

효과적인 현대 C++

++

C++11 및 C++14 사용을 개선하는 42가지 구체적인 방법

스콧 마이어스

효과적인 현대 C++

C++11 및 C++14에 익숙해지는 것은 그들이 도입하는 기능(예: 자동 유형 선언, 이동 의미론, 람다 식 및 동시성 지원)에 익숙해지는 문제 그 이상입니다. 문제는 이러한 기능을 효과적으로 사용하여 소프트웨어가 정확하고 효율적이며 유지 관리 가능하고 이식 가능하도록 하는 방법을 배우는 것입니다. 그것이 바로 이 실용적인 책이 나오는 이유입니다. 이 책은 C++11 및 C++14, 즉 최신 C++를 사용하여 진정으로 훌륭한 소프트웨어를 작성하는 방법을 설명합니다.

주제는 다음과 같습니다.

- 중괄호 초기화의 장단점, noexcept
스펙, 퍼펙트 포워딩, 스마트 포인터 메이크
기능
- std::move, std::forward, rvalue 간의 관계
참조 및 범용 참조
- 명확하고 정확하며 효과적인 람다식 작성 기법
- std::atomic 이 volatile과 어떻게 다른지, 각각 어떻게 되어야 하는지
사용 및 C++의 동시성 API와 관련되는 방식
- "오래된" C++ 프로그래밍(예: C++98)의 모범 사례가 최신 C++에서 소프트
웨어 개발을 위해 수정해야 하는 방법

Effective Modern C++는 Scott Meyers의 초기 책에서 입증된 지침 기반의 예제 중심 형식을 따르지만 완전히 새로운 자료를 다룹니다. 현대 C++ 소프트웨어 개발자라면 누구나 읽어야 할 필독서입니다.

20년 이상 동안 Scott Meyers의 Effective C++ 책(Effective C++, More Effective C++ 및 Effective STL)은 C++ 프로그래밍 지침의 기준을 설정했습니다.

복잡한 기술 자료에 대한 명확하고 매력적인 설명은 그를 전 세계적으로 추종하게 하여 트레이너, 컨설턴트 및 컨퍼런스 발표자로서의 수요를 유지하고 있습니다. 그는 박사 학위를 가지고 있습니다. 브라운 대학교에서 컴퓨터 공학 박사.

[“C++을 배운 후”](#)

[기본, 그때 배운](#)

[에서 C++를 사용하는 방법](#)

[에서 생산 코드](#)

[マイアース 시리즈](#)

[효과적인 C++ 책](#)

[효과적인 현대 C++](#)

[가장 중요하다](#)

[조언을 위한 방법 예약](#)

[주요 지침에 대해,](#)

[스타일 및 사용 관용구](#)

[현대 C++ 효과적으로](#)

[그리고 잘. 그것을 소유하지 마십시오](#)

[아직? 이거 사세요. 지금.”](#)

—허브 셔터

ISO C++ 표준 위원회 의장 및

Microsoft의 C++ 소프트웨어 아키텍트

프로그래밍/C++

US \$49.99 캔 \$52.99

ISBN: 978-1-491-90399-5



9 781491 903995



트위터: @oreillymedia
facebook.com/oreilly

효과적인 현대 C++에 대한 찬사

그렇다면 여전히 C++에 관심이 있으십니까? 당신은해야한다! 최신 C++(즉, C++11/C++14)는 단순한 페이스리프트 그 이상입니다. 새로운 기능을 고려하면 더 많은 재창조인 것 같습니다. 지침과 도움을 찾고 계십니까? 그렇다면 이 책은 분명히 당신이 찾고 있는 것입니다. C++와 관련하여 Scott Meyers는 정확성, 품질 및 기쁨의 동의어였으며 지금도 마찬가지입니다.

—게르하르트 크로이저

연구 개발 엔지니어, Siemens AG

최고의 전문 지식을 찾는 것은 충분히 어렵습니다. 설명을 전략화하고 간소화하는 것에 대한 저자의 집착인 교수 완벽주의를 찾는 것 또한 어렵습니다.

당신은 당신이 같은 사람에게 구현된 두 가지를 발견할 때 당신이 대접을 받고 있다는 것을 압니다.

효과적인 모던 C++는 유능한 기술 작가의 우뚝 솟은 성과입니다.

복잡하고 상호 연결된 주제 위에 명료하고 의미 있고 순서가 잘 정리된 설명을 모두 선명한 문학적 스타일로 겹칩니다.

Effective Modern C++에서는 기술적 실수, 둔한 순간 또는 게으른 문장을 찾을 가능성성이 거의 없습니다.

—Andrei Alexandrescu

Ph.D., 연구 과학자, Facebook, Modern C++ Design 저자

20년 이상의 C++ 경험이 있는 사람으로서 최신 C++(모범 사례와 피해야 할 함정 모두)를 최대한 활용하려면 이 책을 구입하여 철저히 읽고 자주 참조하는 것이 좋습니다!

나는 확실히 그것을 통해 새로운 것을 배웠습니다!

—네빈 리버

수석 소프트웨어 엔지니어, DRW Trading Group

C++의 창시자인 Bjarne Stroustrup은 "C++11은 새로운 언어처럼 느껴진다"고 말했습니다.

Effective Modern C++는 일상적인 프로그래머가 C++11 및 C++14의 새로운 기능과 관용구를 어떻게 활용할 수 있는지 명확하게 설명함으로써 이와 같은 느낌을 공유할 수 있도록 합니다. 또 다른 위대한 Scott Meyers 책.

—카시오 네리

FX 정량 분석가, Lloyds Banking Group

Scott은 기술적 복잡성을 이해할 수 있는 커널까지 끓이는 재주가 있습니다.
그의 Effective C++ 책은 이전 세대 C++ 프로그래머의 코딩 스타일을 높이는 데 도움이 되었습니다.
새 책은 현대 C++를 사용하는 사람들에게도 동일한 작업을 수행할 수 있는 위치에 있는 것 같습니다.

—Roger Orr
OR/2 Limited, ISO C++ 표준 위원회 위원

효과적인 Modern C++는 최신 C++ 기술을 향상시키는 훌륭한 도구입니다. 최신 C++를 어떻게, 언제, 어디서 사용하고 효과적인지 가르쳐줄 뿐만 아니라 그 이유도 설명합니다.

의심할 여지 없이 Scott의 명확하고 통찰력 있는 글은 42개의 잘 생각한 항목에 걸쳐 펼쳐져 있으므로 프로그래머가 언어를 훨씬 더 잘 이해할 수 있습니다.

—Bart Vandewoestyne 연
구 개발 엔지니어 및 C++ 애호가

저는 C++를 사랑하며 수십 년 동안 제 작업 수단이었습니다. 그리고 최신 기능으로 이전에 상상했던 것보다 훨씬 강력하고 표현력이 뛰어납니다. 그러나 이 모든 선택과 함께 "이 기능을 언제, 어떻게 적용합니까?"라는 질문이 나옵니다. 항상 그래왔듯이 Scott의 Effective C++ 책은 이 질문에 대한 확실한 답입니다.

—데미안 앳킨스
전산 소프트웨어 엔지니어링 팀장, CSIRO

최신 C++로의 전환을 위한 훌륭한 읽기 - 새로운 C++11/14 언어 기능은 C++98과 함께 설명되고 주제 항목은 참조하기 쉽고 조언은 각 섹션 끝에 요약되어 있습니다.

캐주얼 및 고급 C++ 개발자 모두에게 재미있고 유용합니다.

—레이첼 청 F5 네트워크

C++98/03에서 C++11/14로 마이그레이션하는 경우 Scott이 Effective Modern C++에서 제공하는 매우 실용적이고 명확한 정보가 필요합니다. 이미 C++11 코드를 작성하고 있다면 Scott의 언어의 중요한 새 기능에 대한 철저한 토론을 통해 새 기능의 문제를 발견할 수 있습니다. 어느 쪽이든, 이 책은 당신의 시간을 할애할 가치가 있습니다.

—롭 스튜어트 부
스트 운영 위원회 위원(boost.org)

효과적인 현대 C++

스콧 마이어스

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Scott Meyers의 효과

적인 현대 C++

저작권 © 2015 스콧 마이어스. 판권 소유.

캐나다에서 인쇄됩니다.

발행: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly 책은 교육, 비즈니스 또는 판촉용으로 구입할 수 있습니다. 온라인 판은 대부분의 책에서도 사용할 수 있습니다 (<http://safaribooksonline.com>). 자세한 내용은 기업/기관 영업 부서(800-998-9938 또는 Corporate@oreilly.com)로 문의하십시오.

편집자: 레이첼 루멜리오티스

교정자: Charles Roumeliotis

프로덕션 편집자: Melanie Yarbrough

인덱서: Scott Meyers

복사: Jasmine Kwityn

인테리어 디자이너: 데이비드 후타토

표지 디자이너: Ellie Volkhausen

일러스트레이터: 레베카 데마레스트

2014년 11월: 초판

초판의 개정 내역

2014-11-07: 첫 번째 릴리스

<http://oreilly.com/catalog/errata.csp?isbn=9781491903995> 참조 릴리스 세부 사항을 위해.

O'Reilly 로고는 O'Reilly Media, Inc.의 등록 상표입니다. Effective Modern C++, Rose-crowned Fruit Dove의 표지 이미지 및 관련 트레이드 드레스는 O'Reilly Media, Inc.의 상표입니다.

발행인과 저자는 이 저작물에 포함된 정보와 지침이 정확한지 확인하기 위해 선의의 노력을 기울였지만, 발행인과 저자는 오류 또는 누락에 대한 모든 책임을 지지 않으며, 여기에는 이 작업에 대한 의존. 이 작업에 포함된 정보 및 지침의 사용에 따른 위험은 사용자가 감수해야 합니다. 이 작업에 포함되거나 설명된 코드 샘플 또는 기타 기술이 오픈 소스 라이선스 또는 타인의 지적 재산권의 적용을 받는 경우, 해당 사용이 해당 라이선스 및/또는 권리를 준수하는지 확인하는 것은 귀하의 책임입니다.

978-1-491-90399-5

[EJ]

달라에게,
블랙 래브라도 리트리버

목차

게시자에서.....	xii
감사합니다.....	xiii
소개.....	1
 1. 유형 추론하기.....	9
항목 1: 템플릿 유형 추론을 이해하십시오.	9
항목 2: 자동 유형 추론을 이해하십시오.	18
항목 3: decltype을 이해하십시오.	23
항목 4: 추론된 유형을 보는 방법을 알고 있습니다.	30
 2. 자동.....	37
항목 5: 명시적 형식 선언보다 auto를 선호합니다.	37
항목 6: 자동 추론할 때 명시적으로 형식화된 초기화 관용구를 사용하세요.	
원하지 않는 유형.	43
 3. 최신 C++로 이동.....	49
항목 7: 객체를 생성할 때 ()와 {}를 구별하세요.	49
항목 8: 0과 NULL보다 nullptr을 선호하십시오.	58
항목 9: typedef보다 별칭 선언을 선호합니다.	63
항목 10: 범위가 지정되지 않은 열거형보다 범위가 지정된 열거형을 선호합니다.	67
항목 11: 정의되지 않은 비공개 함수보다 삭제된 함수를 선호합니다.	74
항목 12: 재정의하는 함수 재정의를 선언하십시오.	79
항목 13: 반복자보다 const_iterator를 선호하십시오.	86
항목 14: 예외를 내보내지 않을 경우 함수 noexcept를 선언하십시오.	90
항목 15: 가능하면 constexpr을 사용하세요.	97

