만 오류를 찾는데 도움을 주거나 (예를 들어 `[[noreturn]]`) 혹은 최적화기에 도움을 주는 (예를 들어 `[[carries_dependency]]`) 형태로만 사용하는 것을 권합니다.

속성을 사용하는 또 다른 이유로 OpenMP 을 더 잘 지원하도록 하기 위함이 있습니다. 예를 들어

```

for [[omp::parallel()]] (int i=0; i<v.size(); ++i) {  

    // ...
}

```

### 참고 자료

- Standard: 7.6.1 Attribute syntax and semantics, 7.6.3-4 `noreturn`, `carries_dependency` 8 Declarators, 9 Classes, 10 Derived classes, 12.3.2 Conversion functions

<sup>7</sup>언어 안에서 같은 언어끼리 의미가 조금씩 달라질 염려를 뜻함

- [N2418 = 07–027] Jens Maurer, Michael Wong: Towards support for attributes in C++ (Revision 3)

## 2.46 람다(Lambdas)

람다식은 함수 객체를 생성하는 메커니즘이라 할 수 있습니다. 람다의 주요 사용 예는 함수를 통해 수행되는 간단한 작업들을 만들어내기 위함이죠. 예를 들어

```
vector<int> v = {50, -10, 20, -30};

std::sort(v.begin(), v.end()); // 디폴트 정렬
// 이제 v 는 -30, -10, 20, 50 가 된다

// 절대값으로 정렬
std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a) < abs(b); });
// 이제 v 는 -10, 20, -30, 50 가 된다
```

인자 [](int a, int b) { return abs(a) < abs(b); } 는 “람다” (혹은 “람다 함수”, “람다식”) 이라 불리며, 두 개의 정수 인자 a 와 b 를 받아서 두 개의 절대값을 비교한 결과를 리턴하는 작업을 의미합니다. 람다식은 그 사용 범위 안에 있는 지역 변수들에 접근할 수 있습니다. 예를 들어

```
void f(vector<Record>& v)
{
    vector<int> indices(v.size());
    int count = 0;
    generate(indices.begin(), indices.end(), [&count](){ return count++; });

    // indices 를 Record 의 name 의 크기 순서대로 정렬한다
    std::sort(indices.begin(), indices.end(), [&](int a, int b) { return v[a].name < v[b].name; });
    // ...
}
```

어떤 사람들은 위 코드를 정말 멋지다고 생각하고 어떤 사람들은 위험할 정도로 모호한 코드라 생각합니다. 제 개인적인 생각으로는 둘 다 맞는 의견이라 봅니다. [&] 은 ‘캡쳐 리스트(capture list)’ 라고 부르며, 지역 변수들이 레퍼런스로 함수에 전달 될 수 있게 해줍니다. 우리는 오직 v 만을 캡쳐 하고 있었기 때문에, [&v] 로 나타내면 됩니다. 만일 우리가 v 를 값으로 전달하고 싶었다면, [=v] 로 쓰면 됩니다. 아무것도 캡쳐하지 않는다면 [], 변수 전부를 레퍼런스로 캡쳐하기 위해서는 [&], 그리고 값으로 캡쳐하려면 [=] 를 해주면 됩니다.

만일 람다식에서 처리하는 과정이 단순한 것이 아니거나 일반적인 것이 아니라면, 저는 함수 객체나 함수를 사용하도록 권합니다. 예를 들어서 위에 쓴 예제 코드는 아래와 같이 쓸 수 있습니다.

```
void f(vector<Record>& v)
{
```

```

vector<int> indices(v.size());
int count = 0;
generate(indices.begin(), indices.end(), [&](){ return ++count; });

struct Cmp_names {
    const vector<Record>& vr;
    Cmp_names(const vector<Record>& r) :vr(r) { }
    bool operator()(int a, int b) const { return vr[a].name < vr[b].name; }
};

// indices 를 Record 의 name 의 크기 순서대로 정렬한다
std::sort(indices.begin(), indices.end(), Cmp_names(v));
// ...
}

```

---

작은 함수들에 대해서는, 예컨대 위에서 Record 이름 필드를 비교하는 함수 등은 굳이 귀찮게 함수 객체로 만들 필요가 없습니다. 왜냐하면 생성되는 코드가 정확히 동일하기 때문이지요. C++ 98 에서, 함수 객체는 템플릿 인자로 사용되기 위해서 비-지역 객체여야 했습니다. 이제 C++ 11 에서는 지역 타입을 템플릿 인자로 받을 수 있는 덕분에 이와 같은 제약 조건이 사라졌습니다.

람다를 생성하기 위해 여러분은

- 캡쳐 리스트 : 사용할 변수들. 만일 어떠한 변수도 캡쳐하고 싶지 않다면 단순히 [] 로 시작하면 된다.
- (선택사항) 인자들과 그들의 타입들
- 함수 내부에서 할 작업들은 중괄호 안에 넣는다 (예를 들어 { **return** v[a].name < v[b].name; } )
- (선택사항) 리턴 타입은 새로운 접미 리턴 타입 문법(suffix return type syntax)을 사용한다. 하지만 통상적으로 우리는 리턴타입으로 부터 타입 유추를 하기도 한다. 만일 어떠한 값도 주지 않는다면 void 라고 가정한다.

### 참고 자료

- Standard 5.1.2 Lambda expressions
- [N1968 = 06 – 0038] Jeremiah Willcock, Jaakko Jarvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine: [Lambda expressions and closures for C++ \(original proposal with a different syntax\)](#)
- [N2550 = 08 – 0060] Jaakko Jarvi, John Freeman, and Lawrence Crowl: [Lambda Expressions and Closures: Wording for Monomorphic Lambdas \(Revision 4\) \(final proposal\)](#).
- [N2859 = 09 – 0049] Daveed Vandevoorde: [New wording for C++0x Lambdas](#).

## 2.47 템플릿 인자로 사용되는 지역 타입

C++ 98에서 지역 타입(local type)과 이름 없는 타입들은 템플릿 인자로 사용할 수 없었습니다. 이는 상당한 부담이 되었는데, 따라서 C++ 11에서는 이러한 제한 조건을 없앴습니다.

---

```
void f(vector<X>& v)
{
    struct Less {
        bool operator()(const X& a, const X& b) { return a.v < b.v; }
    };
    sort(v.begin(), v.end(), Less()); // C++98: 오류 : Less 는 지역 타입
                                    // C++11: 가능
}
```

---

C++ 11에서 우리는 또한 람다식을 통한 버전을 사용할 수 있습니다.

---

```
void f(vector<X>& v)
{
    sort(v.begin(), v.end(),
        [] (const X& a, const X& b) { return a.v < b.v; }); // C++11
}
```

---

사실 이름을 붙이는 것은 나중에 문서화(documentation) 시에나, 좋은 디자인을 만들 때 중요한 작업입니다. 또한 비-지역 (반드시 이름이 붙어야 함) 객체들은 다시 사용될 수도 있습니다. C++ 11은 이름 없는 타입들이 템플릿 인자로 사용될 수 있도록 허용합니다.

---

```
template<typename T> void foo(T const& t){}
enum X { x };
enum { y };

int main()
{
    foo(x); // C++98: 가능; C++11: 가능
    foo(y); // C++98: 오류; C++11: 가능
    enum Z { z };
    foo(z); // C++98: 오류; C++11: 가능
}
```

---

참고 자료

- Standard: Not yet: CWG issue 757
- [N2402 = 07 – 0262] Anthony Williams: [Names, Linkage, and Templates \(rev 2\)](#).
- [N2657] John Spicer: [Local and Unnamed Types as Template Arguments](#).

## 2.48 noexcept – 예외의 전파를 막는다

만일 어떤 함수가 예외를 던질 수도 없거나, 함수가 던지는 예외를 프로그램이 처리할 수 없다면, 이 함수는 noexcept로 선언되어야만 합니다. 예를 들어

```
extern "C" double sqrt(double) noexcept; // 절대로 예외를 던지지 않는다
```

```
vector<double> my_computation(const vector<double>& v) noexcept
// 메모리 문제를 처리할 준비가 되지 않았다.
{
    vector<double> res(v.size()); // 예외를 던질 수 있음
    for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
    return res;
}
```

만일 noexcept로 정의된 함수가 예외를 던진다면 (즉, 예외가 noexcept 함수로 부터 탈출하려고 한다면), 프로그램은 종료됩니다. (terminate() 함수를 호출) terminate() 함수의 호출은 객체가 잘 정의된 상태인지 아닌지에 의존하지 않습니다 (따라서 소멸자가 잘 호출되는지 보장되지 않고, stack 풀기가 잘 수행되는지도 보장되지 않는다. 뿐만 아니라 프로그램에 실제 문제가 없을 경우 다시 재시작 할 수 없다). 이와 같이 결정한 이유는 noexcept를 단순하고, 효율적인 메커니즘으로 설계하기 위함이었습니다. (오래된 throw() 메커니즘 보다는 훨씬 더 효율적이다) 함수들을 특정 조건 한에서만 noexcept가 되도록 만들 수 있습니다. 예를 들어서, 템플릿 인자가 noexcept 일 때만 알고리즘이 noexcept 가 되도록 아래와 같이 구성할 수 있습니다.

```
template<class T>
void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0))))
// f(v.at(0)) 가 예외를 던질 수 있어야지만 예외를 던진다.
{
    for(int i; i<v.size(); ++i)
        v.at(i) = f(v.at(i));
}
```

여기서 저는 연산자에 noexcept를 사용하였습니다 : noexcept(f(v.at(0))) 은 f(v.at(0)) 이 예외를 던질 수 없을 때 참입니다. 즉, f() 과 at() 이 noexcept 일 때 말이지요. noexcept() 연산자는 상수식으로, 이의 피연산자는 계산되지 않습니다.

noexcept 선언의 일반적인 형태는 noexcept(expression) 이고, 그냥 ‘noexcept’ 라고 스는 것은 단순히 noexcept(true) 의 축약형이라 보면 됩니다. 함수의 모든 선언은 반드시 noexcept 호환인 구성을 가지고 있습니다.

소멸자는 예외를 던지면 안됩니다. 따라서 생성된 소멸자는 클래스의 모든 멤버들이 noexcept 소멸자를 갖고 있다면, 암시적으로 noexcept가 되게 (소멸자 내부에서 작업을 하던지 간에) 선언이 됩니다.

noexcept 는 표준 라이브러리에서 광범위하고 체계적이게 사용되고 있으며, 이는 전체적인 성능 향상과 여러 조건들을 명확하게 하였습니다.

## 참고 자료

- Standard: 15.4 Exception specifications [except.spec].
- Standard: 5.3.7 noexcept operator [expr.unary.noexcept].
- [N3103 = 10 – 0093] D. Kohlbrenner, D. Svoboda, and A. Wesie: Security impact of noexcept. (Noexcept must terminate, as it does).
- [N3167 = 10 – 0157] David Svoboda: Delete operators default to noexcept.
- [N3204 = 10 – 0194] Jens Maurer: Deducing ”noexcept” for destructors
- [N3050 = 10 – 0040] D. Abrahams, R. Sharoni, and D. Gregor: Allowing Move Constructors to Throw (Rev. 1).

## 2.49 정렬(alignment)

종종, 특히 우리가 메모리를 직접 제어하는 코드를 짤 때, 메모리 할당 시 우리가 원하는 대로 메모리에 정렬<sup>8</sup>을 해야할 필요성이 있습니다. 예를 들어서

---

```
// 문자들의 배열로 double로 정렬하도록 하였다.
alignas(double) unsigned char c[1024];
alignas(16) char[100]; // 16 바이트 경계로 정렬하도록 함 boundary
```

---

또한 alignof 연산자는 인자로 받는 것의 정렬 형태를 리턴합니다. (인자는 반드시 타입이여야만 함)

---

```
constexpr int n = alignof(int); // ints are aligned on n byte boundaries
```

---

## 참고 자료

- Standard: 5.3.6 Alignof [expr.alignof]
- Standard: 7.6.2 Alignment specifier [dcl.align]
- [N3093 = 10 – 0083] Lawrence Crowl: C and C++ Alignment Compatibility. Aligning the proposal to C’s later proposal.
- [N1877 = 05 – 0137] Attila (Farkas) Fehr: Adding Alignment Support to the C++ Programming Language. The original proposal.

---

<sup>8</sup>CPU 는 더블 워드 혹은 워드의 배수로 데이터들이 딱딱 떨어져야지만 가장 효율적으로 작동할 수 있기 때문에, 크기가 이에 배수가 되지 않는 자료형에 대해서 컴파일러가 일부로 빈 공간을 삽입해서 이에 맞춰 주는 경우가 있습니다. 이를 ‘정렬’이라고 합니다. 실제로 크기가 24비트 구조체의 경우 그 크기를 재보면 32비트가 나오는 경우가 있습니다.

## 2.50 오버라이드 제어 : override

자식 클래스에서 부모 클래스의 함수를 오버라이드 하기 위해서는 특별한 키워드가 필요로 하지 않았습니다. 예를 들어

---

```
struct B {
    virtual void f();
    virtual void g() const;
    virtual void h(char);
    void k(); // 가상이 아님
};

struct D : B {
    void f(); // B::f() 를 오버라이드 한다
    void g(); // B::g() 를 오버라이드 하지 않는다. (다른 타입)
    virtual void h(char); // B::h() 를 오버라이드 한다
    void k(); // B::k() 를 오버라이드 하지 않는다. (B::k() 는 가상함수 아님)
};
```

---

하지만 위와 같이 그냥 사용하는 것은 혼동을 불러오기 마련입니다. (여기서 프로그래머가 의도한 것은 무엇일까? 왜 컴파일러는 이 의심되는 코드에서 경고를 내지 않지?) 예를 들어서

- 프로그래머가 B::g() 를 오버라이드 하려 한 것일까? (아마 맞을 것입니다)
- 프로그래머가 B::h(**char**) 을 오버라이드 하려 한 것일까? (아마 explicit virtual 을 쓴 것을 보아 아닐 것입니다)
- 프로그래머가 B::k() 를 오버라이드 하려 한 것일까? (아마요, 하지만 이는 불가능 합니다)

프로그래머들로 좀더 명시적으로 오버라이드를 할 수 있게 하려면, 우리는 override 라는 키워드를 도입하였습니다. (강제하는 것이 아닌 그냥 보여주기 위함)

---

```
struct D : B {
    void f() override; // 좋음: B::f() 를 오버라이드 함
    void g() override; // 오류 : 다른 타입
    virtual void h(char); // B::h() 를 오버라이드 하겠지만 경고가 발생함
    void k() override; // 오류: B::k() 는 가상함수가 아님
};
```

---

override 로 선언된 것은 함수가 오버라이드 가능할 때만 오직 유효합니다. 프로그래머가 h() 가 오버라이드 되는지 눈치 채지 못할 수 있지만(언어적 측면에서 오류는 아니기 때문) 이는 경고를 내기 때문에 쉽게 알아낼 수 있습니다.

`override` 는 맥락적 키워드(contextual keyword)<sup>9</sup> 이기 때문에 아래와 같은 코드도 오류를 내지는 않습니다.