항목 16: const 멤버 함수를 스레드로부터 안전하게 보호하세요.	103
항목 17: 특수 멤버 함수 생성을 이해하십시오.	109
4. 스마트 포인터.....	117
아이템 18: 베타적 소유권 리소스에 std::unique_ptr를 사용하라 관리.	118
아이템 19: 공유 소유권 리소스에 std::shared_ptr를 사용하라 관리.	125
항목 20: 다음을 수행할 수 있는 std::shared_ptr과 유사한 포인터에 std::weak_ptr를 사용하십시오. 매달다.	134
항목 21: 직접 사용하는 것보다 std::make_unique 및 std::make_shared를 선호하십시오 새로운.	139
항목 22: Pimpl Idiom을 사용할 때 특수 멤버 함수를 정의하십시오. 구현 파일.	147
5. Rvalue 참조, 의미 이동 및 완전 전달.....	157
항목 23: std::move와 std::forward를 이해하라.	158
항목 24: 범용 참조를 rvalue 참조와 구별하십시오.	164
항목 25: rvalue 참조에는 std::move를 사용하고 유니버설에는 std::forward를 사용하세요. 참조.	168
항목 26: 범용 참조에 대한 오버로딩을 피하십시오.	177
항목 27: 유니버설 오버로딩에 대한 대안을 숙지하라 참조.	184
항목 28: 참조 축소를 이해하십시오.	197
항목 29: 이동 작업이 존재하지 않고 저렴하지도 않고 그렇지도 않다고 가정합니다. 사용된.	203
항목 30: 완벽한 전달 실패 사례를 숙지하십시오.	207
6. 람다 표현식.....	215
항목 31: 기본 캡처 모드를 피하세요. 216	224
항목 32: 객체를 클로저로 이동하려면 초기화 캡처를 사용하세요.	229
항목 33: auto&& 매개변수에 decltype을 사용하여 std::forward를 전달하십시오.	232
항목 34: std::bind보다 람다를 선호하십시오.	241
7. 동시성 API.....	241
항목 35: 스레드 기반보다 작업 기반 프로그래밍을 선호하십시오.	241
항목 36: 비동기성이 필수적인 경우 std::launch::async를 지정하십시오.	245
항목 37: 모든 경로에서 std::threads를 결합할 수 없도록 만드세요.	250
항목 38: 다양한 스레드 핸들 소멸자 동작에 주의하십시오.	258
항목 39: 일회성 이벤트 커뮤니케이션을 위해 무효 선물을 고려하십시오.	262

항목 40: 동시성을 위해서는 std::atomic을 사용하고 특수 메모리에는 volatile을 사용하세요.	271
8. 조정.....	281
항목 41: 값이 저렴한 복사 가능한 매개변수에 대한 전달을 고려하십시오.	
이동하고 항상 복사합니다.	281
항목 42: 삽입 대신 배치를 고려하십시오.	292
색인.....	303

게시자로부터

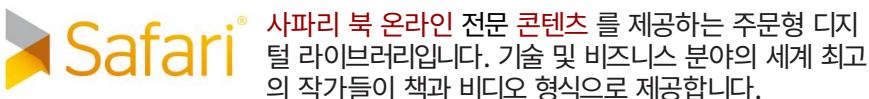
코드 예제 사용

이 책은 당신이 일을 끝내는 데 도움을 주기 위해 여기에 있습니다. 일반적으로 이 책과 함께 예제 코드가 제공되면 프로그램과 문서에서 사용할 수 있습니다. 코드의 상당 부분을 복제하지 않는 한 허가를 받기 위해 당사에 연락할 필요가 없습니다. 예를 들어, 이 책의 여러 코드 덩어리를 사용하는 프로그램을 작성하는 데는 권한이 필요하지 않습니다. O'Reilly 책의 예제 CD-ROM을 판매하거나 배포하려면 허가가 필요합니다. 이 책을 인용하고 예제 코드를 인용하여 질문에 답하는 것은 허가가 필요하지 않습니다. 이 책의 상당한 양의 예제 코드를 제품 설명서에 통합하려면 허가가 필요합니다.

우리는 저작자 표시를 높이 평가하지만 요구하지는 않습니다. 저작자 표시에는 일반적으로 제목, 저자, 발행인 및 ISBN이 포함됩니다. 예: “Scott Meyers(O'Reilly)의 Effective Modern C++. Copyright 2015 Scott Meyers, 978-1-491-90399-5.”

코드 예제의 사용이 공정 사용 또는 위에 제공된 권한에 해당하지 않는다고 생각되면 언제든지 permissions@oreilly.com으로 문의하십시오.

Safari® 온라인 도서



기술 전문가, 소프트웨어 개발자, 웹 디자이너, 비즈니스 및 크리에이티브 전문가는 Safari Books Online을 연구, 문제 해결, 학습 및 인증 교육을 위한 기본 리소스로 사용합니다.

Safari Books Online은 다양한 [계획과 가격](#)을 제공합니다 [기업용](#), [정부](#), [교육](#), 그리고 개인.

회원은 O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco와 같은 출판사의 완전 검색 가능한 하나의 데이터베이스에서 수천 권의 책, 교육 비디오 및 출판 전 원고에 액세스할 수 있습니다. Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology 및 기타 수백 가지. Safari Books Online에 대한 자세한 내용은 온라인을 참조하십시오.

문의 방법

이 책에 대한 의견과 질문은 다음 출판사에 문의할 수 있습니다.

오라일리 미디어, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472 800-998-9938(미국 또는 캐나다) 707-829-0515(국제 또는 지역)
707-829-0104(팩스)

이 책에 대해 논평을 하거나 기술적인 질문을 하려면 bookquestions@oreilly.com으로 이메일을 보내십시오.

책, 코스, 컨퍼런스 및 뉴스에 대한 자세한 내용은 웹사이트 <http://www.oreilly.com>을 참조하십시오.

페이스북에서 저희를 찾으세요: <http://facebook.com/oreilly>

트위터에서 팔로우하세요: <http://twitter.com/oreillymedia>

YouTube에서 우리를 시청하십시오: <http://www.youtube.com/oreillymedia>

감사의 말

저는 2009년에 당시 C++0x(초기 C++11)로 알려진 것을 조사하기 시작했습니다. Usenet 뉴스 그룹 comp.std.c++에 수많은 질문을 게시했으며 해당 커뮤니티의 구성원들에게 감사합니다. (특히 Daniel Krügler) 매우 유용한 게시물에 감사드립니다. 최근 몇 년 동안 저는 [스택 오버플](#)로로 눈을 돌렸습니다. C++11 및 C++14에 대한 질문이 있었고 현대 C++의 세부 사항을 이해하는 데 도움을 준 커뮤니티에도 똑같이 감사합니다.

2010년에 나는 C++0x에 대한 교육 과정을 위한 자료를 준비했습니다(궁극적으로 [New C++ 개요](#), Artima 출판, 2010). 이러한 자료와 나의 지식은 Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober 및 Anthony Williams가 수행한 기술적 검토로부터 큰 도움을 받았습니다. 그들의 도움이 없었다면 나는 Effective Modern C++를 수행할 수 있는 위치에 있지 못했을 것입니다. 부수적으로 이 제목은 2014년 2월 18일 내 블로그 게시물인 "[내 책 이름 지정을 도와주세요](#)"에 응답한 여러 독자 와 Andrei Alexandrescu([Modern C++ Design](#)의 저자, Addison-Wesley, 2001)는 그의 용어에 대한 밀접이 아닌 타이틀을 축복할 만큼 친절했습니다.

이 책에 있는 모든 정보의 출처를 확인할 수는 없지만 일부 출처는 비교적 직접적인 영향을 미쳤습니다. 항목 4의 정의되지 않은 템플릿을 사용하여 컴파일러에서 유형 정보를 추출하는 방법은 Stephan T. Lavavej가 제안했으며 Matt P. Dziubinski는 Boost.TypeIndex를 주목했습니다. 항목 5에서 서명되지 않은 std::vector<int>::size_type 예제는 Andrey Karpov의 2010년 2월 28일 기사 "[C++0x 표준이 64비트 오류를 제거하는 데 어떤 도움을 줄 수 있습니까?](#)"에서 가져온 것입니다. 동일한 항목의 std::pair<std::string, int>/std::pair<const std::string, int> 예제는 Going Native 2012에서 Stephan T. Lavavej의 강연, "[STL11: Magic && Secrets .](#)" 항목 6은 Herb Sutter의 2013년 8월 12일 기사 "[GotW #94 솔루션: AAA 스타일\(거의 항상 자동\)](#)"에서 영감을 받았습니다. 항목 9는 Martinho Fernandes의 2012년 5월 27일 블로그 게시물 "[부속 이름 처리](#)"에서 영감을 받았습니다. 참조 한정자에 대한 오버로딩을 보여주는 항목 12 예제는 "[참조에 대한 멤버 함수를 오버로드하는 사용 사례는 무엇입니까?](#)"라는 질문에 대한 Casey의 답변을 기반으로 합니다.

[예선?"](#) 2014년 1월 14일 Stack Overflow에 게시되었습니다. `constexpr` 기능에 대한 C++14의 확장된 지원에 대한 My [Item 15](#) 처리는 Rein Halbersma로부터 받은 정보를 통합합니다. 항목 16은 Herb Sutter의 C++ 및 Beyond 2012 프레젠테이션, "You don't know const and mutable"을 기반으로 합니다. 팩토리 함수가 `std::unique_ptr`를 반환하도록 하는 항목 18의 조언은 Herb Sutter의 2013년 5월 30일 기사 "[GotW# 90 Solution: Factory](#)"를 기반으로 합니다. 항목 19에서 `fastLoadWidget`은 Herb Sutter의 고ing 네이티브 2013 프레젠테이션, "My Favorite C++ 10-Liner"에서 파생되었습니다. 항목 22의 `std::unique_ptr` 및 불완전한 유형에 대한 나의 처리는 Herb Sutter의 2011년 11월 27일 기사 "[GotW #100: Compilation Firewalls](#)"를 참조합니다. Stack Overflow 질문에 대한 Howard Hinnant의 2011년 5월 22일 답변, "T의 전체 정의를 알기 위해 `std::unique_ptr<T>`가 필요한가요?" 항목 25의 매트릭스 추가 예제는 David Abrahams의 글을 기반으로 합니다. JoeArgonne의 2012년 12월 8일 2012년 11월 30일 블로그 게시물 "람다 이동 캡처의 또 다른 대안"은 C++11에서 초기화 캡처를 에뮬레이트하는 항목 32의 `std::bind` 기반 접근 방식의 출처였습니다. 항목 37의 `std::thread` 소멸자에서 암시적 분리 문제에 대한 설명은 Hans-J에서 가져왔습니다. Boehm의 2008년 12월 4일자 논문, "[N2802: 스레드 개체에 대한 detach-on-destruction](#)을 재고해달라는 요청"입니다. 항목 41은 원래 David Abrahams의 2009년 8월 15일 블로그 게시물 "속도를 원하십니까? 가치로 전달하십시오." 이동 전용 유형이 특별한 대우를 받을 가치가 있다는 생각은 Matthew Fioravante에 기인한 반면 할당 기반 복사의 분석은 Howard Hinnant의 의견에서 비롯되었습니다. 항목 42에서 Stephan T. Lavavej와 Howard Hinnant는 배치 및 삽입 기능의 상대적 성능 프로파일을 이해하는 데 도움을 주었고 Michael Winterberg는 배치가 리소스 누출로 이어질 수 있는 방법에 주의를 기울였습니다. (Michael은 Sean Parent's Looking Native 2013 프레젠테이션, "[C++ Seasoning](#)", 그의 스스로).

Michael은 또한 배치 함수가 직접 초기화를 사용하는 반면 삽입 함수는 복사 초기화를 사용하는 방법을 지적했습니다.

기술 서적의 초안을 검토하는 것은 까다롭고 시간이 많이 걸리며 완전히 중요한 작업이며 많은 사람들이 기꺼이 나를 대신해 줘서 다행입니다. Effective Modern C++의 전체 또는 부분 초안은 Cassio Neri, Nate Kohl, Gerhard Kreuzer, Leor Zolman, Bart Vandewoestyne, Stephan T. Lavavej, Nevin ":" Liber, Rachel Cheng, Rob Stewart, Bob Steagall, Damien이 공식적으로 검토했습니다. Watkins, Bradley E. Needham, Rainer Grimm, Fredrik Winkler, Jonathan Wakely, Herb Sutter, Andrei Alexandrescu, Eric Niebler, Thomas Becker, Roger Orr, Anthony Williams, Michael Winterberg, Benjamin Huchley, Tom Kirby-Green, Alexey A Nikitin, William Dealtry, Hubert Matthews 및 Tomasz Kamiński. 또한 O'Reilly의 [Early Release Ebook](#)을 통해 여러 독자들로부터 피드백을 받았고 [Safari Books Online](#)의 러프 컷, 내 블로그에 대한 의견 ([Aristaia의 보기](#)), 그리고 이메일. 저는 이 모든 분들께 감사드립니다. 이 책은 그들의 도움이 없었을 때보다 훨씬 낫습니다.

특히 Stephan T. Lavavej와 Rob Stewart에게 감사드립니다. 그의 매우 상세하고 포괄적인 발언으로 인해 그들이 거의

나처럼 이 책에 많은 시간을 할애했다. 원고를 검토하고 모든 코드 예제를 다시 확인해 준 Leor Zolman에게도 특별한 감사를 전합니다.

책의 디지털 버전에 대한 전용 리뷰는 Gerhard Kreuzer, Emyr Williams 및 Bradley E. Needham이 수행했습니다.

코드 표시의 줄 길이를 64자로 제한하기로 한 결정은 Michael Maher가 제공한 데이터를 기반으로 했습니다.

Ashley Morgan Williams는 Lake Oswego Pizzicato에서의 식사를 독특하게 즐겁게 만들었습니다. 남자 크기의 시저에 관해서는 그녀가 가장 좋아하는 여자입니다.

연극 작가인 아내 낸시 L.

Urbano는 사임, 분노, 시기적절한 이해와 지원의 칵테일과 함께 여러 달 동안의 산만한 대화를 다시 한 번 참았습니다.

같은 기간 동안 우리 강아지 Darla는 내가 컴퓨터 화면을 보며 보내는 시간을 즐면서 만족했지만, 키보드 너머에 생명이 있다는 것을 결코 잊지 않았습니다.

소개

당신이 경험 많은 C++ 프로그래머이고 나와 같은 사람이라면 처음에 C++11에 접근하면서 “네, 알겠습니다. C++입니다. 더 그렇습니다.” 그러나 더 많이 배우면 변경 사항의 범위에 놀랐습니다. 자동 선언, 범위 기반 for 루프, 람다 식 및 rvalue 참조는 새로운 동시성 기능에 대해 말할 것도 없이 C++의 얼굴을 변경합니다. 그리고 관용적 변화가 있습니다. 0 및 typedef는 out이고 nullptr 및 alias 선언은 in입니다.

이제 열거형의 범위를 지정해야 합니다. 스마트 포인터는 이제 내장 포인터보다 선호됩니다. 일반적으로 개체를 이동하는 것이 복사하는 것보다 낫습니다.

C++14는 말할 것도 없고 C++11에 대해서도 배울 것이 많습니다.

더 중요한 것은 새로운 기능을 효과적으로 사용하는 방법에 대해 배울 것이 많다는 것입니다. "현대적인" C++ 기능에 대한 기본 정보가 필요하면 리소스가 풍부하지만 기능을 사용하여 정확하고 효율적이며 유지 관리 가능하고 이식 가능한 소프트웨어를 만드는 방법에 대한 지침을 찾고 있다면 검색이 더 어렵습니다.

그것이 바로 이 책이 나온 이유입니다. 이 책은 C++11 및 C++14의 기능을 설명하는 것이 아니라 효과적인 응용 프로그램에 대해 설명합니다.

책의 정보는 항목이라는 지침으로 나뉩니다. 다양한 형태의 연역을 이해하고 싶으십니까? 또는 자동 선언을 사용할 때와 사용하지 않을 때를 알고 있습니까? const 멤버 함수가 스레드로부터 안전해야 하는 이유, std::unique_ptr을 사용하여 Pimpl Idiom을 구현하는 방법, 람다 식에서 기본 캡처 모드를 피해야 하는 이유 또는 std::atomic과 volatile의 차이점에 관심이 있습니까? 답은 모두 여기에 있습니다. 또한 플랫폼 독립적이고 표준을 준수하는 답변입니다. 포터블 C++에 대한 책입니다.

가이드라인에는 예외가 있기 때문에 이 책의 항목은 규칙이 아니라 가이드라인입니다. 각 항목의 가장 중요한 부분은 그것이 제공하는 조언이 아니라 조언 뒤에 있는 근거입니다. 그것을 읽고 나면 프로젝트의 상황이 항목 지침의 위반을 정당화하는지 여부를 결정할 수 있습니다. 진실

이 책의 목표는 무엇을 하거나 무엇을 피해야 하는지 알려주는 것이 아니라 C++11 및 C++14에서 작동하는 방식에 대한 더 깊은 이해를 전달하는 것입니다.

용어 및 규칙

우리가 서로를 확실히 이해하기 위해서는 아이러니하게도 "C++"로 시작하는 몇 가지 용어에 동의하는 것이 중요합니다. C++의 네 가지 공식 버전이 있으며, 각각 해당 ISO 표준이 채택된 연도의 이름을 따서 명명되었습니다: C++98, C++03, C++11 및 C++14. C++98과 C++03은 기술적인 부분만 다르므로 이 책에서는 둘 다 C++98이라고 부른다. C++11을 언급할 때 C++11과 C++14를 모두 의미합니다. C++14는 사실상 C++11의 상위 집합이기 때문입니다. 내가 C++14를 작성할 때 구체적으로 C++14를 의미합니다. 그리고 단순히 C++를 언급한다면 모든 언어 버전과 관련된 광범위한 진술을 하는 것입니다.

내가 사용하는 용어	내가 의미하는 언어 버전
C++	모두
C++98	C++98 및 C++03
C++11	C++11 및 C++14
C++14	C++14

결과적으로 나는 C++가 효율성을 중시하고(모든 버전에 해당), C++98은 동시성을 지원하지 않으며(C++98 및 C++03에만 해당), +11은 람다 식(C++11 및 C++14의 경우 true)을 지원하고 C++14는 일반화된 함수 반환 형식 추론을 제공합니다(C++14의 경우에만 true).

C++11의 가장 널리 퍼진 기능은 아마도 이동 의미론이며 이동 의미론의 기초는 rvalue인 표현식과 lvalue인 표현식을 구별하는 것입니다. rvalue는 이동 작업에 적합한 개체를 나타내지만 lvalue는 일반적으로 그렇지 않기 때문입니다. 개념적으로(실제로 항상 그런 것은 아니지만) rvalue는 함수에서 반환된 임시 객체에 해당하는 반면 lvalue는 이름으로 또는 포인터 또는 lvalue 참조를 따라 참조할 수 있는 객체에 해당합니다.

표현식이 lvalue인지 여부를 결정하는 데 유용한 경험적 방법은 해당 주소를 사용할 수 있는지 묻는 것입니다. 가능하다면 일반적으로 그렇습니다. 할 수 없으면 일반적으로 rvalue입니다. 이 휴리스틱의 좋은 기능은 표현식의 유형이 표현식이 lvalue인지 rvalue인지 여부와 무관하다는 것을 기억하는 데 도움이 된다는 것입니다. 즉, 유형 T가 주어지면 유형 T의 lvalue와 유형 T의 rvalue를 가질 수 있습니다. 매개변수 자체가 lvalue이기 때문에 rvalue 참조 유형의 매개변수를 다룰 때 이것을 기억하는 것이 특히 중요합니다.

클래스 위젯 { 공개:

```
    위젯(위젯&& rhs);
    ...
};

// rhs는 lvalue 이지만 // rvalue 참조 유형이 있습니다.
// rvalue 참조이더라도 lvalue입니다. (유사한 추론으로 모든 매개변수는 lvalue입니다.)
```

여기에서 위젯의 이동 생성자 내에서 rhs의 주소를 취하는 것은 완벽하게 유효하므로 rhs는 유형이 rvalue 참조이더라도 lvalue입니다. (유사한 추론으로 모든 매개변수는 lvalue입니다.)

이 코드 조각은 내가 일반적으로 따르는 몇 가지 규칙을 보여줍니다.

- 클래스 이름은 위젯입니다. 임의의 사용자 정의 유형을 참조하고 싶을 때마다 위젯을 사용합니다. 클래스의 구체적인 내용을 보여줘야 하는 경우가 아니면 Widget을 선언하지 않고 사용합니다.
- 매개변수 이름 rhs ("오른쪽")를 사용합니다. 이동 작업(즉, 이동 생성자와 이동 할당 연산자)과 복사 작업(즉, 복사 생성자와 복사 할당 연산자)에 대해 내가 선호하는 매개변수 이름입니다. 또한 이 항 연산자의 오른쪽 매개변수에도 사용합니다. `Matrix operator+(const Matrix& lhs, const Matrix& rhs);`

lhs 가 "left-hand side"를 의미 하는 것은 놀라운 일이 아닙니다 .

- 주의를 끌기 위해 코드의 일부 또는 주석의 일부에 특수 서식을 적용합니다. 위의 위젯 이동 생성자에서 rhs의 선언과 rhs가 lvalue임을 언급하는 주석 부분을 강조 표시했습니다.

강조 표시된 코드는 본질적으로 좋지도 나쁘지도 않습니다. 특별히 주의를 기울여야 하는 코드일 뿐입니다. • "..."를 사용하여 "다른 코드가 여기에 올 수 있음"을 나타냅니다. 이 좁은 줄임표는 C++11의 가변형 템플릿에 대한 소스 코드에서 사용되는 넓은 줄임표("...")와 다릅니다. 혼란스럽게 들리지만 그렇지 않습니다. 예를 들어:

```
template<typename... Ts> // C++입니다. void processVals(const Ts&...
params) // 소스 코드 { // 타원
```

...

// 이것은 "일부 // 코드가 여
기에 간다"를 의미합니다.

}

processVals의 선언은 템플릿에서 유형 매개변수를 선언할 때 typename을 사용한다는 것을 보여주지만 이는 단지 개인의 취향일 뿐입니다. 키워드 클래스도 마찬가지로 작동합니다. 코드 발췌를 보여주는 경우

C++ 표준에서 클래스를 사용하여 유형 매개변수를 선언합니다. 이것이 표준이 하는 일이기 때문입니다.

객체가 동일한 유형의 다른 객체로 초기화되면 이동 생성자를 통해 복사본이 생성된 경우에도 새 객체는 초기화 객체의 복사본이라고 합니다. 유감스럽게도 C++에는 복사 생성 복사본인 객체와 이동 생성 복사본인 객체를 구별하는 용어가 없습니다.

무효 someFunc(위젯 w);

// someFunc의 매개변수 w // 값으로 전달됨

위젯 너비;

// wid는 일부 위젯입니다.

someFunc(와이드);

// someFunc에 대한 이 호출에서 // w는 복사 생성을 통해 생성된 // wid의 복사본입니다.

someFunc(std::move(wid));

// SomeFunc에 대한 이 호출에서 // w는 이동 구성을 통해 // 생성된 wid의 복사본입니다.

rvalue의 복사본은 일반적으로 이동 구성되는 반면 lvalue의 복사본은 일반적으로 복사 구성됩니다. 즉, 개체가 다른 개체의 복사본이라는 것만 알고 있으면 복사본을 구성하는 데 얼마나 많은 비용이 들었는지 말할 수 없습니다. 예를 들어 위의 코드에서 rvalue 또는 lvalue가 someFunc에 전달되는지 여부를 모른 채 매개변수 w를 생성하는 것이 얼마나 비용이 많이 드는지 말할 수 있는 방법은 없습니다.