---

```
int override = 7; //권장하지 않음
```

---

### 참고 자료

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]
- [N3234 = 11 – 0004] Ville Voutilainen: Remove explicit from class-head.
- [N3151 == 10 – 0141] Ville Voutilainen: Keywords for override control. Earlier, more elaborate design.
- [N3163 == 10 – 0153] Herb Sutter: Override Control Using Contextual Keywords. Alternative earlier more elaborate design.
- [N2852 == 09 – 0042] V. Voutilainen, A. Meredith, J. Maurer, and C. Uzdavinis: Explicit Virtual Overrides. Earlier design based on attributes.
- [N1827 == 05 – 0087] C. Uzdavinis and A. Meredith: An Explicit Override Syntax for C++. The original proposal.

## 2.51 오버라이드 컨트롤 : final

어떤 경우 프로그래머는 가상 함수가 오버라이드 되는 것을 막고 싶을 때도 있을 것입니다. 이는 `final` 지시자를 넣음으로써 해결할 수 있습니다. 예를 들어

---

```
struct B {
    virtual void f() const final; // 오버라이드 하지 않는다
    virtual void g();
};

struct D : B {
    void f() const; // 오류: D::f 가 final 인 B::f 를 오버라이드 하려 했다
    void g(); // 가능
};
```

---

오버라이딩을 막는데에는 적절한 이유들이 있지만, 제가 `final` 이 어떻게 사용되는지 보여준 예제들에서 여러분이 가상 함수가 얼마나 비싼지(expensive)<sup>10</sup>에 대한 잘못된 생각을 가질까봐 (특히 다른 언어에서 얻은 경험에 기반을 해서) 걱정이 됩니다. 만일 여러분이 `final` 지시자를 넣고 싶다면, 왜 넣는지 다시 한번 생각해보시기 바랍니다. 만일 어떤 사람이 이 가상 함수를 다르게 작성할 경우 문법적 오류가 발생할

<sup>9</sup>프로그램의 어떠한 사항을 나타내는 것이 아니라 단순히 프로그래머들에게 이것이 무슨 내용인지를 설명해주는 키워드

<sup>10</sup>보통 어떠한 작업에 비해 비교적 많은 시간이 소요되는 것을 ‘비싸다’라고 표현합니다

가능성이 높을까요? `final` 키워드를 넣게 되면, 나중에 이 함수에 대한 (여러분이 생각하지 못한) 더 나은 구현을 만드는 것을 막을 수 있습니다. 만일 이러한 가능성을 염두에 두지 않는다면 왜 애초에 함수를 `virtual`로 구현하나요? 이러한 질문에 제가 들었던 대부분의 대답들은 다음과 같습니다 : 이 함수는 이 프레임워크의 매우 중요한 함수로, 프레임워크 제작자들이 오버라이드 할 수 있지만, 보통의 유저들이 오버라이드 하는 것은 원치 않는다. 하지만 제 견해는 이러한 주장에 대해 약간 미심쩍다는 생각입니다. 만일 성능 문제로 인해 그냥 오버라이드 하고 싶지 않다면, 그냥 애초부터 가상함수로 선언하지 않는게 많은 경우 낫습니다. C++ 은 Java 가 아닙니다.

---

`final` 역시 맥락적 키워드(contextual keyword) 이므로 다음과 같이 사용할 수도 있습니다.

---

`int final = 7; // 권장하지 않음`

---

### 참고 자료

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]

## 2.52 C99 기능들

높은 수준의 호환성을 유지하기 위해서, C 표준 위원회와의 협동 작업 끝에 C++ 에 여러 마이너한 변화들이 도입되었습니다.

- `long long`.
- 확장 정수 타입
- UCN 변경 [*N2170 == 07 – 0030*]
- narrow/wide 문자열의 덧셈(concatenation)
- VLA(가변 길이 배열) 는 도입 안됨

또한 몇 개의 전처리기 규칙들이 추가되었습니다.

- `__func__`
  - `__STDC_HOSTED__`
  - `_Pragma`: `_Pragma( X )` 는 `#pragma X` 의 확장형
  - `vararg` 매크로
- 

```
#define report(test, ...) ((test)?puts(#test):printf(_ _VA_ARGS_ _))
```

---

- 비어있는 매크로 인자들

많은 수의 표준 라이브러리 기능들이 C99 에서 그대로 도입되었습니다.

### 참고 자료

- Standard: 16.3 Macro replacement.
- [N1568 = 04 – 0008] P.J. Plauger: PROPOSED ADDITIONS TO TR-1 TO IMPROVE COMPATIBILITY WITH C99.

## 2.53 확장 정수 타입

확장 정수 타입이 있다면 어떻게 작동해야 할지 몇 가지 규칙들이 정의되어 있습니다.

참고 자료

- [06 – 0058 = N1988] J. Stephen Adamczyk: Adding extended integer types to C++ (Revision 1).

## 2.54 concurrency 를 이용한 동적 초기화와 소멸

참고 자료

- [N2660 = 08 – 0170] Lawrence Crowl: Dynamic Initialization and Destruction with Concurrency (Final proposal).

## 2.55 쓰레드 지역 저장 (`thread_local`)

참고 자료

- [N2659 = 08 – 0169] Lawrence Crowl: Thread-Local Storage (Final proposal).

## 2.56 예외의 복사 및 재던지기(rethrow)

어떻게 하면 예외를 받은(catch) 다음에, 다른 쓰레드에 다시 던질(throw) 수 있을까요? 표준 18.8.5 - 예외의 전파(Exception Propagation)에 나와 있는 멋진 방법으로 수행 할 수 있습니다.

- `exception_ptr current_exception();` 현재 처리하고 있는 예외를 가리키는 `exception_ptr` 객체를 리턴한다. 만일 어떠한 예외도 처리되고 있지 않다면 null `exception_ptr` 객체를 리턴한다. `exception_ptr` 이 가리키고 있는 한 예외 객체는 유효하다.

- `void rethrow_exception(exception_ptr p);`  
`template<class E> exception_ptr copy_exception(E e);`

// 위는 아래를 한 것과 동일하다

```

try {
    throw e;
} catch(...) {
    return current_exception();
}

```

이는 특히 한 쓰레드에서 다른 쓰레드로 예외를 전달할 때 유용합니다.

## 2.57 Extern 템플릿

다중 인스턴스화를 막기 위해서 템플릿 특수가 명시적으로 선언될 수 있습니다. 예를 들어서

---

```
#include "MyVector.h"

extern template class MyVector<int>; // 아래의 암시적 인스턴스화를 막는다
// MyVector<int> 는 다른 곳에서 명시적으로 인스턴스화된다.

void foo(MyVector<int>& v)
{
    // 여기 안에서 vector 를 사용
}
```

---

다른 곳에서는 아마 다음과 같이 보일 것입니다.

---

```
#include "MyVector.h"

template class MyVector<int>; // 사용자들에게 MyVector 를 사용할 수 있다
```

---

이는 컴파일러와 링커가 많은 양의 쓸모없는 일들을 피할 수 있는 좋은 방법입니다.

참고 자료

- Standard 14.7.2 Explicit instantiation
- [N1448 = 03 – 0031] Mat Marcus and Gabriel Dos Reis: [Controlling Implicit Template Instantiation](#).

## 2.58 인라인 네임스페이스

인라인 네임스페이스(namespace)는 라이브러리의 개발이 ‘버전(versioning)’의 형태로 진행될 수 있도록 도와주는 메커니즘이라 볼 수 있습니다. 예를 들어

---

```
// file V99.h:
inline namespace V99 {
    void f(int); // V98 버전보다 더 발전됨
    void f(double); // 새로운 기능
    // ...
}
```

---

```
// file V98.h:
namespace V98 {
    void f(int); // 무언가를 한다
    // ...
}
```

---

```
// file Mine.h:
namespace Mine {
#include "V99.h"
#include "V98.h"
}
```

---

여기에서 우리는 두 가지 버전의 (최신 버전 V99 와 이전 버전의 V98) 네임스페이스 Mine 을 볼 수 있습니다. 예를 들어 여기서 특정 버전을 보고 싶다면

---

```
#include "Mine.h"
using namespace Mine;
// ...
V98::f(1); // 옛날 버전
V99::f(1); // 새로운 버전
f(1); // 디폴트 버전
```

---

중요한 점은 `inline` 지시자가, 중첩된(nested) 네임 스페이스에서, 마치 전체를 감싸는 네임스페이스 안에 정의되어 있는 것처럼 만들어준다는 것입니다.

Mine 을 사용하는 유저로 하여금 디폴트로 V99 가 아닌 V98 을 사용하도록 할 수 없습니다.

참고 자료

- Standard 7.3.1 Namespace definition [7]-[9].

## 2.59 명시적인 타입 변환 연산자

C++ 98 은 암/명시적인 생성자를 지원합니다. 즉, 명시적인 생성자로 정의되어 있는 타입 변환의 경우, 오직 명시적인 타입 변환을 해야지만 사용 할 수 있고 암시적인 생성자의 경우, 암시적인 타입 변환도 사용 가능합니다.

---

```
struct S { S(int); }; // 보통의 생성자는 암시적 변환을 정의한다.
S s1(1); // 가능
S s2 = 1; // 가능
void f(S);
f(1); // 가능하지만 문제가 생길 수 있다 – 만일 S 가 vector 였다면?
```

---

```
struct E { explicit E(int); }; // 명시적 생성자
E e1(1); // 가능
E e2 = 1; // 오류
void f(E);
f(1);
// 오류 (예상치 못하게 발생하는 것들을 막는다 – 예를 들어 int 로 부터 std::vector 의 생성자 호출)
```

---

하지만 생성자 만이 타입 변환을 지원하는 유일한 메커니즘은 아닙니다. 여러분이 클래스를 수정할 수 없다면 다른 클래스에 타입 변환 연산자를 정의할 수 있습니다. 예를 들어

---

```
struct S { S(int) { } /* ... */ };

struct SS {
    int m;
    SS(int x) :m(x) { }
    operator S() { return S(m); } // 왜냐하면 S 에는 S(SS) 가 없기 때문
};

SS ss(1);
S s1 = ss; // 가능; 마치 암시적 생성자 처럼
S s2(ss); // 가능; 마치 암시적 생성자 처럼
void f(S);
f(ss); // 가능; 마치 암시적 생성자 처럼
```

---

불행이도, 이전 까지는 명시적인 타입 변환 연산자는 없었습니다. C++ 11 은 이제 타입 변환 연산자가 명시적일 수 있도록 허락합니다. 예를 들어

---

```
struct S { S(int) { } };

struct SS {
    int m;
    SS(int x) :m(x) { }
    explicit operator S() { return S(m); } // 왜냐하면 S 에는 S(SS) 가 없기 때문
};

SS ss(1);
S s1 = ss; // 오류; 마치 암시적 생성자 처럼
S s2(ss); // 가능; 마치 암시적 생성자 처럼
void f(S);
f(ss); // 오류; 마치 암시적 생성자 처럼
```

---

### 참고 자료

- Standard: 12.3 Conversions
- [N2333 = 07 – 0193] Lois Goldthwaite, Michael Wong, and Jens Maurer: [Explicit Conversion Operator \(Revision 1\)](#).

## 2.60 알고리즘 개선점들

표준 라이브러리 알고리즘들은 몇 개의 새로운 알고리즘을 도입함으로써, 새로운 언어 기능들로 인한 향상된 구현(implementation), 그리고 새로운 언어 기능들로 인한 편의성을 통해 향상되었습니다.

새로운 알고리즘들은 다음과 같습니다. :

---

```

bool all_of(Iter first, Iter last, Pred pred);
bool any_of(Iter first, Iter last, Pred pred);
bool none_of(Iter first, Iter last, Pred pred);

Iter find_if_not(Iter first, Iter last, Pred pred);

OutIter copy_if(InIter first, InIter last, OutIter result, Pred pred);
OutIter copy_n(InIter first, InIter::difference_type n, OutIter result);

OutIter move(InIter first, InIter last, OutIter result);
OutIter move_backward(InIter first, InIter last, OutIter result);

pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last, OutIter1 out_true, OutIter2 out_false, Pred pred);
Iter partition_point(Iter first, Iter last, Pred pred);

RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last);
RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last, Compare comp);
bool is_sorted(Iter first, Iter last);
bool is_sorted(Iter first, Iter last, Compare comp);
Iter is_sorted_until(Iter first, Iter last);
Iter is_sorted_until(Iter first, Iter last, Compare comp);

bool is_heap(Iter first, Iter last);
bool is_heap(Iter first, Iter last, Compare comp);
Iter is_heap_until(Iter first, Iter last);
Iter is_heap_until(Iter first, Iter last, Compare comp);

T min(initializer_list<T> t);
T min(initializer_list<T> t, Compare comp);
T max(initializer_list<T> t);
T max(initializer_list<T> t, Compare comp);
pair<const T&, const T&> minmax(const T& a, const T& b);
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
pair<const T&, const T&> minmax(initializer_list<T> t);
pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp);
pair<Iter, Iter> minmax_element(Iter first, Iter last);
pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp);

void iota(Iter first, Iter last, T value);
// 범위 [first, last) 안에 있는 원소들에 대해 반복자 i 가 ++value 를 하며 *i = value 를 수행한다.

```

---

- 이동(move) 연산의 효과 : 이동 연산은 복사 보다 훨씬 효과적이라 볼 수 있습니다. 예를 들어 이동 연산을 기반으로 한 `std::sort()`나 `std::set::insert()` 등의 함수는 복사를 기반으로 한 버전보다 15 배나 더 빠르다고 밝혀졌습니다. 사실 이러한 사실은 크게 인상적이지는 않는데 왜냐하면 이미 표준 라이브러리에서 (예를 들어 `vector`나 `string`) 이동 연산 없이도 이동 연산과 동일한 효과를 내는 최적화 작업들을 수행하고 있었습니다. (최적화된 `swap` 함수를 통해) 하지만 여러분의 사용자 정의 타입이 이동 연산이 있다면, 이러한 표준 라이브러리에서 제공하는 최적화된 성능을 뽑아낼 수 있게 됩니다. 이동 연산을 이용해서 스마트 포인터 `unique_ptr` 컨테이너의 간단하고 효율적인 `sort` 연산을 살펴보세요 :

---

```
template<class P> struct Cmp<P> { // *P 값들을 비교
    bool operator() (P a, P b) const { return *a < *b; }
}

vector<std::unique_ptr<Big>> vb;
// vb 를 unique_ptr 의 Big 객체들로 채운다.

sort(vb.begin(),vb.end(),Cmp<Big>()); // auto_ptr 로는 하지 마세요!
```

---

- 람다의 사용 : 오랜 시간 동안, 사람들은 위의 `Cmp<T>`와 같은 표준 라이브러리 알고리즘을 위해 함수 객체를 만들어야 한다는 사실에 불만을 토로하였습니다. 이는 특히 문제가 되었던 것이 C++ 98에서는 지역 함수 객체를 인자로 전달할 수 없었기에 큰 함수를 작성하는 것이 많이 불편하였습니다. 하지만 람다는 이러한 작업들을 ‘인라인’으로 할 수 있게 도와줍니다.

---

```
sort(vb.begin(),vb.end(),[](unique_ptr<Big> a, unique_ptr<Big> b) { return *a < *b; });
```

---

저는 람다가 처음에는 좀 과다하게 사용될 것이라 예상됩니다. (다른 강력한 메커니즘들 처럼)

- 초기화자 리스트 사용 : 때때로, 초기화자 리스트는 매우 편리하게 사용될 수 있습니다. 예를 들어 아래처럼 문자열 변수들을 만들고, 비교 연산시 대소문자 구분을 하지 않습니다.

---

```
auto x = max({x,y,z},Nocase());
```

---

## 참고 자료

- 25 Algorithms library [algorithms]
- 26.7 Generalized numeric operations [numeric.ops]
- Howard E. Hinnant, Peter Dimov, and Dave Abrahams: [A Proposal to Add Move Semantics Support to the C++ Language](#). [*N1377 = 02 – 0035*].

## 2.61 컨테이너 개선점들

새로운 언어 기능들과 십년간의 경험 끝에 표준 컨테이너들에는 무슨 일이 생겼을까요? 먼저, 당연하게도 새로운 것들이 추가되었습니다 : `array` (고정 크기 컨테이너), `forward_list` (단열 연결 리스트), 그리고

unordered 컨테이너들 (해시 테이블들). 그리고 새로운 기능들, 예를 들어 초기화자 리스트나, 우측값 참조, 가변인자 템플릿, 그리고 `constexpr` 이 사용 가능해졌지요.

예를 들어 `std::vector` 를 살펴봅시다.

- 초기화자 리스트 : 가장 눈에 띠는 개선점으로, 초기화자 리스트 생성자를 사용해서 초기화자들의 리스트를 인자로 받을 수 있게 됩니다.

---

```
vector<string> vs = { "Hello", ", ", "World!", "\n" };
for (auto s : vs) cout << s;
```

---

- 이동 연산자 : 컨테이너는 이제 이동 생성자와, 이동 대입 연산자를 가지게 됩니다. 이에 가장 큰 효과를 보는 점은 바로 우리는 효과적으로 함수에서 컨테이너를 리턴할 수 있게 됩니다.

---

```
vector<int> make_random(int n)
{
    vector<int> ref(n);
    for(auto& x : ref) x = rand_int(0,255); // 난수가 생성된다
    return ref;
}

vector<int> v = make_random(10000);
for (auto x : make_random(1000000)) cout << x << '\n';
```

---

여기서 중요한 점은 어떠한 `vector` 들도 복사되지 않는다는 점입니다. 이 함수를 자유 저장소에 할당되는 벡터로 구현한다면 여러분들은 메모리 관리에 신경 써주어야 할 것입니다. 이 함수를 `make_random` 의 인자로 난수 값을 채울 `vector` 를 전달한다면, 덜 명백한 코드가 될 뿐더러 오류를 만들 확률이 높아집니다.

- 향상된 삽입(push) 연산들 : 제가 제일 좋아하는 컨테이너 연산은 `push_back()` 으로 컨테이너가 우아하게 크기를 증가시킬 수 있게 해줍니다.

---

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.push_back({s,i});
```

---

이는 `s` 와 `i` 를 바탕으로 `pair<string, int>` 를 생성해서 `vp` 에 이동 시킬 것입니다. 다시 강조하지만 이동은 복사가 아닙니다. `push_back` 함수에는 우측값 레퍼런스를 받는 버전이 있기 때문에 문자열의 이동 생성자의 장점을 취할 수 있습니다. 또한 위 코드에서 일원화 된 초기화자 문법을 사용해서 불필요한 코드를 줄였습니다.

- Emplace 연산 : `push_back()` 은 이동 생성자를 사용하여 이전의 복사 기반의 작업 보다 더 효율적으로 수행할 수 있었습니다. 하지만 좀 더 생각해봅시다. 애초에 복사/이동을 할 필요가 있을까요? 그냥 벡터 안에 공간을 만들어서 그 안에 처음부터 생성을 해버리면 되지 않을까요? 이러한 일들을

하는 작업을 emplace (그 곳에 넣는다 라는 의미) 라 합니다. 예를 들어 `emplace_back()` 함수를 보면

:

---

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.emplace_back(s,i);
```

---

`emplace` 는 가변인자 템플릿 인자를 취하며, 이를 이용해서 원하는 타입의 객체를 생성해줍니다. `emplace_back()` 이 `push_back()` 보다 더 효율적인지에 대한 여부는 타입과 라이브러리와 가변인자 템플릿의 구현에 대한 문제입니다. 만일 여러분이 뭐가 더 빠른지가 중요하게 생각된다면 직접 비교해보세요. 아니면, 그냥 보기에도 좋은 것을 택해도 됩니다. `vp.push_back(s, i);` 나 `vp.emplace_back(s, i);` 를 말이지요. 저는 `push_back()` 버전을 선호합니다.

- 범위 있는(scoped) 할당자 : 컨테이너들은 이제 실제 할당자 객체를 가질 수 있고 이를 통해 중첩된 할당에 사용할 수 있습니다. (즉 컨테이너 안에서의 원소의 할당)

명백하게도, 컨테이너들만이 표준 라이브러리에서 새로운 기능의 장점을 수혜받은 것만은 아닙니다.

- 컴파일 타입 계산(compile time evaluation) : `constexpr` 은 컴파일 타임에 `bitset`, `duration`, `char_traits`, `array`, `atomic types`, 난수, `complex` 등이 계산될 수 있게 해줍니다. 어떤 경우에는, 이를 통해 성능 향상을 할 수도 있고, 복잡한 로우 레벨 코드나 매크로들을 사용하지 않을 수도 있습니다.
- Tuples: `tuple` 들은 가변인자 템플릿 없이는 구현할 수 없었을 것입니다.

## 2.62 범위 있는 할당자

컨테이너 객체들의 단순화와 간결화를 위해서 C++ 98에서는 컨테이너들이 특정한 상태를 포함하는 할당자를 지원하도록 요구하지 않았습니다: 할당자 객체들은 컨테이너 객체 내부에 저장될 필요가 없었습니다. 이는 C++ 11에서도 디폴트로 설정되어 있지만, 특정 상태를 가지는 할당자 (예를 들어 어디에 할당할지에 대한 포인터를 가지는 할당자) 를 가질 수 있게 됩니다. 예를 들어

---

```
template<class T> class Simple_alloc { // C++98
    // 데이터 없음
    // 할당자 관련 평범한 것들
};

class Arena {
    void* p;
    int s;
public:
    Arena(void* pp, int ss);
    // p[0..ss-1] 에서 할당함
```

```

};

template<class T> struct My_alloc {
    Arena& a;
    My_alloc(Arena& aa) : a(aa) { }
    // 할당자 관련 평범한 것들
};

Arena my_arena1(new char[100000],100000);
Arena my_arena2(new char[1000000],1000000);

vector<int> v0; // 디폴트 할당자를 통해 할당

vector<int,My_alloc<int>> v1(My_alloc<int>{my_arena1}); // my_arena1 로 부터 할당

vector<int,My_alloc<int>> v2(My_alloc<int>{my_arena2}); // my_arena2 로 부터 할당

vector<int,Simple_alloc<int>> v3; // Simple_alloc 을 이용해 할당

```

---

통상적으로 길어지는 코드는 `typedef` 를 통해 줄일 수 있습니다. 디폴트 할당자와 `Simple_alloc` 이 벡터 객체의 어떠한 공간도 사용하지 않는다는 것을 보장할 수 없습니다. 하지만, 라이브러리의 우아한 템플릿 메타프로그래밍(template metaprogramming)을 통해서 이를 보장할 수 있게 됩니다. 따라서 할당자 타입을 사용하는 것은 아주 약간의 오버헤드(만일 객체가 실제로 상태를 포함하고 있다면 - 예를 들어 `My_alloc`) 만을 차지하게 됩니다.

컨테이너와 사용자 정의 할당자를 사용할 때 발생할 수 있는 문제로 : 컨테이너와 원소가 같은 할당 장소에 있어도 될까요? 예를 들어 만일 여러분이 `Your_allocator` 를 이용해서 `Your_string` 을 할당하고 저는 `My_allocator` 를 이용해서 `My_vector` 들의 원소들을 할당하다고 할 때, `My_vector<Your_allocator>` 의 문자열 원소들은 어떤 할당자가 사용되어야 할까요? 그 해결책은 컨테이너에게 원소로 어떠한 할당자가 전달될 수 있는지 말해주는 것에 있습니다. 예를 들어 제가 할당자 `My_alloc` 을 사용하고 `vector<string>` 을 `vector` 원소와 문자열 할당 모두에 사용하고 싶다고 해봅시다. 먼저 저는 `My_alloc` 객체를 받는 문자열을 만들어야 할 것입니다.

---

```

using xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
// My_alloc 을 사용하는 문자열

```

---

그리고 저는 이제 이러한 문자열과 `My_alloc` 객체를 받고 객체를 문자열에 전달하는 벡터를 만들어야 합니다.

---

```

using svec = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;

```

---

마지막으로 우리는 `My_alloc<xstring>` 타입의 할당자를 만들면 됩니다.

---

```

svec v(svec::allocator_type(My_alloc<xstring>{my_arena1}));

```

---

이제 svec 은 My\_alloc 이 문자열들을 할당하는 문자열 벡터가 됩니다. 여기서 새로운 것은 표준 라이브러리 adapter (wrapper 타입) scoped\_allocator\_adaptor 로 문자열이 My\_alloc 를 또한 사용하도록 알려줍니다. 참고로 adaptor 는 (자명하게) `My_alloc<xstring>` 을 `My_alloc<char>` 로 바꿀 수 있습니다. 따라서 우리에게는 4 가지의 다른 방법들이 가능합니다.

---

```
// vector 와 string 은 자기 자신의 (디폴트) 할당자를 사용한다.

using svec0 = vector<string>;
svec0 v0;

// vector 만 My_alloc 을 사용하고 string 은 자기 자신의 (디폴트) 할당자를 사용한다

using svec1 = vector<string, My_alloc<string>>;
svec1 v1(My_alloc<string>{my_arena1});

// vector 와 string 이 My_alloc 을 사용한다.

using xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
using svec2 = vector<xstring, scoped_allocator_adaptor<My_alloc<xstring>>>;
svec2 v2(scoped_allocator_adaptor<My_alloc<xstring>>{my_arena1});

// vector 는 My_alloc 을 사용하고 string 은 My_string_alloc 을 사용한다

using xstring2 = basic_string<char, char_traits<char>, My_string_alloc<char>>;
using svec3 = vector<xstring2, scoped_allocator_adaptor<My_alloc<xstring>, My_string_alloc<char>>>;
svec3 v3(scoped_allocator_adaptor<My_alloc<xstring2>, My_string_alloc<char>>{my_arena1, my_string_arena});
```

---

자명하게도, 첫번째 형태인 svec0 은 가장 흔한 형태이지만, 메모리 관련한 성능이 심각하게 중요한 시스템에서는 다른 버전들 (예를 들어 svec2) 들이 더 중요합니다. 사용자는 `typedef`을 사용함으로써 가독성을 높일 수 있습니다. `scoped_allocator_adaptor2` 는 `scoped_allocator_adaptor` 의 다른 버전입니다.

참고 자료

- Standard: 20.8.5 Scoped allocator adaptor [allocator.adaptor]
- Pablo Halpern: [The Scoped Allocator Model \(Rev 2\)](#). [`N2554 = 08 – 0064.`]

## 2.63 std::array

표준 컨테이너 `array` 는 고정 크기의 임의 접근 원소들의 시퀀스로, `<array>` 에 정의되어 있습니다. 이는 원소들을 보관하는 공간 이외에는 어떠한 오버헤드도 가지고 있지 않고, 자유 저장소를 사용하는 것도 아닙니다. 이는 초기화자 리스트를 사용해서 초기화 되는 것도 가능하고, 자신들의 크기 (원소의 개수) 도 알고 있습니다. 또한, 여러분이 직접적으로 요구하지 않는 한 (explicitly) 포인터로 변환하지도 않습니다. 쉽게 말해 이는 내장된 배열과 같지만 이것들이 포함하고 있던 고질적인 문제는 없지요.

---

```
array<int,6> a = { 1, 2, 3 };
a[3]=4;
int x = a[5]; // 원소들이 디폴트로 0 이 되어서 x 는 0 이 된다.
```

---

```
int* p1 = a; // 오류 : std::array 는 암시적으로 포인터로 변환되지 않는다.
int* p2 = a.data(); // 첫 번째 원소를 가리키는 포인터를 리턴한다
```

---

참고로 여러분은 길이가 0 인 배열을 가질 수 있지만, 초기화자 리스트를 통해서 길이를 알아내는 기능은 없습니다.

---

```
array<int> a3 = { 1, 2, 3 }; // 오류 : 사이즈를 알수 없음
array<int,0> a0; // 가능 : 원소가 없다
int* p = a0.data(); // 정의되지 않음
```

---

표준 array 의 기능들은 임베디드 시스템 프로그래밍에 적용하기에 매우 매력적입니다 (그리고 비슷한 제한된 자원의, 성능 혹은 안전성이 매우 중요한 시스템들). array 은 시퀀스 컨테이너기 때문에 통상적인 멤버 타입들과 함수들을 제공합니다. (vector 처럼)

---

```
template<class C> C::value_type sum(const C& a)
{
    return accumulate(a.begin(),a.end(),0);
}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// ...
int x1 = sum(a10);
double x2 = sum(a1000);
int x3 = sum(v);
```

---

배열은 자식에서 부모 클래스 간의 변환이 불가능합니다.

---

```
struct Apple : Fruit { /* ... */ };
struct Pear : Fruit { /* ... */ };

void nasty(array<Fruit*,10>& f)
{
    f[7] = new Pear();
};

array<Apple*,10> apples;
// ...
nasty(apples); // 오류 : array<Apple*,10> 에서 array<Fruit*,10>; 로 변환할 수 없습니다
```

---

만일 이것이 허용되었다면 사과가 배를 포함하고 있을지도 모릅니다.

참고 자료

- Standard: 23.3.1 Class template array

## 2.64 std::forward\_list

표준 컨테이너 `forward_list` 는 `<forward_list>` 에 정의되어 있고, 단열 연결 리스트입니다. 이는 앞방향 반복(forward iteration) 만 가능하고, 여러분이 원소를 삽입하거나 지우더라도 원소들이 이동하지 않는다는 것을 보장합니다. 이는 최소한의 공간 (빈 리스트는 1 워드 정도 될 것입니다) 을 차지하고, `size()` 함수를 제공하지 않습니다. (그래서 `size` 멤버 변수를 보관하고 있지 않도록)

---

```
template <ValueType T, Allocator Alloc = allocator<T> >
    requires NothrowDestructible<T>
class forward_list {
public:
    // 보통의 컨테이너 관련 내용들
    // size() 없음
    // 역방향 반복자 없음
    // back() 혹은 push_back() 없음
};
```

---

### 참고 자료

- Standard: 23.3.3 Class template `forward_list`

## 2.65 unordered 컨테이너

정렬되지 않은(unordered) 컨테이너는 해시 테이블(hash table) 의 한 종류입니다. C++ 11 은 4 가지 표준을 제공합니다.