(위젯을 이동하고 복사하는 비용도 알아야 합니다.)

함수 호출에서 호출 사이트에서 전달된 표현식은 함수의 인수입니다.

인수는 함수의 매개변수를 초기화하는 데 사용됩니다. 위의 someFunc에 대한 첫 번째 호출에서 인수는 wid입니다. 두 번째 호출에서 인수는 std::move(wid)입니다. 두 호출 모두에서 매개변수는 w입니다. 매개변수는 lvalue이지만 초기화되는 인수는 rvalue 또는 lvalue일 수 있기 때문에 인수와 매개변수를 구분하는 것이 중요합니다. 이것은 완전 전달 과정에서 특히 관련이 있습니다. 즉, 함수에 전달된 인수가 원래 인수의 rvalueness 또는 lvalueness가 유지되도록 두 번째 함수에 전달됩니다. (완벽한 전달은 항목 30에서 자세히 설명합니다.)

잘 설계된 기능은 예외 안전합니다. 즉, 최소한 기본 예외 안전 보증(즉, 기본 보증)을 제공합니다. 이러한 함수는 호출자가 예외가 발생하더라도 프로그램 불변성이 그대로 유지되고(즉, 데이터 구조가 손상되지 않음) 리소스가 누출되지 않도록 합니다. 강력한 예외 안전 보장(즉, 강력한 보장)을 제공하는 함수는 예외가 발생하는 경우 호출 이전의 프로그램 상태가 유지되도록 호출자에게 보장합니다.

함수 객체를 언급할 때 일반적으로 `operator()` 멤버 함수를 지원하는 유형의 객체를 의미합니다. 즉, 함수처럼 작동하는 객체입니다.

때때로 나는 비멤버 함수 호출의 구문을 사용하여 호출할 수 있는 모든 것을 의미하기 위해 약간 더 일반적인 의미로 이 용어를 사용합니다(즉, "함수 이름(인수)"). 이 광범위한 정의는 `operator()`를 지원하는 객체뿐만 아니라 함수 및 C와 유사한 함수 포인터도 포함합니다. (좁은 정의는 C++98에서, 더 넓은 정의는 C++11에서 옵니다.) 멤버 함수 포인터를 추가하여 더 일반화하면 호출 가능한 객체로 알려진 것이 생성됩니다. 일반적으로 미세한 구분을 무시하고 함수 객체와 호출 가능한 객체를 일종의 함수 호출 구문을 사용하여 호출할 수 있는 C++의 것으로 생각할 수 있습니다.

세.

람다 식을 통해 생성된 함수 객체를 클로저라고 합니다. 람다 식과 그들이 만드는 클로저를 구별할 필요가 거의 없기 때문에 저는 종종 둘 다 람다라고 부릅니다. 마찬가지로, 나는 함수 템플릿(즉, 함수를 생성하는 템플릿)과 템플릿 함수(즉, 함수 템플릿에서 생성된 함수)를 거의 구별하지 않습니다. 클래스 템플릿 및 템플릿 클래스도 마찬가지입니다.

C++의 많은 것들이 선언되고 정의될 수 있습니다. 선언은 저장 위치 또는 구현 방법과 같은 세부 정보를 제공하지 않고 이름과 유형을 소개합니다.

```
외부 정수 x;                                // 객체 선언

클래스 위젯;                                // 클래스 선언

bool func(const 위젯& w);                   // 함수 선언

열거형 클래스 색상;                          // 범위가 지정된 열거형 선언 // ( 항목
                                                10 참조)
```

정의는 저장 위치 또는 구현 세부 정보를 제공합니다.

```
정수 x;                                      // 객체 정의

클래스 위젯 {
    ...
};

bool func(const Widget& w)
{ return w.size() < 10; }                      // 함수 정의

열거형 클래스
{ 노란색, 빨간색, 파란색 };                  // 범위가 지정된 열거형 정의
```

정의는 또한 선언의 자격을 갖기 때문에 무언가가 정의라는 것이 정말로 중요한 경우가 아니면 저는 선언을 참조하는 경향이 있습니다.

매개변수 및 반환 유형을 지정하는 선언의 일부로 함수의 서명을 정의합니다. 함수 및 매개변수 이름은 서명의 일부가 아닙니다. 위의 예에서 func의 서명은 `bool(const Widget&)`입니다. 매개변수 및 반환 유형(예: `noexcept` 또는 `constexpr`이 있는 경우) 이외의 함수 선언 요소는 제외됩니다. (`noexcept` 및 `constexpr`은 항목 14 및 15에 설명되어 있습니다.) "서명"의 공식 정의는 나와 약간 다르지만 이 책에서는 내 정의가 더 유용합니다. (공식 정의는 때때로 반환 유형을 생략합니다.)

새로운 C++ 표준은 일반적으로 이전 표준으로 작성된 코드의 유효성을 유지하지만 때때로 표준화 위원회는 기능을 더 이상 사용하지 않습니다. 이러한 기능은 표준화 사항 선고를 받았으며 향후 표준에서 제거될 수 있습니다. 컴파일러는 더 이상 사용되지 않는 기능의 사용에 대해 경고할 수도 있고 경고하지 않을 수도 있지만 이를 피하기 위해 최선을 다해야 합니다. 그들은 미래의 포팅 골칫거리로 이어질 수 있을 뿐만 아니라 일반적으로 이를 대체하는 기능보다 열등합니다. 예를 들어, `std::auto_ptr`는 C++11에서 더 이상 사용되지 않습니다.

`std::unique_ptr`이 동일한 작업을 수행하기 때문에 더 좋습니다.

때때로 표준에서는 작업의 결과가 정의되지 않은 동작이라고 말합니다. 즉, 런타임 동작은 예측할 수 없으며 이러한 불확실성을 피하고 싶은 것은 말할 필요도 없습니다. 정의되지 않은 동작이 있는 작업의 예로는 대괄호("[]")를 사용하여 `std::vector`의 경계를 넘어 색인을 생성하거나, 초기화되지 않은 반복자를 역참조하거나, 데이터 경쟁에 참여하는 것(즉, 적어도 두 개 이상의 스레드가 있는 경우 그 중 하나는 기록적이며 동시에 동일한 메모리 위치에 액세스함).

나는 새로운 원시 포인터에서 반환된 것과 같은 내장 포인터를 호출합니다. 원시 포인터의 반대는 스마트 포인터입니다. 스마트 포인터는 일반적으로 포인터 역참조 연산자(`operator->` 및 `operator*`)를 오버로드하지만 항목 20에서는 `std::weak_ptr`이 예외라고 설명합니다.

소스 코드 주석에서 나는 때때로 "생성자"를 `ctor`로, "소멸자"를 `dtor`로 약칭합니다.

버그 보고 및 개선 제안

나는 이 책을 명확하고 정확하며 유용한 정보로 채우기 위해 최선을 다했지만 분명히 더 나은 방법이 있을 것입니다. 모든 종류의 오류(기술적, 설명적, 문법적, 인쇄적 등)를 발견하거나 책을 개선할 수 있는 방법에 대한 제안 사항이 있는 경우 emc++@aristeia.com으로 이메일을 보내주십시오. 새로운 인쇄물은 나에게

Effective Modern C++를 수정할 수 있는 기회가 있고 내가 모르는 문제를 해결할 수 없습니다!

내가 알고 있는 문제 목록을 보려면 책의 정오표 페이지([http://www.aristeia.com/BookErrata/
emc++-errata.html](http://www.aristeia.com/BookErrata/emc++-errata.html))를 참조하십시오.

1장

유형 추론하기

C++98에는 형식 추론에 대한 단일 규칙 집합이 있었습니다. 바로 함수 템플릿에 대한 규칙입니다. C++11은 해당 규칙 집합을 약간 수정하고 두 개를 추가합니다. 하나는 `auto`용이고 다른 하나는 `decltype`용입니다. 그런 다음 C++14는 `auto` 및 `decltype`이 사용될 수 있는 사용 컨텍스트를 확장 합니다. 유형 추론이 점점 더 광범위하게 적용되면서 명백하거나 중복되는 유형을 철자법으로 만드는 횡포에서 벗어날 수 있습니다. 소스 코드의 한 지점에서 유형을 변경하면 자동으로 유형 추론을 통해 다른 위치로 전파되기 때문에 C++ 소프트웨어를 보다 쉽게 적응할 수 있습니다. 그러나 컴파일러에서 추론한 유형이 원하는 만큼 명확하지 않을 수 있기 때문에 코드를 추론하기가 더 어려울 수 있습니다.

형식 추론이 어떻게 작동하는지에 대한 확실한 이해 없이는 현대 C++에서 효과적인 프로그래밍이 거의 불가능합니다. 형식 추론이 발생하는 컨텍스트가 너무 많습니다. 함수 템플릿 호출, `auto`가 나타나는 대부분의 상황, `decltype` 표현식, 그리고 C++14부터 수수께끼 같은 `decltype(auto)` 구문이 사용됩니다.

이 장에서는 모든 C++ 개발자에게 필요한 형식 추론에 대한 정보를 제공합니다. 템플릿 유형 추론이 작동하는 방식, `auto`가 이를 기반으로 구축하는 방식, `decltype`이 자체적으로 진행되는 방식을 설명합니다. 또한 컴파일러가 형식 추론 결과를 표시하도록 강제하여 컴파일러가 원하는 형식을 추론하도록 하는 방법도 설명합니다.

항목 1: 템플릿 유형 추론을 이해하십시오.

복잡한 시스템의 사용자가 작동 방식에 대해 알지 못하면서도 작동 방식에 만족할 때 시스템 설계에 대해 많은 것을 알 수 있습니다. 이 측정으로 볼 때 C++의 템플릿 유형 추론은 엄청난 성공을 거두었습니다. 수백만 명의 프로그래머가 통과했습니다.

템플릿 함수에 대한 인수는 완전히 만족스러운 결과를 얻었지만 많은 프로그래머가 해당 함수에서 사용하는 유형이 어떻게 추론되었는지에 대한 가장 모호한 설명 이상을 제공하기가 어렵습니다.

그 그룹에 당신이 포함된다면 좋은 소식과 나쁜 소식이 있습니다. 좋은 소식은 템플릿에 대한 형식 추론이 최신 C++의 가장 강력한 기능 중 하나인 auto의 기반이라는 것입니다. C++98이 템플릿 유형을 추론하는 방식에 만족했다면 C++11이 자동 유형을 추론하는 방식에 만족한 것입니다. 나쁜 소식은 템플릿 유형 추론 규칙이 자동 컨텍스트에서 적용될 때 템플릿에 적용될 때보다 직관적이지 않은 것처럼 보일 때가 있다는 것입니다. 이러한 이유로 auto가 기반으로 하는 템플릿 유형 추론의 측면을 진정으로 이해하는 것이 중요합니다. 이 항목은 알아야 할 사항을 다룹니다.

의사 코드를 약간만 간과하고 싶다면 함수 템플릿을 다음과 같이 생각할 수 있습니다.

```
template<typename T> 무
효 f(ParamType 매개변수);
```

통화는 다음과 같이 표시될 수 있습니다.

f(expr);	// 어떤 식으로 f를 호출
----------	-----------------

컴파일하는 동안 컴파일러는 expr을 사용하여 T와 ParamType의 두 가지 유형을 추론합니다.

ParamType은 종종 const 또는 참조 한정자와 같은 장식을 포함하기 때문에 이러한 유형은 종종 다릅니다. 예를 들어 템플릿이 다음과 같이 선언되면

```
template<typename T> 무
효 f(const T& param);           // ParamType은 const T&입니다.
```

우리는 이 전화를 받았습니다.

정수 x = 0;	
f(x);	// f를 int로 호출

T는 int로 추론되지만 ParamType은 const int&로 추론된다.

T에 대해 추론된 유형이 함수에 전달된 인수의 유형과 동일할 것으로 예상하는 것은 당연합니다. 즉, T는 expr의 유형입니다. 위의 예에서 x는 int이고 T는 int로 추론됩니다. 그러나 항상 그런 식으로 작동하지는 않습니다. T에 대해 추론되는 유형은 expr의 유형뿐만 아니라 ParamType의 형식에도 의존합니다. 세 가지 경우가 있습니다.

- ParamType 은 포인터 또는 참조 유형이지만 범용 참조는 아닙니다. (대학-
sal 참조는 항목 24 에 설명되어 있습니다. 이 시점에서 알아야 할 것은
존재하며 lvalue 참조 또는 rvalue 참조와 동일하지 않습니다.)
- ParamType 은 범용 참조입니다.
- ParamType 은 포인터도 참조도 아닙니다.

따라서 우리는 세 가지 유형 연역 시나리오를 검토해야 합니다. 각각을 기반으로 할 것입니다.
템플릿 및 호출에 대한 일반 양식:

템플릿<유형이름 T>
무효 f(ParamType 매개변수);

f(expr); // expr 에서 T 및 ParamType 추론

사례 1: ParamType 은 참조 또는 포인터이지만 범용은 아닙니다. 참조

가장 간단한 상황은 ParamType 이 참조 유형 또는 포인터 유형이지만
보편적인 참조가 아닙니다. 이 경우 유형 연역은 다음과 같이 작동합니다.

1. expr의 유형이 참조인 경우 참조 부분을 무시하십시오.
2. 그런 다음 T 를 결정하기 위해 ParamType 에 대해 expr의 유형을 패턴 일치시킵니다.

예를 들어 이것이 우리의 템플릿이라면,

템플릿<유형이름 T>
무효 f(T& 매개변수); // param은 참조입니다.

그리고 우리는 이러한 변수 선언을 가지고 있습니다.

정수 x = 27;	// x는 정수입니다.
const int cx = x; 상수 정	// cx는 const int입니다.
수 및 rx = x;	// rx는 const int로 x에 대한 참조입니다.

다양한 호출에서 param 및 T에 대한 추론 유형은 다음과 같습니다.

f(x);	// T는 int, param의 유형은 int&
f(cx);	// T는 const int, // param의 유형은 const int& 입니다.
f(rx);	// T는 const int, // param의 유형은 const int& 입니다.

두 번째 및 세 번째 호출에서 cx 및 rx가 const 값을 지정하기 때문에 T const int로 추론되어 const int&의 매개변수 유형을 생성합니다. 그건 호출자에게 중요합니다. 참조 매개변수에 const 자체를 전달할 때 그 객체가 수정할 수 없는 상태로 남아 있을 것으로 예상합니다. 즉, 매개변수가 const에 대한 참조가 되도록 합니다. 이것이 T& 매개변수를 사용하는 템플릿에 const 자체를 전달하는 것이 안전한 이유입니다.

객체의 constness는 T에 대해 추론된 유형의 일부가 됩니다.

세 번째 예에서 rx의 유형이 참조이지만 T는 다음과 같이 추론됩니다. 참고가 되지 않습니다. rx의 참조성은 유형 연역 중에 무시되기 때문입니다. 응.

이 예는 모두 lvalue 참조 매개변수를 보여주지만 유형 추론은 작동합니다. rvalue 참조 매개변수에 대해 정확히 동일한 방식입니다. 물론 rvalue 인수만 ment는 rvalue 참조 매개변수에 전달될 수 있지만 그 제한에는 아무 것도 없습니다. 유형 추론과 관련이 있습니다.

f의 매개변수 유형을 T&로 변경하면 상황이 약간 변경되지만 정말 놀라운 방법이 아닙니다. cx와 rx의 constness는 계속해서 존중됩니다. 테드, 하지만 이제 param이 const에 대한 참조라고 가정하기 때문에 더 이상 const가 T의 일부로 추론되어야 합니다.

템플릿<유형이름 T>

무효 `f(const T& 매개변수); // param은 이제 ref-to-const입니다.`

정수 x = 27;	// 이전과
const int cx = x; 상수 정	// 이전과
수 및 rx = x;	// 이전과
f(x);	// T는 int, param의 유형은 const int&
f(cx);	// T는 int, param의 유형은 const int&
f(rx);	// T는 int, param의 유형은 const int&

이전과 마찬가지로 rx의 참조성은 유형 추론 중에 무시됩니다.

param이 참조가 아닌 포인터(또는 const에 대한 포인터)라면 상황은 기본적으로 같은 방식으로 작동합니다.

템플릿<유형이름 T>

무효 `f(T* 매개변수); // param은 이제 포인터입니다.`

정수 x = 27; 상	// 이전과
수 정수 *px = &x;	// px는 const int로서의 x에 대한 ptr입니다.

```
f(&x);           // T는 int, param의 유형은 int*
```

```
f(픽셀);       // T는 const int,
                // param의 유형은 const int* 입니다.
```

지금쯤이면 C++의 유형 연역

참조 및 포인터 매개변수에 대해 매우 자연스럽게 작동하는 규칙

서면 양식은 정말 둔합니다. 모든 것이 분명합니다! 정확히 당신이 원하는 것입니다
유형 추론 시스템에서.

사례 2: ParamType 은 범용 참조입니다.

범용 참조 매개변수를 사용하는 템플릿의 경우 상황이 덜 명확합니다. 그런

매개변수는 rvalue 참조처럼 선언됩니다(즉,

형식 매개변수 T, 범용 참조의 선언된 형식은 T&&)이지만 서로 다르게 동작합니다.

완전히 lvalue 인수가 전달될 때. 전체 이야기는 [항목 24에](#) 설명되어 있지만

다음은 헤드라인 버전입니다.

- expr 이 lvalue이면 T와 ParamType 이 모두 lvalue 참조로 추론됩니다.

두 배로 이례적인 일입니다. 첫째, 템플릿 유형 추론의 유일한 상황입니다.

여기서 T는 참조로 추론됩니다. 둘째, ParamType 이 선언 되었지만

rvalue 참조에 대한 구문을 사용하여 추론된 유형은 lvalue 참조입니다.

- expr 이 rvalue이면 "정상"(즉, 사례 1) 규칙이 적용됩니다.

예를 들어:

```
템플릿<유형이름 T>
무효 f(T&& 매개변수);           // param은 이제 범용 참조입니다.
```

```
정수 x = 27;                      // 이전과
const int cx = x; 상수 정          // 이전과
수 및 rx = x;                      // 이전과
```

```
f(x);                            // x는 lvalue이므로 T는 int&,
                                // param의 유형도 int& 입니다.
```

```
f(cx);                           // cx는 lvalue이므로 T는 const int&,
                                // param의 유형도 const int& 입니다.
```

```
f(rx);                           // rx는 lvalue이므로 T는 const int&,
                                // param의 유형도 const int& 입니다.
```

```
f(27);                           // 27은 rvalue이므로 T는 int,
                                // 따라서 param의 유형은 int&& 입니다.
```

항목 24는 이러한 예가 어떻게 작동하는지 정확히 설명합니다. 여기서 핵심은 범용 참조 매개변수에 대한 형식 추론 규칙이 lvalue 참조 또는 rvalue 참조인 매개변수에 대한 형식 추론 규칙과 다르다는 것입니다. 특히, 범용 참조가 사용 중일 때 형식 추론은 lvalue 인수와 rvalue 인수를 구분합니다. 범용 참조가 아닌 경우에는 절대 발생하지 않습니다.

사례 3: ParamType 이 포인터도 참조도 아닙니다.

ParamType 이 포인터도 참조도 아닌 경우 통과 값을 처리합니다.

```
template<typename T>
void f(T 매개변수);           // param은 이제 값으로 전달됩니다.
```

이는 param이 전달된 모든 것의 복사본이 됨을 의미합니다. 즉, 완전히 새로운 객체입니다. param이 새로운 객체가 될 것이라는 사실은 T가 expr에서 추론되는 방법을 제어하는 규칙에 동기를 부여합니다.

1. 이전과 마찬가지로 expr의 유형이 참조인 경우 참조 부분을 무시합니다.
2. expr의 참조성을 무시하고 expr이 const라면 그것도 무시합니다. 휘발성이라면 무시하십시오. (휘발성 객체는 흔하지 않습니다. 일반적으로 장치 드라이버를 구현하는 데만 사용됩니다. 자세한 내용은 항목 40을 참조하세요.)

따라서:

```
x = 27; // 이전과 같이 const int // 이전과 같은 초기화
과                                및 rx = x; // 이전
```

```
f(x);          // T와 param의 유형은 모두 int입니다.
```

```
f(cx);         // T와 param의 유형은 다시 모두 int입니다.
```

```
f(rx);         // T와 param의 유형은 여전히 둘 다 int입니다.
```

cx 및 rx가 const 값 나타내더라도 param은 const가 아닙니다. 그것은 의미가 있습니다. param은 cx 및 rx(cx 또는 rx의 복사본)와 완전히 독립적인 객체입니다. cx와 rx가 수정될 수 없다는 사실은 param이 수정될 수 있는지 여부에 대해 아무 말도 하지 않습니다. 이것이 param의 유형을 추론할 때 expr의 constness(및 휘발성이 있는 경우)가 무시되는 이유입니다. expr을 수정할 수 없다고 해서 복사본이 수정될 수 없다는 의미는 아닙니다.

const(및 volatile)는 값 기준 매개변수에 대해서만 무시된다는 점을 인식하는 것이 중요합니다. 우리가 보았듯이 const에 대한 참조 또는 포인터에 대한 매개변수의 경우 expr의 constness는 형식 추론 동안 유지됩니다. 그러나 고려

expr이 const 객체에 대한 const 포인터이고 expr이 by value param으로 전달되는 경우:

```
template<typename T>
void f(T 매개변수); // param은 여전히 값으로 전달됩니다.

const char* const ptr = // ptr은 const 객체에 대한 const 포인터입니다.
    "재미있는 포인터"; // const char * const 유형의 arg 전달
```

여기에서 별표 오른쪽에 있는 const는 ptr을 const로 선언합니다. ptr은 다른 위치를 가리키도록 만들 수 없으며 null로 설정할 수도 없습니다. (별표 왼쪽에 있는 const는 ptr이 가리키는 문자열(문자열)이 const이므로 수정할 수 없습니다.) ptr이 f에 전달되면 포인터를 구성하는 비트가 param에 복사됩니다. 따라서 포인터 자체(ptr)는 값으로 전달됩니다. 값에 의한 매개변수에 대한 형식 추론 규칙에 따라 ptr의 constness는 무시되고 param에 대해 추론된 형식은 const char*, 즉 const 문자열에 대한 수정 가능한 포인터가 됩니다. ptr이 가리키는 constness는 형식 추론 중에 유지되지만 ptr 자체의 constness는 복사하여 새 포인터 param을 만들 때 무시됩니다.

배열 인수 주류 템플릿

유형 추론에 대해 거의 다릅니다. 하지만 알아둘 가치가 있는 틈새 사례가 있습니다. 배열 유형은 때때로 상호 교환 가능한 것처럼 보이지만 포인터 유형과 다릅니다. 이러한 환상의 주요 원인은 많은 컨텍스트에서 배열이 첫 번째 요소에 대한 포인터로 붕괴된다는 것입니다. 이 붕괴로 인해 다음과 같은 코드가 컴파일될 수 있습니다.

```
const char name[] = "JP Briggs"; // 이름의 유형은 // const char[13]
```

```
const char * ptrToName = 이름; // 배열은 포인터로 소멸
```

여기서 const char* 포인터 ptrToName은 const char[13]인 name으로 초기화됩니다. 이러한 유형(const char* 및 const char[13])은 동일하지 않지만 배열-포인터 붕괴 규칙 때문에 코드가 컴파일됩니다.