- `unordered_map`
- `unordered_set`
- `unordered_multimap`
- `unordered_multiset`

이들은 원래 `hash_map` 등으로 불렸어야 했는데, 이 이름을 사용하게 되면 너무나 많은 호환성 문제가 발생하게 되서 위원회는 새로운 이름인 `unordered_map` 등으로 붙이게 되었습니다. “정렬되지 않는”은 `map` 과 `unordered_map` 의 구분짓는 가장 큰 특징으로 “여러분이 `map` 에서 탐색(iterate) 하게 된다면, `map` 에서 제공하는 미만(less than) 비교 연산자(`i`)를 이용해서 특정한 순서로 수행하게 됩니다. 하지만 `unordered_map` 의 값 탑입은 미만 비교 연산자를 가지고 있지 않고, 해시 테이블 역시 원래 순서를 제공하지 않습니다. 역으로, `map` 의 원소들은 해시 함수를 가질 필요가 없습니다.

`unordered_map` 을 사용하는 간단한 아이디어는 `map` 을 필요한 곳에 최적화된 버전으로 사용하는 것입니다. 예를 들어

---

```
map<string,int> m {
    {"Dijkstra",1972}, {"Scott",1976}, {"Wilkes",1967}, {"Hamming",1968}
```

```

};

m["Ritchie"] = 1983;
for(auto x : m) cout << '{' << x.first << ',' << x.second << '}';

unordered_map<string,int> um {
    {"Dijkstra",1972}, {"Scott",1976}, {"Wilkes",1967}, {"Hamming",1968}
};
um["Ritchie"] = 1983;
for(auto x : um) cout << '{' << x.first << ',' << x.second << '}';

```

---

m 안에서 탐색은 알파벳 순서로 수행됩니다. 반면에 um 안에서의 탐색은 그렇지 않죠. m 과 um 의 탐색은 서로 아주 다른 방식으로 구현되어 있습니다. m 의 탐색의 경우  $\log_2(m.size())$  미만의 비교가 필요한 반면, um 의 경우 단 한번의 해시 함수 호출과 한 번 혹은 그 이상의 비교 연산을 통해 찾을 수 있게 됩니다. 적은 수의 원소 (몇십개 정도)에 대해서는 무엇이 빠른지 판단하기 어렵지만 그보다 훨씬 많은 큰 수의 원소들 (수천개 정도)에 대해서는 unordered\_map에서의 탐색이 map 보다 훨씬 빠르게 수행됩니다.

#### 참고 자료

- Standard: 23.5 Unordered associative containers.

## 2.66 std::tuple

표준 라이브러리 <tuple>에 정의되어 있는 tuple (N - tuple)은 순서가 있는 N 개의 시퀀스로, N은 0부터 구현마다 다른 어떠한 최대값 까지의 값을 가질 수 있습니다. 여러분은 tuple을 지정된 원소 타입들을 멤버로 가지는 이름이 없는 구조체라고 생각하면 됩니다. 특히, tuple의 원소들은 효율적으로 저장될 수 있습니다. 왜냐하면 tuple은 링크된 구조(linked structure)가 아니기 때문이지요. tuple의 원소들의 타입은 명시적으로 지정되던지, make\_tuple()을 통해 유추될 수도 있습니다. 그리고 원소들은 get()을 이용한 인덱스로 접근 가능합니다.

```

tuple<string,int> t2("Kylling",123);

auto t = make_tuple(string("Herring"),10, 1.23); // t의 타입은 tuple<string,int,double>
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);

```

---

tuple은 우리가 타입이 다른 원소들의 리스트를 컴파일 타입에 사용하고 싶지만, 이들을 포함하는 이름 있는 클래스를 정의하고 싶지 않을 때 사용하면 됩니다. 예를 들어 tuple은 std::function과 std::bind 함수들에서 인자들을 보관하기 위해 내부적으로 사용됩니다.

가장 많이 사용되는 경우는 2 개의 원소를 가지는 tuple로, 마치 pair와 같습니다. 하지만 pair는 표준 라이브러리 std::pair로 직접적으로 지원됩니다. pair은 tuple을 초기화 하기 위해 사용될 수 있지만, 그 반대로는 사용할 수 없습니다.

비교 연산자 (`==`, `!=`, `<`, `<=`, `>`, and `>=`) 들은 `tuple` 의 비교 가능한 원소 탑입들에 대해 정의되어 있습니다.

참고 자료

- Standard: 20.5.2 Class template tuple
- Variadic template paper
- Boost::tuple

## 2.67 std::function and std::bind

`bind` 와 `function` 은 `<functional>` 에 정의되어 있는 표준 함수 객체입니다. (물론 이들 뿐만이 아니라 엄청나게 많은 다른 함수 객체들도) 이들은 함수들과 함수 인자들을 처리하는데 사용됩니다. `bind` 는 함수를 받아서 (혹은 함수 객체들, 아니면 (...) 문법으로 호출할 수 있는 아무거나) 한 개 혹은 그 이상의 인자 함수들의 ‘bound’ 된 인자들을 가지는 함수 객체를 리턴합니다.

예를 들어

---

```
int f(int,char,double);
auto ff = bind(f,_1,'c',1.2); // 리턴 탑입을 유추
int x = ff(7); // f(7,'c',1.2);
```

---

인자들을 결합(bind)하는 것은 “커링(currying)<sup>11</sup>” 이라 부릅니다. `_1` 은 위치 지정 객체로, `f` 가 `ff` 를 통해 호출되었을 때 `ff` 의 첫번째 인자가 들어갈 위치입니다. 첫 번째 인자는 `_1` 이라 불리고, 두번째는 `_2`, .. 입니다. 예를 들어

---

```
int f(int,char,double);
auto frev = bind(f,_3,_2,_1); // 인자의 역순
int x = frev(1.2,'c',7); // f(7,'c',1.2);
```

---

다행인 것은 `auto` 를 통해 `bind` 의 결과 탑입을 우리가 직접 지정해 주지 않아도 됨을 알 수 있습니다. 오버로드 된 함수들의 인자들을 `bind` 하는 것은 가능하지만. 우리가 `bind` 하고 싶은 오버로드 된 함수를 정확히 지정해 주어야만 합니다.

---

```
int g(int);
double g(double); // g() 는 오버로드 됨

auto g1 = bind(g,-1); // 오류 : 어떤 g()
auto g2 = bind((double(*)(double))g,-1); // 가능 : 다만 보기 안좋다
```

---

`bind()` 는 두 가지 형태가 있습니다. 하나는 옛날 버전으로 여러분이 명시적으로 리턴 탑입을 지정해 주어야 합니다.

<sup>11</sup>영어 단어의 원래 의미는 향신료로 절임한다는 뜻이다

---

```
auto f2 = bind<int>(f,7,'c',_1); // 명시적 리턴 타입
int x = f2(1.2); // f(7,'c',1.2);
```

---

두 번째 버전은 첫번째 버전이 C++ 98에서 구현히 불가능 하였기에 널리 사용되고 (간단한) 버전입니다. `function`은 (...) 형태로 호출할 수 있는 모든 것들의 값을 보관할 수 있는 타입입니다. 특히, `bind`의 결과는 `function`에 대입될 수 있습니다. `function`은 사용하기 매우 간단합니다. 예를 들어

```
function<float (int x, int y)> f; // 함수 객체를 만든다
```

```
struct int_div {
    float operator()(int x, int y) const { return ((float)x)/y; }
};

f = int_div(); // 대입
cout << f(5, 3) << endl; // function 객체를 통해 호출
std::accumulate(b,e,1,f); // 잘 전달된다
```

---

멤버 함수들은 인자 하나만 추가함으로써 마치 자유 함수처럼 사용할 수 있습니다.

```
struct X {
    int foo(int);
};

function<int (X*, int)> f;
f = &X::foo; // 멤버를 가리키는 포인터

X x;
int v = f(&x, 5); // x에 5를 전달하며 X::foo() 호출
function<int (int)> ff = std::bind(f,&x,_1); // f의 첫번째 인자는 &x
v=ff(5); // x.foo(5)를 호출
```

---

`function`은 콜백 함수, 연산(operation) 등을 인자로 전달할 때 사용하면 유용합니다. 또한, 이는 C++ 98의 표준 라이브러리 함수인 `mem_fun_t`나 `pointer_to_unary_function` 등의 대체로 사용할 수도 있습니다. 비슷하게도 `bind()`는 `bind1()`이나 `bind2()`의 대체로 볼 수 있습니다.

### 참고 자료

- Standard: 20.7.12 Function template bind, 20.7.16.2 Class template function
- Herb Sutter: [Generalized Function Pointers](#). August 2003.
- Dougla Gregor: [Boost.Function](#).
- Boost::bind

## 2.68 unique\_ptr

`unique_ptr` (<memory>에 정의됨)은 엄격한 소유 관계를 표현하기 위한 것으로 사용됩니다.

- 자신이 가리키는 객체를 소유한다.
- 복사 생성 및 복사 대입이 불가능 하지만 이동 생성 및 이동 대입은 가능하다.
- 객체를 가리키는 포인터를 저장하며, 자기 자신이 파괴될 때 그 객체도 파괴한다. (예를 들어 자신이 정의된 범위를 벗어날 때)

`unique_ptr`의 사용 예로는

- 동적으로 할당된 메모리에 대한 예외 안전성(exception safety)<sup>12</sup>을 제공한다.
- 함수에게 동적으로 할당된 메모리의 소유권을 전달한다.
- 함수에서 동적으로 할당된 메모리를 반환한다.
- 컨테이너에 포인터들을 보관한다.

사실 `auto_ptr`이 이와 같은 역할을 했어야 했지만 (이는 C++ 98에서는 작성할 수 없었다), `unique_ptr`이 우측값 레퍼런스와 이동 연산에 매우 의존적이므로 현재에 완성되었습니다. 아래는 통상적인 예외 안전하지 않는 코드입니다.

---

```
X* f()
{
    X* p = new X;
    // 작업을 수행한다 - 예외를 던질 수도 있다.
    return p;
}
```

---

그 해결책은 자유공간의 객체를 가리키는 포인터를 `unique_ptr`에 저장하는 것입니다.

---

```
X* f()
{
    unique_ptr<X> p(new X); // {new X} 도 가능하지만 = new X 는 안됨
    // 작업을 수행한다 - 예외를 던질 수도 있다.
    return p.release();
}
```

---

이제 예외가 던져졌다고 해도 `unique_ptr`이 암시적으로 가리키고 있는 객체를 파괴할 것입니다. 이는 간단한 RAII(자원의 획득은 자원의 초기화이다 - Resource Acquisition Is Initialization)라 볼 수 있지요. 하지만, 우리가 꼭 내장 포인터를 리턴해야 되지 않는 한, 이 작업을 `unique_ptr`을 리턴함으로써 수행할 수 있습니다.

---

<sup>12</sup>예외가 발생하더라도 메모리에 잘 해제될 수 있도록 보장한다

---

```
unique_ptr<X> f()
{
    unique_ptr<X> p(new X); // {new X} 도 가능하지만 = new X 는 안됨
    // 작업을 수행한다 - 예외를 던질 수도 있다
    return p; // 소유권이 f() 밖으로 전달된다.
}
```

---

우리는 f 를 다음과 같이 사용할 수 있습니다.

```
void g()
{
    unique_ptr<X> q = f(); // 이동 생성자를 이용해서 이동한다.
    q->memfct(2); // q 를 사용
    X x = *q; // 가리키고 있는 객체를 복사
    // ...
} // q 와 q 가 소유하던 객체는 파괴된다.
```

---

unique\_ptr 은 이동 연산을 가지고 있기 때문에 f() 를 호출함으로써 우측값으로 q 를 초기화 하는 것은 단순히 소유권을 q 로 이전하는 것이라 볼 수 있습니다.

unique\_ptr 을 사용하는 또 다른 예로 컨테이너에 포인터를 저장할 때입니다. 물론 내장 포인터를 사용해도 되지만 예외 안전성이라던지, 원소의 파괴를 보장하기 위해서는 이를 이용하는 것이 낫습니다.

---

```
vector<unique_ptr<string>> vs { new string{"Doug"}, new string{"Adams"} };
```

---

unique\_ptr 은 내장 포인터와 약간의 오버헤드를 포함하고 있지만, 내장 포인터를 사용할 때에 비해 거의 차이가 나지 않습니다. 특히 unique\_ptr 은 어떠한 동적 확인(dynamic checking)을 제공하지 않습니다.

참고 자료

- the C++ draft section 20.7.10
- Howard E. Hinnant: [unique\\_ptr Emulation for C++03 Compilers](#).

## 2.69 shared\_ptr

shared\_ptr 은 공유되는 소유권을 표현하기 위해 사용됩니다. 즉, 코드의 두 부분에서 특정한 데이터에 접근하고 싶지만, 둘 다 독점적인 소유권(exclusive ownership) 이 없을 때 (이 객체를 파괴시킬 책임이 없을 때) 사용됩니다. shared\_ptr 은 카운트 되는 포인터로, 카운트 수가 0 이 될 때 가리키고 있는 객체가 파괴됩니다. 아래는 예시로

---

```
void test()
{
    shared_ptr<int> p1(new int); // 카운트는 1
    {
```

```

shared_ptr<int> p2(p1); // 카운트는 2
{
    shared_ptr<int> p3(p1); // 카운트는 3
} // 카운트가 다시 2 가 된다
} // 카운트가 다시 1 이 된다
} // 카운트가 0 이 되며 int 는 파괴된다.

```

---

좀 더 현실적인 예제로는 노드들을 포인터로 가리키는 일반적인 그래프에서 특정한 포인터가 가리키고 있는 노드를 제거하고 싶지만, 이 노드를 가리키고 있는 다른 포인터들을 알 수 없을 때 사용할 수 있습니다. 만일 노드가 어떠한 자원(예를 들어 파일 핸들러는 노드가 파괴되기 전에 반드시 이를 반환(close)해야 합니다)을 가지고 있다면, 그 자원을 반환하기 위해 꼭 소멸자가 필요 합니다. 여러분은 또한 `shared_ptr` 을 가비지 컬렉터 (garbage collector) 로 사용하는 것으로 생각할 수 있는데 사실 가비지 컬렉터는 메모리만 관리할 수 있지만, `shared_ptr` 을 사용한다면 메모리 뿐만이 아니라 파일 핸들러와 같은 자원을 제대로 관리할 수 있게 됩니다. 예를 들어

```

struct Node { // 참고로 이 노드는 다른 노드들이 가리킬 수 있음
    shared_ptr<Node> left;
    shared_ptr<Node> right;
    File_handle f;
    // ...
};

```

---

여기서 `Node` 의 소멸자 (암시적으로 생성된 소멸자 역시 상관 없다) 는 자기 노드들의 서브 노드들을 파괴할 것입니다. 즉 `left` 와 `right` 들의 소멸자가 호출됩니다. 이 때 `left` 가 `shared_ptr` 이므로, 만일 `Node` 가 가리키고 있는 `left` 중 자신이 마지막으로 가리키고 있던 것일 때만 파괴됩니다. `right` 역시 동일하게 동작합니다. 만일 여러분이 포인터의 소유권을 이전하고 싶다면 `shared_ptr` 대신에 `unique_ptr` 을 사용해야 합니다. 이는 `unique_ptr` 을 만든 목적일 뿐더러 `shared_ptr` 에 비해 더 적은 비용으로 수행합니다. 만일 여러분의 프로그램이 카운트 되는 포인터들을 팩토리 함수(factory function)들의 리턴 타입으로 사용했다면, `unique_ptr` 을 사용하는 버전으로 업그레이드 하는 것을 추천합니다.