그러나 배열이 값에 의한 매개변수를 사용하는 템플릿에 전달되면 어떻게 될까요? 그러면 어떻게 됩니까?

```
template<typename T>
void f(T 매개변수); // 값에 의한 매개변수가 있는 템플릿
```

```
f(이름); // T와 param에 대해 추론되는 유형은 무엇입니까?
```

배열인 함수 매개변수와 같은 것은 없다는 관찰로 시작합니다. 예, 예, 구문은 합법적입니다.

```
무효 myFunc(int param[]);
```

그러나 배열 선언은 포인터 선언으로 처리됩니다. 즉, myFunc는 다음과 같이 동등하게 선언될 수 있습니다.

```
무효 myFunc(int* 매개변수); // 위와 같은 기능
```

배열과 포인터 매개변수의 이러한 동등성은 C++의 기초에 있는 C 루트에서 파생된 약간의 잎사귀이며 배열과 포인터 유형이 동일하다는 환상을 조장합니다.

배열 매개변수 선언은 포인터 매개변수인 것처럼 취급되기 때문에 템플릿 함수에 값으로 전달되는 배열의 유형은 포인터 유형으로 추론됩니다. 즉, 템플릿 f에 대한 호출에서 형식 매개변수 T는 const char*로 추론됩니다.

```
f(이름); // 이름은 배열이지만 T는 const char*로 추론됨
```

그러나 이제 커브 볼이 옵니다. 함수는 진정한 배열인 매개변수를 선언할 수 없지만 배열에 대한 참조인 매개변수는 선언할 수 있습니다! 따라서 참조로 인수를 사용하도록 템플릿 f를 수정하면

```
template<typename T>
void f(T& param); // 참조 매개변수가 있는 템플릿
```

배열을 전달합니다.

```
f(이름); // 배열을 f에 전달
```

T에 대해 추론된 유형은 배열의 실제 유형입니다! 그 유형에는 배열의 크기가 포함되므로 이 예에서 T는 const char [13]으로 추론되고 f의 매개변수(이 배열에 대한 참조) 유형은 const char(&)[13]입니다. 예, 구문은 유독해 보이지만, 그것이 관심을 갖는 소수의 영혼들과 함께 당신에게 몇도 포인트를 줄 것이라는 것을 알고 있습니다.

흥미롭게도 배열에 대한 참조를 선언하는 기능을 사용하면 배열에 포함된 요소의 수를 추론하는 템플릿을 만들 수 있습니다.

```
// 배열의 크기를 컴파일 타임 상수로 반환합니다. (배열 매개변수에는 // 포함된 요소의 수에
```

```
만 // 신경을 쓰기 때문에 // 배열 매개변수에는 이름이 없습니다.) // 정보 보기
```

```
template<typename T, std::size_t N> constexpr std::size_t arraySize(T (&) [N])
```

```
래 { // constexpr // noexcept // 아
```

```

    반환 N;
}

// 그리고
// 예외 없음

```

항목 15 에서 설명하는 것처럼 이 함수를 constexpr 선언하면 결과를 사용할 수 있습니다.

컴파일 중. 이를 통해 동일한 배열을 선언할 수 있습니다.

크기가 중괄호 이나셜에서 계산되는 두 번째 배열의 요소 수

아이저:

```

int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };
// keyVals는
// 7개의 요소

```

```

int mappingVals[arraySize(keyVals)];
// 마찬가지입니다
// 매핑된 값

```

물론 현대의 C++ 개발자라면 당연히 std::array보다 std::array를 선호할 것입니다.

내장 배열:

```

std::array<int, arraySize(keyVals)> mappingVals; // 매핑된 값
// 크기는 7

```

arraySize가 noexcept로 선언되면 컴파일러가 더 잘 생성하는 데 도움이 됩니다.

암호. 자세한 내용은 항목 14를 참조하십시오.

함수 인수

배열은 C++에서 포인터로 붕괴될 수 있는 유일한 것이 아닙니다. 함수 유형은

함수 포인터로의 붕괴, 그리고 우리가 유형 연역에 관해 논의한 모든 것

배열에 대한 tion은 함수에 대한 형식 추론과 함수로의 붕괴에 적용됩니다.

포인터. 결과적으로:

```

무효 someFunc(int, double); // someFunc는 함수입니다.
// 타입은 void(int, double)

```

```

템플릿<유형이름 T>
무효 f1(T 매개변수); // f1에서 값으로 전달된 매개변수

```

```

템플릿<유형이름 T>
무효 f2(T& 매개변수); // f2에서 ref에 의해 전달된 param

```

```

f1(someFunc); // ptr-to-func로 추론된 param;
// 타입은 void (*)(int, double)

```

```

f2(someFunc); // param은 ref-to-func로 추론됨;
// 타입은 void (&)(int, double)

```

이것은 실제로 거의 차이가 없지만 배열 포인터 붕괴에 대해 알고자 한다면 함수 대 포인터 붕괴에 대해서도 알고 있을 것입니다.

템플릿 유형 추론에 대한 자동 관련 규칙이 있습니다. 나는 처음에 그것들이 매우 간단하고 대부분이 그렇다고 언급했습니다. 그러나 범용 참조에 대한 유형을 추론할 때 lvalue를 특별하게 처리하면 물이 약간 흐려지고 배열 및 함수에 대한 포인터 감쇠 규칙은 훨씬 더 큰 탁도를 유발합니다. 때때로 당신은 단순히 당신의 컴파일러를 잡고 “당신이 추론하고 있는 유형을 말해주세요!”라고 요구하고 싶을 때가 있습니다. 그런 일이 발생하면 항목 4로 넘어가십시오. 컴파일러가 그렇게 하도록 유도하는 데 전념하기 때문입니다.

기억해야 할 사항 • 템

플릿 유형 추론 동안 참조인 인수는 참조가 아닌 것으로 처리됩니다. 즉, 참조성은 무시됩니다.
 • 범용 참조 매개변수의 유형을 추론할 때 lvalue 인수는 특별한 처리를 받습니다.
 값에 의한 매개변수의 유형을 추론할 때 const 및/또는 volatile 인수는 비-const 및 비휘발성으로 처리됩니다.

- 템플릿 유형 추론 중 배열 또는 함수 이름인 인수
참조를 초기화하는 데 사용되지 않는 한 포인터로 소멸됩니다.

항목 2: 자동 유형 추론을 이해하십시오.

템플릿 유형 추론에 대한 항목 1 을 읽었다면 자동 유형 추론에 대해 알아야 할 거의 모든 것을 이미 알고 있을 것입니다. 한 가지 흥미로운 예외를 제외하고 자동 유형 추론은 템플릿 유형 추론이기 때문입니다. 그러나 어떻게 그럴 수 있습니까?

템플릿 유형 추론에는 템플릿, 함수 및 매개변수가 포함되지만 auto는 이러한 항목을 전혀 처리하지 않습니다.

그것은 사실이지만 그것은 중요하지 않습니다. 템플릿 유형 추론과 자동 유형 추론 사이에는 직접적인 매핑이 있습니다. 문자 그대로 하나에서 다른 것으로 알고리즘 변환이 있습니다.

항목 1 에서는 이 일반 함수 템플릿을 사용하여 템플릿 유형 추론을 설명합니다.

```
template<typename T> 무
효 f(ParamType 매개변수);
```

그리고 이 일반 호출:

```
f(expr); // 어떤 식으로 f를 호출
```

f에 대한 호출에서 컴파일러는 expr 을 사용 하여 T 및 ParamType의 유형을 추론합니다.

auto를 사용하여 변수를 선언하면 auto는 템플릿에서 T의 역할을 하고 변수의 유형 지정자는 ParamType으로 작동합니다. 이것은 설명하는 것보다 보여주는 것이 더 쉬우므로 다음 예를 고려해십시오.

자동 `x = 27;`

여기서 x에 대한 유형 지정자는 그 자체로 자동입니다. 한편 이 선언문에서는

`const 자동 cx = x;`

유형 지정자는 `const auto`입니다. 그리고 여기,

상수 자동& `rx = x;`

유형 지정자는 `const auto&`입니다. 이 예에서 x, cx 및 rx의 유형을 추론하기 위해 컴파일러는 각 선언에 대한 템플릿이 있는 것처럼 작동할 뿐만 아니라 해당 초기화 표현식을 사용하여 해당 템플릿에 대한 호출도 있는 것처럼 작동합니다.

```
template<typename T> void // x의 유형을 추론하기 위한 // 개념적 템플릿
func_for_x(T 매개변수);
```

```
func_for_x(27); // 개념적 호출: param의 // 추론된 유형은 x의
유형입니다.
```

```
template<typename T> // void에 대한 개념적 템플릿 func_for_cx(const T param); // cx의 유형
추론
```

```
func_for_cx(x); // 개념적 호출: param의 // 추론된 유형은 cx
의 유형입니다.
```

```
template<typename T> // void에 대한 개념적 템플릿 func_for_rx(const T& param); // rx의 유형
추론
```

```
func_for_rx(x); // 개념적 호출: param의 // 추론된 유형은 rx
의 유형입니다.
```

내가 말했듯이 auto에 대한 유형을 추론하는 것은 단 하나의 예외(곧 논의할 것임)를 제외하고 템플릿에 대한 유형을 추론하는 것과 동일합니다.

항목 1은 일반 함수 템플릿에서 param에 대한 유형 지정자인 ParamType의 특성에 따라 템플릿 유형 추론을 세 가지 경우로 나눕니다. auto를 사용하는 변수 선언에서 유형 지정자는 ParamType을 대신 하므로 이에 대한 세 가지 경우도 있습니다.

- 사례 1: 유형 지정자가 포인터 또는 참조이지만 범용 참조는 아닙니다. • 사례 2: 유형 지정자가 범용 참조입니다.

- 사례 3: 유형 지정자가 포인터도 참조도 아닙니다.

우리는 이미 사례 1과 3의 예를 보았습니다.

```
자동 x = 27; // 경우 3(x는 ptr도 참조도 아님)
```

```
const 자동 cx = x; // 사례 3(cx도 아님)
```

```
상수 자동& rx = x; // 사례 1(rx는 비범용 참조입니다.)
```

사례 2는 예상대로 작동합니다.

```
자동&& uref1 = x; // x는 int와 lvalue,  
// 따라서 uref1의 유형은 int&입니다.
```

```
자동&& uref2 = cx; // cx는 const int와 lvalue,  
// 따라서 uref2의 유형은 const int&입니다.
```

```
자동&& uref3 = 27; // 27은 int와 rvalue,  
// 따라서 uref3의 유형은 int&&입니다.
```

항목 1 은 배열 및 함수 이름이

비참조 유형 지정자에 대한 포인터입니다. 이는 자동 유형 추론에서도 발생합니다.

```
const char name[] = "RN  
Briggs"; // 이름의 유형은 const char[13]입니다.
```

```
자동 arr1 = 이름; // arr1의 유형은 const char*입니다.
```

```
자동& arr2 = 이름; // arr2의 유형은  
// const char (&)[13]
```

```
무효 someFunc(int, double); // someFunc는 함수입니다.  
// 타입은 void(int, double)
```

```
자동 func1 = someFunc; // func1의 유형은  
// 무효 (*) (int, double)
```

```
auto& func2 = someFunc; // func2의 유형은  
// 무효 (&) (int, double)
```

보시다시피 자동 유형 추론은 템플릿 유형 추론처럼 작동합니다. 그들은 본질적으로 같은 동전의 양면입니다.

한 가지 방법을 제외하고는 다릅니다. 초기 값이 27인 int를 선언하려는 경우 C++98에서 두 가지 구문 선택을 제공한다는 관찰로 시작하겠습니다.

```
정수 x1 = 27; 정수
x2(27);
```

C++11은 균일한 초기화 지원을 통해 다음을 추가합니다.

```
정수 x3 = {27}; 정수
x4[27];
```

전체적으로 4가지 구문이지만 결과는 하나뿐입니다. 값이 27인 int입니다.