그리고 주의할 점으로, 메모리 누수를 막기 위해 생각 없이 포인터들을 그냥 `shared_ptr` 로 바꾸면 안됩니다. `shared_ptr` 은 만병통치약이 아닐 뿐더러, 이들을 사용하는데 비용이 없는 것은 아닙니다.

- 원형으로 링크된 구조일 경우 `shared_ptr` 은 메모리 누수를 발생시킬 수 있습니다. (이 경우 `weak_ptr` 을 사용하는 것을 권합니다)
- 소유권이 공유되는 객체들의 경우 보통의 범위 안의(scoped) 객체들 보다 오래 메모리에 남아 있습니다.
- 다중 쓰레드 환경에서 `shared_ptr` 를 이용하는 것은 비용이 비쌀 수 있습니다.
- 공유되는 객체들의 소멸자는 예상과 다른 때에 실행될 수 있고, 공유되는 객체들에 대한 알고리즘의 경우 공유되지 않는 객체들에 대한 알고리즘 보다 더 오류를 발생시키기 쉽습니다.

공유되는 소유권을 나타내는 `shared_ptr` 은 제가 생각하기에 이상적인 것은 아닙니다. 차라리 명확한 소유자와 명확한 생존 시간(lifespan) 을 가지는 것이 더 낫습니다.

#### 참고 자료

- the C++ draft: `shared_ptr` (20.7.13.3)

## 2.70 weak\_ptr

`weak_ptr` 는 흔히 `shared_ptr` 로 이루어진 데이터 구조에서 루프를 없일 때 사용하는 것이라 설명됩니다. 제가 생각하기에 `weak_ptr` 은 포인터로

- (오직) 존재하는 것에 접근하기 위해
- (다른 것들에 의해) 파괴될 수도 있고
- 마지막으로 사용 뒤에 반드시 소멸자를 호출해야 하는 것

옛날의 ‘별똥별 게임’ 의 구현을 생각해보세요. 모든 별똥별들은 ‘게임’ 에 의해 소유되지만, 각각의 별똥별들은 근처의 별똥별들의 움직임을 추적해서 충돌을 처리해야 합니다. 충돌은 통상적으로 한 개 혹은 그 이상의 별똥별들의 소멸을 발생시킵니다. 각각의 별똥별들은 그 이웃에 근처에 위치한 별똥별들의 목록들을 보관해야 합니다. 이러한 리스트에 들어 있는 별똥별의 경우 별똥별을 항상 ‘생존’ 시키면 안됩니다 (따라서 `shared_ptr` 은 적합하지 않죠). 반면에 다른 별똥별이 어떤 별똥별의 값을 사용하고 있을 경우 (예를 들어 충돌의 효과를 계산하기 위해) 그 별똥별은 파괴되면 안됩니다. 그리고 명백하게도, 별똥별의 소멸자는 리소스 (컴퓨터 그래픽 등)를 반환하기 위해서 반드시 호출되어야 합니다. 우리가 별똥별 리스트로 부터 필요로 하는 것은 “객체가 존재한다면 이를 한동안 참조하고 있는 것” 입니다. `weak_ptr` 가 바로 그 역할을 하지요.

---

```
void owner()
{
    // ...
    vector<shared_ptr<Asteroid>> va(100);
    for (int i=0; i<va.size(); ++i) {
        // 새로운 별똥별의 이웃한 별똥별들을 계산한다. ...
        va[i].reset(new Asteroid(weak_ptr<Asteroid>(va[neighbor])));
        launch(i);
    }
    // ...
}
```

---

`reset()` 은 새로운 객체를 `shared_ptr` 로 가리키는 포인터입니다. 위 코드에서 나타나듯이, 저는 “소유자” 를 매우 단순화 시켜서, 각 별똥별마다 1 개의 근처 별똥별을 추적하도록 코드를 단순화 시켰습니다. 중요한 점은, 우리가 근처의 별똥별을 `weak_ptr` 로 가리키게 하였다는 점입니다. 소유자는 `shared_ptr` 을 이용해서 객체의 소유권을 계속 나타낼 수 있습니다. 충돌 계산을 처리하는 부분은 아마 아래와 같을 것입니다.

---

```

void collision(weak_ptr<Astroid> p)
{
    if (auto q = p.lock()) { // p.lock 은 p 의 객체를 가리키는 shared_ptr 를 리턴한다
        // 별똥별이 존재한다면 충돌 여부를 계산한다
    }
    else {
        //별똥별이 이미 파괴되었다 : 그냥 이를 가리켰던 weak_ptr 을 제거하면 된다.
    }
}

```

---

중요한 점은 소유자가 게임을 종류하고 모든 별똥별들을 파괴시킬 경우에도 (소유권을 나타내는 shared\_ptr 들을 파괴함으로써) 계산 중에 있는 모든 별똥별들은 연산이 올바르게 수행이 됩니다. (왜냐하면 p.lock() 을 하면 그 별똥별에 대한 shared\_ptr 를 가지게 되므로 소멸되지 않는다.) 저는 weak\_ptr 을 사용하는 경우는 shared\_ptr 을 사용하는 것보다 훨씬 더 드물게 사용될 것이라 생각되며, unique\_ptr 이 shared\_ptr 보다 더 많이 쓰이기를 바랍니다. 왜냐하면 unique\_ptr 이 소유권을 표현하는데 좀더 단순하고 (그리고 더 효율적) 논리적으로 더 낫기 때문입니다.

참고 자료

- the C++ draft: Weak\_ptr (20.7.13.3)

## 2.71 가비지 컬렉션 ABI

가비지 컬렉션(garbage collection - GC) 은 자동적으로 메모리의 참조되지 않는 부분을 재활용 하는 기능으로 C++ 에서는 구현 상에서 반드시 필요로 해야 되는 부분은 아닙니다. 하지만 C++ 11 은 사용자가 ABI (application binary interface) 를 사용한다면 GC 가 제공하는 기능을 그대로 쓸 수 있습니다.

포인터와 생존 시간(lifetime) 에 대한 규칙은 “안전하게 유도된 포인터(safely derived pointer)” 으로써 표현됩니다. 이는 쉽게 말해 new 혹은 그것의 서브 객체들에 의해 할당된 것의 포인터를 의미합니다. 아래, “안전하지 않게 유도된 포인터(위장된 포인터, 올바르게 작동하기를 기대하는 프로그램에서 하지 말아야 할 것)”에 대한 몇 예시를 보여주겠습니다.

- 먼저 포인터가 다른 곳을 잠시 가리키도록 합니다.

---

```

int* p = new int;
p+=10;
//가비지 컬렉터가 실행됩니다.
p-=10;
*p = 10; // int 가 아직도 거기에 있을 것이라 보장 가능?

```

---

- 포인터를 int 로 숨깁니다.

---

```

int* p = new int;
int x = reinterpret_cast<int>(p);

```

---

---

```
p=0;
//가비지 컬렉터가 실행됩니다.
p = reinterpret_cast<int*>(x);
*p = 10; // int 가 아직도 거기에 있을 것이라 보장 가능?
```

---

- 사실 이것보다 더 난잡한 트릭들이 있는데, 예를 들어 I/O 를 생각해보세요 - 비트들을 각기 다른 워드들에 훑뿌려 놓는다.

사실 포인터들을 위장하기 위한 적법한 이유들이 있습니다. (예를 들어 매우 메모리가 제한적인 시스템에서 xor 트릭을 이용한다던지) 하지만 일부 프로그래머들이 기대하는 것만큼 많지는 않습니다. 프로그래머는 포인터가 하나도 없는 곳(예를 들어 이미지)을 사용할 수 있고 콜렉터가 거기를 가리키는 포인터를 찾을 수 없다면 메모리 또한 환수 할 수 없게 됩니다.

---

```
void declare_reachable(void* p); // p 부터 시작하는 메모리 구역
                                // 자신의 크기를 기억하는 어떤
                                // 할당자에 의해 할당됨
                                // 반드시 반환되면 안된다.

template<class T> T* undeclare_reachable(T* p);

void declare_no_pointers(char* p, size_t n); // p[0..n] 에는 어떤 포인터들도 없다
void undeclare_no_pointers(char* p, size_t n);
```

---

프로그래머는 포인터에 어떠한 포인터 안전(pointer safety) 규칙이 적용되는지 알아낼 수 있고, 강제적으로 적용하도록 할 수 있습니다 .

---

```
enum class pointer_safety {relaxed, preferred, strict };

pointer_safety get_pointer_safety();
```

---

3.7.4.3[4]: 만일 안전하게 유도된 포인터 값이 아닌 포인터 값이 역참조(dereference) 혹은 해제(deallocate)되거나 참조되어 있는 완전한 객체가 동적 저장 단계(dynamic storage duration)에 있고 접근할 수 있다고 이전에 선언되지 않았다면 (20.7.13.7), 이 때의 동작은 정의되지 않았습니다.

- relaxed** : 안전하게 유도된 것, 안전하지 않게 유도된 것 모두 동일하게 취급합니다. 마치 C 와 C++ 98 같이 말이지요. 하지만 사실 이것은 제 의도가 아니였습니다 - 저는 만일 사용자가 객체에 대한 유효한 포인터를 보관하지 않는 경우 GC 를 사용할 수 있게 하고 싶었습니다.
- preferred** : **relaxed** 와 같지만, 가비지 컬렉터가 메모리 누수 탐지기나, 나쁜 포인터들의 역참조를 탐지하는데 사용될 수 있습니다.
- strict** : 안전하게 유도된 포인터와 안전하지 않게 유도된 포인터 둘이 다르게 취급됩니다. 즉, 가비지 컬렉터는 실행될 수 있지만, 안전하지 않게 유도된 포인터들을 무시할 수 있습니다.

여러분이 무엇을 골라야 할지에 대한 표준은 없습니다.

참고 자료

- the C++ draft 3.7.4.3
- the C++ draft 20.7.13.7
- Hans Boehm's [GC page](#)
- Hans Boehm's [Discussion of Conservative GC](#)
- final proposal
- Michael Spertus and Hans J. Boehm: [The Status of Garbage Collection in C++0X](#). ACM ISMM'09.

## 2.72 메모리 모델

메모리 모델은 대부분의 프로그래머들이 현대 컴퓨터 하드웨어들의 세세한 내용에 대해 생각하지 않고도 프로그래밍 할 수 있도록 하드웨어 설계자들과 컴파일러 제작자들이 합의를 본 것입니다. 메모리 모델 없이는, 쓰레딩(threading), 잠금(locking), 잠금 없음(lock-free) 의 극히 일부분만 말이 될 것입니다.

보장되는 내용 중 주요한 것으로 : 두 개의 쓰레드가 실행될 때, 서로 방해하지 않고 각기 다른 장소의 메모리에 접근할 수 있습니다. 하지면 “과연 메모리 장소(memory location)” 이 무엇인가요? 메모리 장소는 스칼라 타입의 객체이거나, 0 이 아닌 너비를 가지는 인접한 비트 필드 시퀀스라고 볼 수 있습니다. 예를 들어 여기 S 는 정확히 4 개의 각기 다른 메모리 장소들을 가지고 있습니다.

---

```
struct S {
    char a; // 장소 #1
    int b:5, // 장소 #2
    int c:11,
    int :0, // 참고로 :0 은 특별하게 처리됨
    int d:8; // 장소 #3
    struct {int ee:8;} e; // 장소 #4
};
```

---

왜 이것이 중요할까요? 왜 이것이 명백한 것이 아닐까요? 위 사실이 언제나 참이지 않을까요? 문제는 여러 연산이 병렬(pararell)로 진행될 수 있기 때문에 여러 관련되지 않아 (보이는) 연산들이 동시에 진행될 수 있다는 점입니다. 따라서 메모리 하드웨어에서 이상한 일들일 벌어질 수 있지요. 사실, 컴파일러의 지원 없이는 명령어 문제라던지, 데이터 파이프라인(pipelining)이나 캐시를 사용하는 등에 관련한 것들을 모두 응용 프로그램 프로그래머들이 직접 처리하기란 불가능한 일입니다. 이는 심지어 두 개의 쓰레드가 공유 데이터를 사용하지 않는 아주 간단한 것에 대해서도 말이지요. 예를 들어 아래의 두 개의 독립적인 쓰레드를 봅시다.

---

```
// 쓰레드 1
char c;
c = 1;
int x = c;

// 쓰레드 2
```

```
char b;
b = 1;
int y = b;
```

---

좀더 현실적으로 생각하기 위해, 각 쓰레드에 대해 독립적으로 컴파일 해서 컴파일러/최적화기가 메모리 접근을 없앨 수 없고, 단순히 c 와 b 를 무시하고 직접적으로 x 와 y 를 1 로 초기화 할 수 없다고 해봅시다. 그렇다면 x 와 y 에 대한 가능한 값은 무엇이 될까요? C++ 11 에 따르면, 유일하게 가능한 답은 둘다 1 이 되는 것입니다. 이것이 흥미로운 이유는, 만일 전통적인 C 나 C++ 컴파일러의 좋은 동시성(currency) 기능을 사용한다면 가능한 답으로 둘 다 0 이 되거나 (물론 낮을 확률로), 1 과 0, 0 과 1, 그리고 1 과 1 모두가 될 수 있다는 것입니다. 이는 실제로 많이 보고되었지요. 어떻게 그게 가능하냐고요? 링커가 c 와 b 를 서로 바로 옆에 할당한다는 것입니다 (즉 같은 워드 안에) – C 나 C++ 의 1990 년 대표준은 이에 어떠한 것도 규정화 하고 있지 않지요. 하지만 그 때의 C++ 은 다른 모든 언어들 처럼 실제 병행 하드웨어를 염두에 두고 있지 않았습니다. 하지만 현대의 대부분의 프로세서들은 단일 문자에 쓰고 읽을 수 없습니다. 반드시 전체 워드에 읽고 쓰는 작업을 수행해야 하므로, c 에 대한 대입은 실제로 ‘c 를 포함하고 있는 워드를 읽은 뒤에 c 에 부분을 치환하고, 다시 전체 워드에 쓴다’ 라는 개념으로 수행된 것입니다. 따라서 b 에 대한 대입도 동일하게 수행되므로, 두 개의 쓰레드들이 엉켜서 메모리를 공유하지 않는 데도 불구하고 위와 같은 문제를 일으킬 수 있다는 점이지요.

따라서 C++ 11 에서는 다른 메모리 위치에 대해 위와 같은 문제들을 일으키지 않도록 보장합니다. 정확히 말해, 메모리 장소는 잠금(locking)을 하지 않는 한 두 개의 쓰레드에 의해 안전하게 접근할 수 없습니다. 참고로 같은 워드에 있는 비트필드는 서로 다른 메모리 장소가 아니므로, 비트필드를 가지는 구조체를 잠금 없이 쓰레드 사이에서 공유하면 안됩니다. 위와 같은 내용 빼고는 C++ 메모리 모델은 여러분 모두가 예상하는 바와 같습니다.

사실 언제나 저수준(low level)의 동시성 문제에 대해 생각하는 것은 쉬운 일이 아닙니다. 예를 들어

---

```
// x==0 와 y==0 에서 시작한다
```

```
if (x) y = 1; // 쓰레드 1
```

---

```
if (y) x = 1; // 쓰레드 2
```

위에 문제가 있을까요?

다행이도 우리는 이미 현대 시대에 적응되었고, (제가 아는) 모든 현재의 C++ 컴파일러는 위 코드에 대해 한 개의 유일한 답을 내놓습니다. 사실 꽤 골치아픈 코드들에 대해서도 컴파일러는 유일한 답을 내놓습니다. 사실 C++ 은 오랜 세월동안 고도의 병행 시스템의 프로그래밍에 사용 되어 왔습니다. 표준은 이러한 것을 좀 더 개선시킬 뿐이지요.

- Standard: 1.7 The C++ memory model [intro.memory]
- Paul E. McKenney, Hans-J. Boehm, and Lawrence Crowl: [C++ Data-Dependency Ordering: Atomics and Memory Model](#). [N2556 = 08 – 0066].

- Hans-J. Boehm: [Threads basics](#), HPL technical report 2009-259. “what every programmer should know about memory model issues.”
- mHans-J. Boehm and Paul McKenney: [A slightly dated FAQ on C++ memory model issues.](#)

## 2.73 쓰레드(Thread)

쓰레드는 프로그램의 실행(execution)/연산(computation)을 나타내는 것입니다. C++ 11 에서는, 대부분의 현대 컴퓨팅에서처럼 쓰레드가 (주로) 다른 쓰레드들과 주소 공간을 공유합니다. 반면에 프로세스(process)의 경우 일반적으로 다른 프로세스들과는 데이터를 공유하지 않지요. C++ 은 과거에 많은 종류의 하드웨어와 운영체제에서 쓰레드 구현을 포함하고 있었습니다. 현재의 C++ 은 표준 라이브러리에 쓰레드 라이브러리 ABI 가 포함이 되었습니다.

많은 두꺼운 책들과 수만개의 논문들에서 동시성, 병렬성, 그리고 쓰레딩을 다루고 있습니다. 이 FAQ 는 그 겉표면을 살짝 다루는 정도에 불과합니다. 만일 여러분이 병행 프로그래밍을 제대로 하고 싶다면 이 메뉴얼이나, 표준 문서를 보는 것으로는 충분하지 않습니다.

쓰레드는 std::thread 를 함수나 함수 객체로 생성함으로써 실행될 수 있습니다. (람다도 가능)

---

```
#include<thread>

void f();

struct F {
    void operator()();
};

int main()
{
    std::thread t1{f}; // f() 는 다른 쓰레드에서 실행된다.
    std::thread t2{F()}; // F() 늦은 쓰레드에서 실행된다.
}
```

---

불행이도, 위 코드에서는 f() 나 F() 안에서 무엇을 하던 간에 아마 유용한 결과를 낼 수는 없을 것입니다. 문제는 프로그램이 t1 이 f() 를 호출하기 전이나 다음에 혹은 t2 가 F() 를 실행하거나 실행하기 전에 종료될 수 있다는 점입니다. 우리는 두 작업이 완료될 때 까지 기다려야 합니다.

---

```
int main()
{
    std::thread t1{f}; // f() 가 다른 쓰레드에서 실행된다.
    std::thread t2{F()}; // F() 늦은 쓰레드에서 실행된다.

    t1.join(); // t1 을 기다림
    t2.join(); // t2 를 기다림
}
```

---

`join()` 는 쓰레드들이 종료되기 전 까지 프로그램이 종료되지 않도록 보장하는 역할을 합니다. ‘join’의미는 쓰레드가 종료될 때 까지 기다려라 라는 의미입니다. 통상적으로 우리는 실행될 작업에 인자로 전달하고 싶을 것입니다. (저는 쓰레드에서 실행될 것을 작업(task) 라고 부를 것입니다). 예를 들어

---

```

void f(vector<double>&);

struct F {
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();
};

int main()
{
    std::thread t1{std::bind(f,some_vec)}; // f(some_vec) 는 다른 쓰레드에서 실행된다.
    std::thread t2{F(some_vec)}; // F(some_vec)() 는 다른 쓰레드에서 실행된다.
    t1.join();
    t2.join();
}

```

---

기본적으로 표준 라이브러리는 함수 객체와 이 인자들을 결합해줍니다. 일반적으로, 우리는 실행된 작업에서 그 결과를 받고 싶을 것입니다. 평범한 작업의 경우, 리턴값을 줄 수 없습니다. 저는 이를 위해 `std::future` 를 추천하고 싶군요. 그에 대한 대안으로 우리는 작업에 인자로 어디에 리턴값을 넣을지 알려줄 수 있습니다. 예를 들어

---

```

void f(vector<double>&, double* res); // res 에 결과값을 넣는다.

struct F {
    vector<double>& v;
    double* res;
    F(vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()(); // res 에 결과값을 넣는다.
};

int main()
{
    double res1;
    double res2;

    std::thread t1{std::bind(f,some_vec,&res1)}; // f(some_vec,&res1) 는 다른 쓰레드에서 실행된다.
    std::thread t2{F(some_vec,&res2)}; // F(some_vec,&res2)() 는 다른 쓰레드에서 실행된다.

    t1.join();
}

```

---

```
t2.join();

    std::cout << res1 << ' ' << res2 << '\n';
}
```

---

하지만 오류들에 경우 어떨까요? 만일 작업이 예외를 던지면 어떻게 될까요? 만일 작업이 예외를 던졌는데 받을 수 없다면 `std::terminate()` 가 호출될 것입니다. 이는 보통 프로그램이 종료 된다는 의미이지요. 우리는 이러한 것들을 되도록 피하려고 할 것입니다. `std::future` 은 예외를 부모/혹은 호출하는 쓰레드에 전달할 수 있습니다. 이는 제가 `future` 을 좋아하는 한 가지 이유이기도 하지요. 다른 방법으로는 오류 코드를 리턴하는 방법도 있습니다. 쓰레드가 프로그램의 범위를 벗어나게 되면 프로그램은 종료되게 됩니다. 이는 반드시 피해야 할 상황이지요.

쓰레드가 종료되도록 요구하거나 강제로 종료(kill) 하는 방법은 없습니다. 우리는 이를 위해 몇 가지 대책을 남겨놓았습니다.

- 직접 인터럽트 메커니즘(interruption mechanism) 을 만든다 (즉, 쓰레드 간에 공유할 수 있는 데이터를 만들어서 쓰레드로 하여금 이 데이터를 지속적으로 체크하도록 하는데 호출 쓰레드가 이 값을 설정하면, 쓰레드가 종료되도록 한다)
- `thread::native_handle()` 을 이용해서 운영체제의 입장에서 쓰레드에 접근하여 종료시킨다.
- 프로세스를 종료시킨다 (`std::quick_exit()`)
- 프로그램을 종료시킨다 (`std::terminate()`)

이와 같은 내용은 위원회가 모두 합의한 바입니다. 특히 POSIX 의 대표들은 쓰레드 종료(thread cancellation)에 대해 열렬하게 반대하였는데, 반면에 많은 C++ 의 리소스 모델은 소멸자에 의존하고 있습니다. 따라서 모든 시스템의 모든 가능한 응용 프로그램에 맞는 완벽한 해결책은 없습니다.

쓰레드가 가지는 기본적인 문제는 데이터 경쟁(data race)가 있습니다. 이는 같은 주소 공간에서 돌아가는 두 개의 쓰레드가 독립적으로 객체에 접근 할 때 정의되지 않은 결과를 야기할 수 있다는 점입니다. 따라서 어떤 쓰레드가 특정 객체에 쓰고, 다른 쓰레드가 객체를 읽을 때 그들은 누가 어떤 작업을 먼저 끝낼지 ‘경쟁(race)’을 하게 됩니다. 물론 그 결과 역시 정의되어 있지 않고, 매우 예측 불가 입니다. 결과적으로 C++ 11 은 프로그래머들이 데이터 경쟁을 방지하기 위해 일련의 규칙과 보장을 제공해줍니다.

- C++ 표준 라이브러리 함수는 현재의 쓰레드 말고 다른 쓰레드들이 객체에 (함수의 인자로 간접/혹은 직접적으로 접근되는 것 말고) 직접/혹은 간접적으로 접근하는 것을 막아야 합니다. (`this` 도 포함)
- C++ 표준 라이브러리 함수는 현재의 쓰레드 말고 다른 쓰레드들이 객체에 (함수의 `non-const` 인자로 간접/혹은 직접적으로 접근되는 것 말고) 직접/혹은 간접적으로 수정하는 것을 막아야 합니다. (`this` 도 포함)
- C++ 표준 라이브러리 구현은 같은 시퀀스에 다른 원소들이 동시에 수정될 때 데이터 경쟁 막을 필요가 있습니다.

스트림 객체, 스트림 버퍼 객체, 혹은 다중 쓰레드를 사용한 C 라이브러리 스트림은 명시되지 않는 한, 동시 접근 시 데이터 경쟁을 발생시킬 수 있습니다. 따라서 여러분이 어떻게 제어할지 알지 않는다면 두 개 쓰레드의 출력 스트림을 공유하지 말아야 합니다.

여러분은

- 정의 된 시간 동안 쓰레드를 대기 시킬 수 있습니다.
- 데이터 접근을 상호 배제(mutual exclusion)을 통해 제어할 수 있습니다.
- 데이터 접근을 lock 을 이용해 제어할 수 있습니다.
- 조건 변수를 이용해서 다른 작업을 대기할 수 있습니다.
- future 을 이용해서 쓰레드의 값을 리턴할 수 있습니다.

참고 자료

- Standard: 30 Thread support library [thread]
- 17.6.4.7 Data race avoidance [res.on.data.races]
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.: [Multi-threading Library for Standard C++ \(Revision 1\)](#) [N2497 = 08 – 0007]
- H.-J. Behm, L. Crowl: C++ object lifetime interactions with the threads API N2880 = 09 – 0070.
- L. Crowl, P. Plauger, N. Stoughton: Thread Unsafe Standard Functions N2864 = 09 – 0054.
- WG14: [Thread Cancellation](#) N2455 = 070325.

## 2.74 상호 배제(Mutual exclusion)

`mutex` 는 멀티 쓰레드 시스템에서 데이터 접근을 제어할 때 사용하는 객체입니다. 가장 간단한 사용으로

---

```
std::mutex m;
int sh; // 공유되는 데이터
// ...
m.lock();
// 공유되는 데이터를 조작한다.
sh+=1;
m.unlock();
```

---

`lock()` 과 `unlock()` 지역 안에 한 개의 쓰레드 만이 들어갈 수 있습니다. (보통 임계 구역(critical region) 이라 부릅니다). 만일 첫 번째 쓰레드가 임계 구역 안에서 실행 중일 때 두 번째 쓰레드가 `m.lock()` 하려고 한다면, 두 번째 쓰레드는 첫 번째 쓰레드가 `m.unlock()` 에 도달하기 전 까지 제한됩니다. 이는 매우 간단하지요. 하지만 간단하지 않는 문제는 `mutex` 를 위와 같이만 사용해서 문제를 안일으키게 하는 것입니다. 만일 쓰레드가 `unlock()` 하는 것을 잊은다면 어떨까요. 만일 쓰레드가 같은 `mutex` 에 대해 두

번 lock() 을 한다면 어떨까요? 만일 쓰레드가 unlock() 하기 전에 매우 오랜 시간 대기 한다면 어떨까요? 만일 쓰레드가 일을 처리하기 위해 두 개의 mutex 를 lock() 해야 된다면 어떨까요? 이에 대한 모든 대답들로 책을 쓸 수도 있습니다. 여기에 (그리고 잠금 섹션에서) 간단한 기초만 설명하겠습니다. lock() 외에도, mutex 는 try\_lock() 작업이 있어서, 블록되는 위험성 없이 임계 지역에 들어갈 수 있게 하는 것입니다.

---

```
std::mutex m;
int sh; // 공유되는 데이터
// ...
if (m.try_lock()) {
    // 공유되는 데이터를 조작한다.
    sh+=1;
    m.unlock();
} else {
    // 다른 일들을 한다
}
```

---

recursive\_mutex 는 쓰레드에서 한 번 이상 얻어질 수 있는 mutex 입니다.

---

```
std::recursive_mutex m;
int sh; // 공유되는 데이터
// ...
void f(int i)
{
    // ...
    m.lock();
    // 공유되는 데이터를 조작한다.
    sh+=1;
    if (--i>0) f(i);
    m.unlock();
    // ...
}
```

---

위 코드에서 f() 는 자기 자신을 호출합니다. 물론 재귀적인 호출이 위처럼 직접 수행되는 것 말고도, f() 가 g() 를 호출하고, g() 가 h() 를 호출하고 h() 가 f() 를 호출하는 등으로 이루어질 수도 있습니다.

만일 우리가 mutex 를 다음 10 초 동안 얻어야 된다면 어떨까요? timed\_mutex 클래스는 이를 지원합니다. 이 클래스의 작업들은 try\_lock() 의 특수화 된 형태입니다.

---

```
std::timed_mutex m;
int sh; // 공유되는 데이터
// ...
if (m.try_lock_for(std::chrono::seconds(10))) {
    // 공유되는 데이터를 조작한다.
    sh+=1;
    m.unlock();
```

```

    }
else {
    // mutex 를 얻지 못했으므로 다른 일들을 한다.
}

```

---

`try_lock_for()` 은 상대적인 시간을 인자로 받습니다. 만일 여러분이 정확히 고정된 시간동안 대기시키고 싶다면, `time_point` 인 `try_lock_until()` 을 사용하셔도 됩니다.

```

std::timed_mutex m;
int sh; // 공유되는 데이터
// ...
if (m.try_lock_until(midnight)) {
    //공유되는 데이터를 조작한다
    sh+=1;
    m.unlock();
}
else {
    // mutex 를 얻지 못했으므로 다른 일들을 한다.
}

```

---

사실 위 코드에서 자정(midnight) 이라 한 것은 장난이고, 저수준의 `mutex` 와 같은 메커니즘들에 대해서는 스케일이 거의 밀리초에 가까울 것입니다.

물론 `recursive_timed_mutex` 도 있습니다.

`mutex` 는 자원으로 고려되며 (통상적으로 실제 리소스를 표현하기 때문), 이를 사용하기 위해서는 적어도 두 개 쓰레드에서 참조할 수 있어야 합니다. 따라서, `mutex` 는 복사되거나 이동될 수 없습니다. (여러누이 하드웨어의 입력 레지스터의 복사본을 만들 수 없는 것처럼 말이죠)

놀랍게도 `lock()` 과 `unlock()` 짹을 맞추는 일이 매우 어려울 수 도 있습니다. 복잡한 제어 구조, 오류, 예외를 생각해보세요. 만일 여러분이 선택할 수 있다면 `lock` 을 이용해서 `mutex` 를 관리하세요.

참고 자료

- Standard: 30.4 Mutual exclusion [thread.mutex]
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.: [Multi-threading Library for Standard C++ \(Revision 1\)](#)

## 2.75 잠금(Locks)

`lock` 은 `mutex` 의 레퍼런스를 보관할 수 있는 객체로, `mutex` 를 `lock` 의 소멸 동안 `unlock()` 할 수 있습니다. (예를 들어 블록 범위를 벗어날 때) 쓰레드는 `lock` 을 이용해서 예외 시 `mutex` 소유권을 안전하게 처리할 수 있습니다. 다시 말해 `lock` 은 상호 배제에서의 RAII (자원의 획득은 초기화 이다 - Resource Acquisition Is Initialization) 를 구현하고 있다고 볼 수 있습니다. 예를 들어

```
std::mutex m;
```

```

int sh; // 공유되는 데이터
// ...
void f()
{
    // ...
    std::unique_lock lck(m);
    // 공유되는 데이터를 조작한다
    sh+=1;
}

```

---

`lock` 은 이동될 수 있고 (`lock` 이 비지역 리소스의 지역 소유권을 표현한다는 점에서) 복사될 수는 없습니다. (어떠한 복사본이 자원/`mutex` 를 소유할 것인가?)