그러나 항목 5에서 설명하는 것처럼 고정 유형 대신 auto를 사용하여 변수를 선언하는 것이 이점이 있으므로 위의 변수 선언에서 int를 auto로 바꾸는 것이 좋습니다. 간단한 텍스트 대체는 다음 코드를 생성합니다.

```
자동 x1 = 27; 자동
x2(27); 자동 x3 =
{ 27 }; 자동 x4{ 27 };
```

이러한 선언은 모두 컴파일되지만 대체하는 것과 같은 의미는 아닙니다. 처음 두 명령문은 실제로 값이 27인 int 유형의 변수를 선언합니다. 그러나 두 번째 두 명령문은 값이 27인 단일 요소를 포함하는 std::initializer_list<int> 유형의 변수를 선언합니다!

자동 x1 = 27;	// 타입은 정수, 값은 27
자동 x2(27);	// 마찬가지
자동 x3 = { 27 };	// 유형은 std::initializer_list<int>, // 값은 { 27 }
자동 x4{ 27 };	// 마찬가지

이것은 auto에 대한 특별한 유형 연역 규칙 때문입니다. 자동 선언된 변수의 이니셜라이저가 중괄호로 묶인 경우 추론된 유형은 std::initializer_list입니다. 그러한 유형을 추론할 수 없는 경우 (예: 중괄호 이니셜라이저의 값이 다른 유형이기 때문에) 코드는 거부됩니다.

```
자동 x5 = { 1, 2, 3.0 }; // 오류! // std::initializer_list<T>에 대한 T를 추론할 수 없습니다
다.
```

주석에서 알 수 있듯이 이 경우 유형 연역은 실패하지만 실제로 두 가지 유형의 유형 연역이 발생한다는 것을 인식하는 것이 중요합니다. 한 종류는 `auto`: `x5`의 유형을 추론해야 하는 사용에서 비롯됩니다. `x5`의 이니셜라이저는 중괄호 안에 있으므로 `x5`는 `std::initializer_list`로 추론되어야 합니다. 그러나 `std::initializer_list`는 템플릿입니다. 인스턴스화는 일부 유형 `T`에 대한 `std::initializer_list<T>`이며, 이는 `T`의 유형도 추론해야 함을 의미합니다. 이러한 연역은 여기서 발생하는 두 번째 유형 연역인 템플릿 유형 연역의 범위에 속합니다. 이 예에서는 중괄호 이니셜라이저의 값에 단일 유형이 없기 때문에 해당 추론이 실패합니다.

중괄호 이니셜라이저의 처리는 자동 유형 추론과 템플릿 유형 추론이 다른 유일한 방법입니다. 자동 선언 변수가 중괄호 이니셜라이저로 초기화되면 추론된 유형은 `std::initializer_list`의 인스턴스화입니다.

그러나 해당 템플릿이 동일한 이니셜라이저를 전달하면 형식 추론이 실패하고 코드가 거부됩니다.

`자동 x = { 11, 23, 9 }; // x의 유형은 // std::initializer_list<int>`
입니다.

`template<typename T>` // 매개변수가 있는 템플릿 // // `x`의 선언
`void f(T 매개변수);` 과 동일한 선언

`f({ 11, 23, 9 });` // 오류! `T`의 유형을 추론할 수 없습니다.

그러나 템플릿에서 `param`이 알 수 없는 `T`에 대한 `std::initializer_list<T>`임을 지정하면 템플릿 유형 추론은 `T`가 무엇인지 추론합니다.

`template<typename T> 무`
효 `f(std::initializer_list<T> initList);`

`f({ 11, 23, 9 });` // `T`는 `int`로 추론되고 `initList`의 // 유형은
`std::initializer_list<int>`입니다.

따라서 `auto`와 템플릿 유형 추론의 유일한 실제 차이점은 `auto`는 중괄호 이니셜라이저가 `std::initializer_list`를 나타내는 것으로 가정하지만 템플릿 유형 추론은 그렇지 않다는 것입니다.

자동 유형 추론에 중괄호 이니셜라이저에 대한 특별한 규칙이 있지만 템플릿 유형 추론에는 없는 이유가 궁금할 수 있습니다. 나는 이것이 스스로 궁금하다. 아아, 설득력 있는 설명을 찾지 못했습니다. 그러나 규칙은 규칙이며 이는 `auto`를 사용하여 변수를 선언하고 중괄호 이니셜라이저를 사용하여 초기화하면 추론된 유형은 항상 `std::initializer_list`가 된다는 것을 기억해야 합니다. 균일한 초기화의 철학을 수용하는 경우 이를 염두에 두는 것이 특히 중요합니다. 즉, 초기화 값을 당연히 중괄호로 묶는 것입니다. 고전적인 실수

C++11 프로그래밍에서는 다른 것을 선언하려고 할 때 실수로 std::initializer_list 변수를 선언합니다. 이 함정은 일부 개발자가 필요한 경우에만 이니셜라이저 주위에 중괄호를 두는 이유 중 하나입니다. (해야 하는 경우 항목 7에서 설명합니다.)

C++11의 경우 이것이 전체 이야기이지만 C++14의 경우 이야기가 계속됩니다. C++14는 auto가 함수의 반환 유형이 추론되어야 함을 나타내도록 허용하고(항목 3 참조), C++14 람다는 매개변수 선언에서 auto를 사용할 수 있습니다. 그러나 이러한 자동 사용은 자동 유형 추론이 아닌 템플릿 유형 추론을 사용합니다. 따라서 중괄호 이니셜라이저를 반환하는 자동 반환 유형의 함수는 컴파일되지 않습니다.

```
자동 createInitList() {
    반환 { 1, 2, 3 }; // 오류: 유형을 추론할 수 없습니다. // for { 1, 2,
} 3 }
```

auto가 C++14 람다의 매개변수 유형 사양에 사용될 때도 마찬가지입니다.

```
표준::벡터<int> v;
...

자동 재설정 V =
[&v](const auto& newValue) { v = newValue; }; // C++14
...
리셋V({ 1, 2, 3 }); // 오류! 유형을 추론할 수 없습니다 // for { 1, 2,
3 }
```

기억해야 할 사항

- 자동 유형 추론은 일반적으로 템플릿 유형 추론과 동일하지만 자동 유형 추론은 중괄호 이니셜라이저가 std::initializer_list를 나타내고 템플릿 유형 추론은 그렇지 않다고 가정합니다.
- 함수 반환 유형 또는 람다 매개변수의 auto는 자동 유형 추론이 아니라 템플릿 유형 추론을 의미합니다.

항목 3: decltype을 이해하십시오.

decltype은 이상한 생물입니다. 이름이나 표현식이 주어지면 decltype은 이름이나 표현식의 유형을 알려줍니다. 일반적으로 이 문서에서 알려주는 내용은

예측하다. 그러나 때로는 머리를 굵적이며 참고 문헌이나 온라인 Q&A 사이트로 눈을 돌리는 결과를 제공합니다.

놀라운 사실이 없는 전형적인 경우부터 시작하겠습니다. 템플릿 및 auto(항목 1 및 2 참조)에 대한 형식 추론 중에 발생하는 것과 대조적으로 decltype은 일반적으로 사용자가 제공한 이름 또는 표현식의 정확한 형식을 되돌립니다.

```
상수 정수 i = 0;           // decltype(i)은 const int 입니다.

bool f(const 위젯& w); // decltype(w)은 const Widget& 입니다.
                        // decltype(f)은 bool(const Widget&)입니다.

구조체 포인트 { int x,
    y; };                  // decltype(Point::x) is int //
                        // decltype(Point::y) is int

위젯 w;                 // decltype(w)는 위젯 입니다.

만약 (f(w)) ...          // decltype(f(w))은 bool 입니다.

template<typename T> 클
래스 벡터 { 공개:
    ...
    T& 연산자[](std::size_t 인덱스);
    ...
};

벡터<int> v;             // decltype(v)은 vector<int> 입니다.
...
만약 (v[0] == 0) ...       // decltype(v[0])은 int&
```

보다? 놀라움이 없습니다.

C++11에서 decltype의 주요 용도는 함수의 반환 유형이 매개변수 유형에 따라 달라지는 함수 템플릿을 선언하는 것일 수 있습니다. 예를 들어, 대괄호(즉, "[]" 사용)와 인덱스를 통해 인덱싱을 지원하는 컨테이너를 취하는 함수를 작성하고 인덱싱 결과를 반환하기 전에 사용자를 인증한다고 가정합니다. 작업. 함수의 반환 유형은 인덱싱 작업에서 반환된 유형과 동일해야 합니다.

T 유형의 개체 컨테이너에 대한 operator[]는 일반적으로 T&를 반환합니다. 이것은 예를 들어 std::deque의 경우이며 거의 항상 std::vector의 경우입니다. 그러나 std::vector<bool>의 경우 operator[]는 bool&를 반환하지 않습니다. 대신 완전히 새로운 객체를 반환합니다. 이 상황의 이 유와 방법은 다음에서 탐색됩니다.

항목 6, 하지만 여기서 중요한 것은 컨테이너의 연산자[]가 반환하는 유형이 컨테이너에 따라 다르다는 것입니다.

decltype을 사용하면 이를 쉽게 표현할 수 있습니다. 다음은 우리가 작성하려는 템플릿의 첫 번째 것으로, 반환 유형을 계산하기 위해 decltype을 사용하는 것을 보여줍니다. 템플릿은 약간의 개선이 필요하지만 지금은 연기하겠습니다.

```
template<typename 컨테이너, typename 인덱스> auto           // 작동하지만 // 개
authAndAccess(컨테이너& c, 인덱스 i) -> decltype(c[i])    선이 필요합니다.

{
    인증 사용자(); 반환 c[i];
}

}
```

함수 이름 앞에 auto를 사용하는 것은 형식 추론과 관련이 없습니다.

오히려 이것은 C++11의 후행 반환 유형 구문이 사용 중임을 나타냅니다. 즉, 함수의 반환 유형이 매 개변수 목록("->" 뒤) 다음에 선언될 것임을 나타냅니다. 후행 반환 유형은 함수의 매개변수를 반환 유형의 사양에 사용할 수 있다는 이점이 있습니다. 예를 들어 authAndAccess에서는 c와 i를 사용하여 반환 유형을 지정합니다. 일반적인 방식으로 함수 이름 앞에 반환 유형을 지정하면 c와 i가 아직 선언되지 않았기 때문에 사용할 수 없습니다.

이 선언을 통해 authAndAccess는 우리가 원하는 대로 전달된 컨테이너에 적용될 때 operator[]가 반환하는 모든 유형을 반환합니다.

C++11은 단일 문 람다의 반환 유형을 추론할 수 있도록 허용하고 C++14는 이를 모든 람다와 여러 문을 포함하는 모든 함수로 확장합니다. authAndAccess의 경우 이는 C++14에서 후행 반환 유형을 생략하고 선행 auto만 남길 수 있음을 의미합니다. 이러한 형식의 선언에서 auto는 형식 추론이 발생함을 의미합니다. 특히, 이는 컴파일러가 함수의 구현에서 함수의 반환 유형을 추론한다는 것을 의미합니다.

```
template<typename 컨테이너, typename 인덱스> auto           // C++14; //
authAndAccess(컨테이너& c, 인덱스 i) {authenticateUser(); 반   정확 하지 않음 //
    환 c[i];}                                                 정확함

// c[i]에서 추론된 반환 유형
```

항목 2는 자동 반환 유형 사양이 있는 함수의 경우 컴파일러가 템플릿 유형 추론을 사용한다고 설명합니다. 이 경우 문제가 됩니다. 우리가 논의한 바와 같이, 대부분의 T 컨테이너에 대한 operator[]는 T&를 반환하지만 **항목 1**은 다음과 같이 설명합니다.