사실 `mutex` 가 할 수 있는 거의 대부분의 기능들은 `unique_lock` 이 좀 더 안전하게 수행할 수 있습니다. 예를 들어

```

std::mutex m;
int sh; // 공유되는 데이터
// ...
void f()
{
    // ...
    std::unique_lock lck(m,std::defer_lock); // lock 은 하지만 mutex 는 얻지 않는다
    // ...
    if (lck.try_lock()) {
        // 공유되는 데이터를 조작한다
        sh+=1;
    }
    else {
        // 다른 작업들을 한다
    }
}

```

---

비슷하게도, `unique_lock` 은 `try_lock_for()` 과 `try_lock_until()` 을 지원합니다. 여러분이 `mutex` 대신에 `lock` 을 사용함으로써 얻을 수 있는 것들로, 예외 처리나, `unlock()` 을 하지 않는 문제에서 해방될 수 있습니다. 병행 프로그래밍에서 우리가 얻을 수 있는 도움은 모두 얻어야 합니다.

만일 두 개의 `mutex` 에 의해 표현되는 두 개의 자원을 필요로 한다면 어떨까요? 순진한 방법으로 `mutex` 들을 순서대로 얻는 것입니다.

```

std::mutex m1;
std::mutex m2;
int sh1; // 공유되는 데이터
int sh2
// ...

```

---

```
void f()
{
    // ...
    std::unique_lock lck1(m1);
    std::unique_lock lck2(m2);
    // 공유되는 데이터를 조작한다
    sh1+=sh2;
}
```

---

이는 심각한 실행 흐름상에 문제가 있을 수 있는데, 다른 쓰레드가 m1 과 m2 를 다른 순서로 얻으려고 할 수 있는데, 각각이 처리하기 위해 lock 을 필요로 하므로, 둘 중 하나가 영원히 기다리게 될 수 있습니다. (이를 데드록(deadlock) 이라 합니다) 시스템의 많은 lock 들 중에서 이 것이야 말로 가장 문제라 볼 수 있습니다. 결과적으로 표준 lock 은 두 개의 함수를 제공해서 안전하게 두 개 이상의 lock 을 얻을 수 있게 지원합니다.

---

```
void f()
{
    // ...
    std::unique_lock lck1(m1,std::defer_lock); // lock 을 만들지만 아직 mutex 를 얻으려 하지 않는다
    std::unique_lock lck2(m2,std::defer_lock);
    std::unique_lock lck3(m3,std::defer_lock);
    lock(lck1,lck2,lck3);
    // 공유되는 데이터를 조작한다
}
```

---

명백하게도 lock() 의 구현은 데드락을 피하기 위해 조심스럽게 구현되어야만 합니다. 실제로, try\_lock() 들을 조심스럽게 쓴다면 이를 수행할 수 있습니다. 만일 lock() 이 모든 lock 을 획득하는데 실패한다면 이는 예외를 던질 것입니다. 실제로 lock() 은 lock(), try\_lock(), unlock() 멤버 함수들(예를 들어 mutex)을 인자로 받을 수 있기에 우리는 어떠한 예외를 lock() 이 던질지 알 수 없습니다. 단지 lock() 이 받는 인자에 따라 다르지요.

만일 여러분이 try\_lock() 을 사용하는 것을 선호한다면, lock() 과 동일한 것이 있습니다.

---

```
void f()
{
    // ...
    std::unique_lock lck1(m1,std::defer_lock); // lock 을 만들지만 아직 mutex 를 얻으려 하지 않는다
    std::unique_lock lck2(m2,std::defer_lock);
    std::unique_lock lck3(m3,std::defer_lock);
    int x;
    if ((x = try_lock(lck1,lck2,lck3)) == -1) { // C 의 세계에 오신 것을 환영합니다
        // 공유되는 데이터를 조작한다
    }
    else {
```

```
// x 는 우리가 얻을 수 없는 mutex 의 인덱스를 의미한다.  
// 예를 들어 lck2.try_lock() 가 실패했다면 x==1  
}  
}
```

참고 자료

- Standard: 30.4.3 Locks [thread.lock]

## 2.76 Time 유틸리티

우리는 종종 어떤 것의 시간을 쟁거나 시간에 관련된 작업을 처리하기 마련입니다. 예를 들어 표준 라이브러리의 `mutex` 나 `lock` 들에 쓰레드가 특정 시간 동안 기다리게 하거나, 특정 시간까지 기다리게 하는 명령을 내릴 수 있습니다. (`time_point`)

만일 여러분이 현재 `time_point` 를 알고 싶다면 `now()` 를 호출하거나, 다음 세 개의 시계들 중 하나를 사용하면 됩니다 : `system_clock`, `monotonic_clock`, `high_resolution_clock`. 예를 들어

```
monotonic_clock::time_point t = monotonic_clock::now();
// 무언가를 한다
monotonic_clock::duration d = monotonic_clock::now() - t;
// 어떤 작업이 d 시간 만큼 소요된다
```

`clock` 은 `time_point` 를 리턴하고 `duration` 은 두 개의 `time_point` 사이 시간입니다. 통상적으로, 여러분이 디테일에 관심이 없다면 `auto` 는 여러분의 친구 입니다.

```
auto t = monotonic_clock::now();
// 무언가를 한다
auto d = monotonic_clock::now() - t;
// 어떤 작업이 d 시간 만큼 소요된다
```

`time` 기능들은 효율적으로 심화된 시스템 지원을 위해서 사용합니다. 이 기능들은 여러분의 달력을 효율적으로 관리하려고 지원하는 것은 아닙니다. 사실, `time` 기능들은 고에너지 물리 쪽에서 요구하는 기준들을 맞추는 것에서 유래하였습니다. 모든 시간 스케일들(세기 부터 피코초 까지)에 대해 표현할 수 있고, 단위, 오타, 반올림 오차 등에 대한 혼란을 막기 위해서, 모든 것들은 컴파일 타입 유리수 처리 패키지에서 처리됩니다. `duration`은 두 개의 부분으로 구성되어 있습니다. 숫자 클록 (한 tick - period) 과, 한 tick 이 무엇을 의미하는지 (초인지 밀리초인지 - duration)에 대한 것입니다. `period`는 `duration` 타입의 일부입니다. 아래 테이블은 표준 헤더 `<radio>`에 SI 시스템 period 들을 정의해 놓은 것입니다.

```

typedef ratio<1, 1000000000> nano;
typedef ratio<1, 1000000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio<10, 1> deca;
typedef ratio<100, 1> hecto;
typedef ratio<1000, 1> kilo;
typedef ratio<1000000, 1> mega;
typedef ratio<1000000000, 1> giga;
typedef ratio<1000000000000000, 1> tera;
typedef ratio<10000000000000000000, 1> peta;
typedef ratio<10000000000000000000000000, 1> exa;
typedef ratio<10000000000000000000000000000000, 1> zetta; //조건부로 지원됨
typedef ratio<100000000000000000000000000000000000000000, 1> yotta; // 조건부로 지원됨

```

---

컴파일 타입 유리수들에는 보통의 산술연산과 비교 연산자가 제공되며, 이를 연산자를 이용해서 `duration` 과 `time_points` 를 조합하는 어떤한 것들도 가능합니다. 이러한 연산들은 오버플로우와 0 으로 나눠짐도 확인이 됩니다. 이 기능이 컴파일 타임 기능이기 때문에 런타임 성능은 걱정하지 않아도 됩니다. 게다가 여러분은 `++, -, +=, -=, *=, /=` 를 `duration` 에 사용할 수 있고 `tp+= d` 와 `tp-= d` 를 `time_point` `tp` 와 `duration d` 에 사용할 수 있습니다. 아래는 `<chrono>` 에 정의된 표준 `duration` 타입들을 사용한 예제입니다.

```

microseconds mms = 12345;
milliseconds ms = 123;
seconds s = 10;
minutes m = 30;
hours h = 34;

auto x = std::chrono::hours(3); // 명시적으로 네임스페이스를 표시해야된다
auto x = hours(2)+minutes(35)+seconds(9);

```

---

여러분은 `duration` 을 분수로 초기화 할 수는 없습니다. 예를 들어 2.5 초는 하지 마세요. 대신 2500 밀리초를 하면 됩니다. 이는 `duration` 이 `tick` 의 개수로 해석되기 때문입니다. 각각의 `tick` 은 `duration` 의 `period` 의 단위가 됩니다. 예를 들어 위에 정의한 `milli` 나 `kilo` 가 되겠지요. 디폴트 단위는 초 입니다. 즉, `period` 가 1 인 `duration` 의 `tick` 은 1 초가 된다는 것이지요. 우리는 명시적으로 `duration` 을 어떻게 표현할 지 정의할 수 있습니다.

```

duration<long> d0 = 5; // 초 (디폴트)
duration<long,kilo> d1 = 99; // 킬로초
duration<long,ratio<1000,1>> d2 = 100; //d1 와 d2 는 같은 타입이다 ("kilo" 의 뜻은 "*1000")

```

---

만일 여러분이 duration 으로 무엇을 하고 싶다면 우리는 단위 (분인지 마이크로 초 인지) 를 알려주어야 합니다. 예를 들어

---

```
auto t = monotonic_clock::now();
//무언가를 한다
nanoseconds d = monotonic_clock::now() - t; // 결과값을 나노초 형태로 원합니다
cout << "something took " << d << "nanoseconds\n";
```

---

우리는 duration 을 부동 소수점 수로 변환할 수도 있습니다.

---

```
auto t = monotonic_clock::now();
//무언가를 한다
auto d = monotonic_clock::now() - t;
cout << "something took " << duration_cast<double>(d).count() << "seconds\n";
```

---

count() 는 tick 의 개수입니다.

참고 자료

- Standard: 20.9 Time utilities [time]
- Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno: [A Foundation to Sleep On](#). N2661 = 08 – 0171. Including “A Brief History of Time” (With apologies to Stephen Hawking).

## 2.77 std::future 와 std::promise

병행(concurrent) 프로그래밍은 여러분이 특히 쓰레드와 잠금을 제대로 이용하기 위해서는 어려울 수 있습니다. 여러분이 비록 조건 변수(condition variable) 과 std::atomic (lock 을 사용하지 않는 프로그래밍에서) 을 이용해도 여전히 여러운건 마찬가지입니다. C++ 11 은 future 와 promise 를 제공해서, 다른 쓰레드의 작업의 결과를 리턴받고, packaged\_task 를 이용해 작업을 생성하는데 사용할 수 있습니다. future 과 promise 의 좋은 점은, 두 개의 작업들 간의 값 교환을 lock 을 사용하지 않고도 수행할 수 있습니다; 특히 ‘시스템이’ 이러한 전달을 효율적으로 구현해줍니다. 이를 구현한 아이디어는 단순합니다: 만일 작업에서 값을 자신을 실행한 쓰레드로 리턴하고 싶을 때, 이는 promise 에 값을 넣습니다. 그리고 특정한 방식으로 promise 에 부착된 future 에 전달된 값이 나타나도록 하지요. 호출자 (작업을 실행한 것) 은 이를 통해 값을 읽을 수 있게 됩니다. 더 간단하게 하려면 async() 를 참조하세요. 표준은 3 가지 형태의 future 을 제공하는데, 간단한 사용을 위한 future, 조금 더 복잡한 상황을 위한 shared\_future 와 atomic\_future 입니다. 여기에서 저는 그냥 future 을 보여줄 것인데, 왜냐하면 이 것이 가장 단순하고 제가 필요로 하는 것들을 모두 할 수 있기 때문입니다. 만일 여러분의 future<X> 가 f 를 호출하였다면, 여러분은 X 의 값을 get() 을 통해 얻을 수 있습니다.

---

```
X v = f.get(); // 우너하는 값이 계산될 때 까지 기다린다
```

---

만일 아직 값이 리턴되지 않았으면, 쓰레드는 그 값이 리턴될 때 까지 기다립니다. 만일 값이 계산될 수 없다면, get() 의 결과 대신 예외를 던질 수 있습니다. 우리는 결과를 위해 기다리고 싶지 않을 수 있는데, 이를 위해 아래와 같이 결과가 왔는지 확인할 수도 있습니다.

---

```

if (f.wait_for(0)) { // get() 을 할 수 있는 값이 있다.
    // 무언가를 한다
}
else {
    // 다른 일을 한다
}

```

---

하지만 future 의 주된 목적은 단순한 get() 을 제공하는 것이지요. promise 의 주요 목적은 간단히 future 의 get() 에 대응시키도록 하는 것입니다. future 과 promise 라 이름 붙인 것은 역사적인 이유가 있기에 저를 비난하지는 말아주세요.

만일 여러분이 promise 를 해서, 결과를 future 에 X 타입으로 얻고 싶다면, 기본적으로 여러분은 2 가지 방법으로 할 수 있습니다. 값을 전달하거나, 예외를 전달하거나 :

---

```

try {
    X res;
    // res 의 값을 계산한다
    p.set_value(res);
}
catch (...) { // res 를 계산할 수 없다
    p.set_exception(std::current_exception());
}

```

---

위 까지는 좋습니다만, 어떻게 하면 각기 다른 쓰레드에 future 와 promise 가 있을 때, future/promise 쌍을 매치시킬 수 있을까요? 사실 future 과 promise 는 이동 될 수 있기 때문에 (복사는 안됨), 여러가지 방법이 있습니다.

packaged\_task 는 작업을 쓰레드로 하여금 실행하기 위한 매우 단순화된 방법입니다. 특히, 이는 알아서 future 와 promise 를 연결시키고, 작업에서 발생 하는 리턴값 혹은 예외에 대한 wrapper 코드를 제공해줍니다. 예를 들어

---

```

double comp(vector<double>& v)
{
    // 작업을 묶는다.
    // 아래에서 하는 작업은 double 배열에 대한 표준 accumulate() 이다.
    packaged_task<double(double*,double*,double)> pt0{std::accumulate<double*,double*,double>};
    packaged_task<double(double*,double*,double)> pt1{std::accumulate<double*,double*,double>};

    auto f0 = pt0.get_future(); // future 를 얻는다
    auto f1 = pt1.get_future();

    pt0(&v[0],&v[v.size()/2],0); // 쓰레드를 시작한다
    pt1(&v[v.size()/2],&v[v.size()],0);
}

```

---

---

```

return f0.get() + f1.get(); // 결과값을 얻는다
}

```

---

### 참고 자료

- Standard: 30.6 Futures [futures]
- Anthony Williams: **Moving Futures** - Proposed Wording for UK comments 335, 336, 337 and 338. N2888 = 09 – 0078.
- Detlef Vollmann, Howard Hinnant, and Anthony Williams **An Asynchronous Future Value (revised)** N2627 = 08 – 0137
- Howard E. Hinnant: **Multithreading API for C++0X - A Layered Approach**. N2094 = 06 – 0164. The original proposal for a complete threading package..

## 2.78 std::async()

`async()` 는 간단한 작업 실행 함수로, 아직 드래프트 표준에 투표되지 않는 이 FAQ 의 유일한 항목입니다. 아마 두 개의 살짝 다른 제안서들 간에 무엇을 채택할 지에 대한 투표가 10월 쯤에 이루어질 것 같습니다. 아래는 프로그래머가 복잡한 쓰레드-lock 을 사용한 병행 프로그래밍에서 어떻게 사용하는지 보여주고 있습니다.

---

```

template<class T, class V> struct Accum { // 단순한 함수 객체
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& v) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)
    // v 가 충분히 크다면 많은 수의 작업들을 생성한다
{
    if (v.size() < 10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

    return f0.get() + f1.get() + f2.get() + f3.get();
}

```

---

이는 매우 간단하게 동시성을 사용하는 방법으로, 다만 `explicit` 쓰레드, `lock`, 베퍼들은 사용하지 않습니다. `f` 변수들의 타입은 표준 라이브러리 함수 `async()`의 리턴타입으로 결정되며, 이는 `future`입니다. 만일 필요에 따라 `future`의 `get()`은 쓰레드가 종료될 때 까지 기다립니다. 여기서 `async()` 가 하는 일은 필요한 만큼 쓰레드를 소환하는 것이고, `future`의 역할은 각각의 쓰레드들을 알맞게 `join()` 해주는 것입니다. `async()/future` 디자인 철학의 가장 중요한 것은 ‘단순함’입니다; `future`은 일반적으로 쓰레드와 함께 사용될 수 있는데, `async()`를 이용해서 I/O, `mutex` 조작, 혹은 다른 작업들과 상호작용하는 것들을 수행하는 일은 생각조차 하지마세요. `async()` 구현에 중요한 철학은 `range-for` 문의 철학과 매우 유사합니다: 간단하고 흔히 사용되는 것들을 최대한 간단하게 처리하고, 복잡한 문제들은 완벽히 일반화된 메커니즘을 통해 처리하는 것입니다. `async()`는 어떠한 쓰레드 안에서도 새로운 쓰레드를 호출하도록 요청될 수 있거나, `async()` 가 올바른 것이라 판단되는 다른 쓰레드 안에서 실행될 수 있습니다. 후자가 좀더 유저 입장에서 간단하고, 좀더 효율적일 가능성이 높습니다. (간단한 작업들에 대해서만 말이지요)

#### 참고 자료

- Standard: ???
- Lawrence Crowl: [An Asynchronous Call for C++](#). N2889 = 09 – 0079.
- Herb Sutter : [A simple async\(\)](#) N2901 = 09 – 0091

## 2.79 난수(random number) 생성

랜덤 난수는 게임, 시뮬레이션, 테스트, 보안등의 여러 분야에서 매우 유용하게 사용됩니다. 이러한 넓은 분야에서 사용되는 만큼 표준 라이브러리에서 난수 생성을 제공할 수 있는 것이 중요하게 되었습니다. 난수 생성은 2개의 부분으로 구성되는데, 하나는 의사 난수(pseudo-random)들의 수열을 생성하는 것이고, 다른 하나는 이러한 값들을 범위 안으로 분포시키는 것입니다. 이러한 분포들의 예로, `uniform_int_distribution` (모든 정수들을 동일하게 생성됨)과 `normal_distribution` ('벨 커브') 가 있습니다. 예를 들어

---

```
uniform_int_distribution<int> one_to_six {1,6}; // 1 부터 6 까지의 분포를 생성
default_random_engine re {}; // 디폴트 엔진
```

---

난수를 얻어내기 위해 여러분은 엔진을 통해 분포할 수 있습니다.

---

```
int x = one_to_six(re); // x 는 [1:6] 안의 값이 됩니다
```

---

각각의 호출에서 엔진을 전달하는 것은 불편한 일이므로, 우리는 인자를 `bind` 해서, 함수 객체를 이용해 인자 없이 사용할 수 있게 됩니다.

---

```
auto dice {bind(one_to_six,re)}; // 난수 생성기를 만든다
```

---



---

```
int x = dice(); // 주사위 굴리기: x 는 [1:6] 안의 값이 됩니다
```

---

한 전문가는 이 표준 라이브러리의 우수한 일반성과 성능에 대해, “모든 난수 라이브러리가 갖춰야 할 것들” 이라고 평했습니다. 하지만, 초보자들이 쉽게 사용할 수 있지 않아보이기도 합니다. 저는 한번도

난수 인터페이스가 성능에서의 병목현상을 발생시키리라 생각한적이 한번도 없지만, 매우 간단한 난수 생성기를 필요로 하지 않는 초보자들을 가르친 적도 한번도 없습니다. 초보자용 코드는 아래 코드로 충분할 것입니다.

---

```
int rand_int(int low, int high); // [low:high] 안의 균등 분포 난수를 생성한다
```

---

그렇다면 위를 어떻게 얻을까요? 우리는 `rand_int()` 안에 `dice()` 와 비슷한 것을 넣으면 됩니다.

---

```
int rand_int(int low, int high)
{
    static default_random_engine re {};
    using Dist = uniform_int_distribution<int>;
    static Dist uid {};
    return uid(re, Dist::param_type{low,high});
}
```

---

위 정의 역시 ‘전문가 수준’ 이겠지만, `rand_int()` 를 사용하는 것은 C++ 수업의 첫째주에도 도입할 수 있습니다. 아래는 정규 분포를 생성하는 프로그램 입니다.

---

```
default_random_engine re; // 디폴트
normal_distribution<int> nd(31 /* 평균 */, 8 /* 분산 */);

auto norm = std::bind(nd, re);

vector<int> mn(64);

int main()
{
    for (int i = 0; i < 1200; ++i) ++mn[round(norm())]; // 생성한다

    for (int i = 0; i < mn.size(); ++i) {
        cout << i << '\t';
        for (int j = 0; j < mn[i]; ++j) cout << '*';
        cout << '\n';
    }
}
```

---

그 결과는 아래와 같습니다.

---

```
0
1
2
3
4 *
5
```

```
6
7
8
9 *
10 ***
11 ***
12 ***
13 ****
14 *****
15 ****
16 ******
17 ******
18 *********
19 *********
20 *********
21 *********
22 *********
23 *********
24 *********
25 *********
26 *********
27 *********
28 *********
29 *********
30 *********
31 *********
32 *********
33 *********
34 *********
35 *********
36 *********
37 *********
38 *********
39 *********
40 *********
41 *********
42 *********
43 *********
44 *********
45 *********
46 *********
47 *********
```

```

48 *****
49 *****
50 *****
51 ***
52 ***
53 **
54 *
55 *
56
57 *
58
59
60
61
62
63

```

---

### 참고 자료

- Standard 26.5: Random number generation

## 2.80 Concepts

경고 : “concepts” 는 C++ 11 에 포함되어 있지 않고 많은 부분 다시 디자인 되고 있습니다. “concepts” 는 타입, 타입의 조합, 그리고 정수와 타입의 조합들을 요구하는데 사용하는 메커니즘입니다. 이는 특히 템플릿들의 타입 체크를 수행하는데 매우 유용하게 사용됩니다. 역으로 보면, 이는 템플릿 몸체에서 오류를 미리 확인하는데 도움이 됩니다. 아래와 같은 표준 라이브러리 알고리즘 `fill` 을 살펴봅시다.

---

```

template<ForwardIterator Iter, class V> //타입들의 종류들
    requires Assignable<Iter::value_type,V> // 인자 타입들의 관계
void fill(Iter first, Iter last, const V& v); // 정의가 아니라 선언이다

```

```

fill(0, 9, 9.9); // Iter 은 int 이다; 오류; int 는 ForwardIterator 가 아니다
                  // int 변수 앞에 * 를 붙일 수 없습니다
fill(&v[0], &v[9], 9.9); // Iter 는 int 이다; 가능: int* 는 ForwardIterator 이다

```

---

여기서 중요한 점은 우리는 오직 `fill()` 만을 선언하였다는 점입니다. 우리는 이를 정의하지 않았습니다. 반면에 우리는 명시적으로 `fill()` 이 다음과 같은 인자가 필요하다는 사실을 나타냈습니다 :

- `first` 와 `last` 인자의 타입은 `ForwardIterator` 이어야 한다. (그리고 같은 타입)
- 세 번째 인자인 `v` 는 반드시 `ForwardIterator` 의 `value_type` 으로 대입될 수 있어야만 한다.

우리는 당연히도, 표준을 읽어보았기에 이미 위 내용들을 알고 있습니다. 하지만 컴파일러들은 알 수 없기 때문에 코드에 concept ForwardIterator 와 Assignable 을 넣어서 알려주어야만 합니다. fill() 을 사용함으로써 발생하는 오류는 즉시 확인할 수 있으며 발생하는 오류 메세지의 의미도 쉽게 이해할 수 있습니다. 컴파일러는 이제 프로그래머들의 의도에 대한 정보를 알 수 있고, 이를 통해 알맞은 메세지를 보내줄 수 있게 됩니다. concept 은 또한 템플릿 제작자들에게도 도움이 됩니다.

---

```
template<ForwardIterator Iter, class V>
    requires Assignable<Iter::value_type, V>
void fill(Iter first, Iter last, const V& v)
{
    while (first!=last) {
        *first = v;
        first=first+1; // 오류 : + 는 Forward_iterator 에 정의되지 않았다.
                        // (++first) 를 사용해라
    }
}
```

---

이 오류는 즉시 발견되므로, 지루한 테스트를 할 필요가 없게 됩니다. 다른 종류들의 타입을 구별하고, 분류할 수 있기 때문에 우리는 전달되는 타입에 따라 오버로드를 할 수 있습니다. 예를 들어

```
// concept 을 이용한 반복자 기반의 표준 정렬 :
template<Random_access_iterator Iter>
    requires Comparable<Iter::value_type>
void sort(Iter first, Iter last); // 일반적인 방법으로 구현한다.

// 컨테이너 기반의 정렬
template<Container Cont>
    requires Comparable<Cont::value_type>
void sort(Cont& c)
{
    sort(c.begin(),c.end()); // 단순히 반복자 버전을 호출하면 된다
}

void f(vector<int>& v)
{
    sort(v.begin(), v.end()); // 한가지 방법
    sort(v); // 다른 방법
    // ...
}
```

---

여러분은 자신만의 concept 를 정의할 수 있지만, 초보자를 위해서 표준 라이브러리들은 여러가지 유용한 concept 들, 예를 들어 ForwardIterator, Callable, LessThanComparable, Regular 를 제공합니다.

참고로 C++ 0x 표준 라이브러리는 concept 들을 이용해서 정의되었습니다.

### 참고 자료

- the C++ draft 14.10 Concepts
- [N2617 = 08 – 0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: [Proposed Wording for Concepts \(Revision 5\) \(Final proposal\)](#).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: Concepts: [Linguistic Support for Generic Programming in C++](#). OOPSLA'06, October 2006.

## 2.81 concept 맵(map)

`int*` 는 `ForwardIterator` 입니다; 우리는 `concept` 를 이야기 할 때 그렇다고 이야기 해왔고, 표준 역시 그렇다고 해왔고, 심지어 STL 의 첫번째 버전 역시 포인터들을 반복자로 사용해왔습니다. 하지만 우리는 또한 `ForwardIterator` 의 `value_type` 에 대해서도 이야기 하였습니다. 하지만 `int*` 는 `value_type`이라는 멤버를 가지고 있지는 않습니다; 사실, 멤버를 아예 가지고 있지 않습니다. 그렇다면 `int*` 가 어떻게 `ForwardIterator` 가 될 수 있을까요? 이것이 가능한 것은, 우리가 그렇게 정했기 때문입니다. `concept_map` 을 이용해서, 우리는 `T*` 이 `ForwardIterator` 로 사용될 때 `value_type` 을 `T` 로 사용되게 만들 수 있습니다.

---

```
template<Value_type T>
concept_map ForwardIterator<T*> { // T* 의 value_type 은 T
    typedef T value_type;
};
```

---

`concept_map` 은 우리로 하여금 타입이 어떻게 보여지는지 조정할 수 있기 때문에, 타입을 수정하거나 다른 타입으로 감쌀(wrap) 필요가 없게 됩니다. “concept 맵핑”은 널리 사용되는 독립적으로 개발된 소프트웨어에서 사용되기에 매우 유통성있고, 일반화된 메커니즘이입니다.

### 참고 자료

- the C++ draft 14.10.2 Concept maps
- [N2617 = 08 – 0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: [Proposed Wording for Concepts \(Revision 5\) \(Final proposal\)](#).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine: Concepts: [Linguistic Support for Generic Programming in C++](#). OOPSLA'06, October 2006.

## 2.82 Axioms

`axiom` 은 `concept` 을 설명하기 위한 서술자들의 집합이라 볼 수 있습니다. `axiom` 을 사용하는 주된 목적으로, 외부 툴들, 예를 들어 도메인 특이적인 최적화 을 위한 것들입니다. 두 번째 사용 예로, 간단히 표준의

정의들(specification)을 단순하고 정확하게 표현하기 위함입니다 (실제로 표준 라이브러리 specification의 많은 부분 사용됩니다). `axiom` 들을 특히 일부 최적화 과정에 유용한데, 다만 컴파일러들이 사용자 정의 `axiom` 들의 것들을 사용하도록 의무화 되어 있는 것은 아닙니다; 이들은 그냥 표준에 정의되어 있는 것들만 기반으로 하지요. `axiom` 은 동일한 것으로 간주될 수 있는 연산들의 짝을 나열합니다. 예를 들어

---

```
concept Semigroup<typename Op, typename T> : CopyConstructible<T> {
    T operator()(Op, T, T);
    axiom Associativity(Op op, T x, T y, T z) {
        op(x, op(y, z)) <=> op(op(x, y), z); // T 의 연산자가 결합법칙이 성립한다고 가정됨
    }
}
// monoid 은 항등원소를 가진 세미그룹이다.
concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
    T identity_element(Op);
    axiom Identity(Op op, T x) {
        op(x, identity_element(op)) <=> x;
        op(identity_element(op), x) <=> x;
    }
}
```

---

`<=>` 은 항등 연산자로 `axiom` 에서만 유일하게 사용됩니다. 참고로 여러분은 (일반적인 경우) `axiom` 을 증명할 수 없습니다; 우리는 `axiom` 들을 우리가 증명할 수 없지만, 프로그래머가 인정할 만한 가정으로 생각할 만한 것들을 나타내기 위해 사용합니다. 항등 문장의 양변으로 어떤 값들은 사용할 수 없습니다. 예를 들어 부동 소수점의 NaN (not a number); 만일 항등 문장의 양변으로 NaN 을 사용한다면, (명백하게도) 참과 거짓 둘다 되기 때문입니다. 하지만 한쪽에서만 NaN 을 사용한다면 `axiom` 을 유용하게 이용할 수 있습니다. `axiom` 은 항등 문장(`<=>` 를 사용한)과 조건 문장들의 나열이라 볼 수 있습니다.

---

```
// concept TotalOrder 의 경우
axiom Transitivity(Op op, T x, T y, T z)
{
    if (op(x, y) && op(y, z)) op(x, z) <=> true; // 조건부적 항등
}
```

---

- the C++ draft 14.9.1.4 Axioms