템플릿 유형 추론에서 초기화 표현식의 참조성은 무시됩니다.
이 클라이언트 코드에서 이것이 의미하는 바를 고려하십시오.

```
std::deque<int> d;
...
인증 및 액세스(d, 5) = 10; // 사용자를 인증하고 d[5]를 반환한 다음 // 10을 할당합니다. // 컴파일되지 않습니다!
```

여기서 d[5]는 int&를 반환하지만 authAndAccess에 대한 자동 반환 유형 추론은 참조를 제거하여 int의 반환 유형을 생성합니다. 함수의 반환 값인 int는 rvalue이고 위의 코드는 rvalue int에 10을 할당하려고 시도합니다. 이는 C++에서 금지되어 있으므로 코드가 컴파일되지 않습니다.

authAndAccess가 원하는 대로 작동하도록 하려면 반환 유형에 대해 decltype 유형 추론을 사용해야 합니다. 유형이 유추되는 일부 경우에 decltype 유형 추론 규칙을 사용할 필요가 있을 것으로 예상하는 C++의 수호자는 decltype(auto) 지정자를 통해 C++14에서 이를 가능하게 합니다. 처음에는 모순되는 것처럼 보일 수 있는 것(decltype 및 auto?)이 실제로는 완벽합니다. auto는 유형이 추론되도록 지정하고 decltype은 decltype 규칙을 추론 중에 사용해야 한다고 지정합니다. 따라서 다음과 같이 authAndAccess를 작성할 수 있습니다.

```
template<typename 컨테이너, typename 인덱스> // C++14; 작동, decltype(auto) // 하지만 여전히 authAndAccess(Container& c, Index i) // {} 개선이 필요합니다.
```

인증 사용자(); 반환 c[i];

}

이제 authAndAccess는 c[i]가 반환하는 모든 것을 진정으로 반환합니다. 특히, c[i]가 T&를 반환하는 일반적인 경우에 authAndAccess도 T&를 반환하고, c[i]가 객체를 반환하는 드문 경우에 authAndAccess도 객체를 반환합니다.

decltype(auto)의 사용은 함수 반환 유형에만 국한되지 않습니다. 또한 초기화 표현식에 decltype 유형 연역 규칙을 적용하려는 경우 변수 선언에 편리할 수 있습니다.

위젯 w;

const 위젯& cw = w;

자동 myWidget1 = cw;

// 자동 유형 추론:

```
// myWidget1의 유형은 Widget입니다.
```

```
decltype(자동) myWidget2 = cw; // decltype 유형 추론: // myWidget2의 유형은 //
const Widget& 입니다.
```

하지만 두 가지가 당신을 괴롭히고 있다는 것을 압니다. 하나는 내가 언급했지만 아직 설명하지 않은 authAndAccess에 대한 개선 사항입니다. 이제 해결해 보겠습니다.

authAndAccess의 C++14 버전에 대한 선언을 다시 살펴보세요.

```
template<typename 컨테이너, typename 인덱스> decltype(auto)
authAndAccess(컨테이너& c, 인덱스 i);
```

컨테이너의 요소에 대한 참조를 반환하면 클라이언트가 해당 컨테이너를 수정할 수 있기 때문에 컨테이너는 lvalue-reference-to-non-const에 의해 전달됩니다. 그러나 이것은 이 함수에 rvalue 컨테이너를 전달할 수 없음을 의미합니다. Rvalue는 lvalue 참조에 바인딩할 수 없습니다(여기서는 그렇지 않은 lvalue-references-to-const가 아닌 경우).

분명히 rvalue 컨테이너를 authAndAccess에 전달하는 것은 극단적인 경우입니다. 임시 객체인 rvalue 컨테이너는 일반적으로 authAndAccess에 대한 호출을 포함하는 명령문의 끝에서 파괴되며, 이는 해당 컨테이너의 요소에 대한 참조(일반적으로 authAndAccess가 반환하는 것)가 그것을 만든 문장의 끝. 그래도 임시 객체를 authAndAccess에 전달하는 것이 합리적일 수 있습니다. 클라이언트는 단순히 임시 컨테이너에 있는 요소의 복사본을 만들고 싶을 수 있습니다. 예를 들면 다음과 같습니다.

```
std::deque<std::string> makeStringDeque(); // 팩토리 함수
```

```
// makeStringDeque에서 // 반환된 deque의 5번째 요소의 복사본을
만듭니다. auto s = authAndAccess(makeStringDeque(), 5);
```

이러한 사용을 지원한다는 것은 lvalue와 rvalue를 모두 허용하도록 authAndAccess에 대한 선언을 수정해야 함을 의미합니다. 오버로딩은 작동하지만(하나의 오버로드는 lvalue 참조 매개변수를 선언하고 다른 하나는 rvalue 참조 매개변수를 선언함) 유지해야 할 두 가지 기능이 있습니다. 이를 피하는 방법은 authAndAccess가 lvalue 및 rvalue에 바인딩할 수 있는 참조 매개변수를 사용하도록 하는 것입니다. **항목 24**는 이것이 바로 범용 참조가 하는 일이라고 설명합니다. 따라서 authAndAccess는 다음과 같이 선언할 수 있습니다.

```
template<typename 컨테이너, typename 인덱스> decltype(auto) // c는 이제 // 범
authAndAccess(컨테이너&& c, // 용 // 참조입니다.
    색인 i);
```

이 템플릿에서 우리는 우리가 어떤 유형의 컨테이너에서 작업하고 있는지 알지 못합니다. 이는 우리가 사용하는 인덱스 객체의 유형에 대해서도 똑같이 무지하다는 것을 의미합니다. 알 수 없는 유형의 객체에 대한 전달 값을 사용하면 일반적으로 불필요한 복사의 성능 저하, 객체 슬라이싱의 동작 문제([항목 41 참조](#)), 동료의 조롱의 위험이 있지만 컨테이너 인덱스의 경우 , 인덱스 값에 대한 표준 라이브러리의 예를 따르는 것(예: std::string, std::vector 및 std::deque에 대한 operator[]에서)이 합리적으로 보이므로 값에 의한 전달을 고수할 것입니다. 그들을.

그러나 std::forward를 범용 참조에 적용하라는 [항목 25](#)의 권고 와 일치하도록 템플릿의 구현을 입데 이트해야 합니다 .

```
template<typename 컨테이너, typename 인덱스> // 최종
decltype(auto) authAndAccess(컨테이너&& c, 인덱스 i) { // C++14 //
    // 버전
```

```
    인증 사용자(); return
    std::forward<컨테이너>(c)[i];
}
```

이것은 우리가 원하는 모든 것을 수행해야 하지만 C++14 컴파일러가 필요합니다. 템플릿이 없으면 C++11 버전의 템플릿을 사용해야 합니다. 반환 유형을 직접 지정해야 한다는 점을 제외하고는 C++14와 동일합니다.

```
template<typename 컨테이너, typename 인덱스>자동 // 최종
// C++11 //
authAndAccess(컨테이너&& c, 인덱스 i) ->
    decltype(std::forward<컨테이너>(c)[i]) {authenticateUser();
    return std::forward<컨테이너>(c)[i];
```

```
}
```

당신에게 잔소리를 할 가능성이 있는 또 다른 문제는 decltype이 거의 항상 당신이 예상하는 유형을 생성하고 거의 놀라지 않는다는 이 항목의 시작 부분에서 언급한 것입니다. 솔직히 말해서, 강력한 라이브러리 구현자가 아닌 한 규칙에 대한 이러한 예외가 발생할 가능성은 거의 없습니다.

decltype의 동작을 완전히 이해하려면 몇 가지 특수한 경우에 익숙해져야 합니다. 이를 중 대부분은 이와 같은 책에서 논의를 보증하기에는 너무 모호하지만 하나를 보면 decltype과 그 사용법에 대한 통찰력을 얻을 수 있습니다.

이름에 decltype을 적용하면 해당 이름에 대해 선언된 유형이 생성됩니다. 이름은 lvalue 표현식이지만 decltype의 동작에는 영향을 미치지 않습니다. 그러나 이름보다 복잡한 lvalue 표현식의 경우 decltype은 보고된 유형이 다음과 같은지 확인합니다.

항상 lvalue 참조입니다. 즉, 이름이 아닌 lvalue 표현식의 유형이 T인 경우 decltype은 해당 유형을 T&로 보고합니다. 대부분의 lvalue 식 유형에는 본질적으로 lvalue 참조 한정자가 포함되어 있기 때문에 이는 거의 영향을 미치지 않습니다. 예를 들어 lvalue를 반환하는 함수는 항상 lvalue 참조를 반환합니다.

그러나 인식할 가치가 있는 이 동작의 의미가 있습니다. ~ 안에

정수 x = 0;

x는 변수의 이름이므로 decltype(x)는 int입니다. 그러나 이름 x를 괄호("(x)")로 묶으면 이름보다 더 복잡한 표현식이 생성됩니다. 이름이기 때문에 x는 lvalue이고 C++에서는 표현식(x)도 lvalue로 정의합니다. 따라서 decltype((x)) 은 int&입니다. 이름 주위에 괄호를 넣으면 decltype이 보고하는 유형을 변경할 수 있습니다!

C++11에서 이것은 단순한 호기심에 불과하지만 C++14의 decltype(auto) 지원과 함께 이는 return 문을 작성하는 방식의 겉보기에 사소한 변경이 추론된 결과에 영향을 미칠 수 있음을 의미합니다. 함수에 대한 유형:

decltype(자동) f1() {

정수 x = 0;

...

반환 x; // decltype(x) 는 int 이므로 f1은 int 를 반환합니다.
}

decltype(자동) f2() {

정수 x = 0;

...

리턴(x); // decltype((x)) 은 int& 이므로 f2는 int& 를 반환합니다.
}

f2는 f1과 다른 반환 유형을 가질 뿐만 아니라 지역 변수에 대한 참조도 반환합니다! 이것은 정의되지 않은 행동으로 급행열차에 오르게 하는 종류의 코드입니다.

기본 교훈은 decltype(auto)을 사용할 때 매우 세심한 주의를 기울이는 것입니다.

유형이 추론되는 표현식에서 겉보기에 중요하지 않은 세부 정보가 decltype(auto)이 보고하는 유형에 영향을 줄 수 있습니다. 추론되는 유형이 예상한 유형인지 확인하려면 **항목 4**에 설명된 기술을 사용하십시오.

동시에 더 큰 그림을 놓치지 마십시오. 물론, decltype(단독 및 auto와 함께)은 때때로 놀라운 유형 추론을 생성할 수 있지만 이는 정상적인 상황이 아닙니다. 일반적으로 decltype은 예상한 유형을 생성합니다.

이것은 decltype이 이름에 적용될 때 특히 사실입니다. 그 경우 decltype은 그 이름의 선언된 유형을 보고하는 것처럼 들리기 때문입니다.

기억해야 할 사항

- decltype은 거의 항상 수정 없이 변수 또는 표현식의 유형을 생성합니다.
- 이름이 아닌 유형 T의 lvalue 표현식의 경우 decltype은 항상 다음을 보고합니다.
T&의 종류.
- C++14는 auto와 마찬가지로 이나설라이저에서 유형을 추론하지만 decltype 규칙을 사용하여 유형 추론을 수행하는 decltype(auto)을 지원합니다.

항목 4: 추론된 유형을 보는 방법을 알고 있습니다.

유형 추론 결과를 보기 위한 도구 선택은 정보를 원하는 소프트웨어 개발 프로세스 단계에 따라 다릅니다. 코드를 편집할 때 형식 추론 정보를 가져오고, 컴파일하는 동안 가져오고, 런타임에 가져오는 세 가지 가능성을 살펴보겠습니다.

IDE 편집기

IDE의 코드 편집기는 커서를 엔터티 위로 가져가는 것과 같은 작업을 수행할 때 프로그램 엔터티 유형(예: 변수, 매개 변수, 함수 등)을 표시하는 경우가 많습니다.

예를 들어 이 코드가 주어지면

```
const int 답변 = 42;
```

```
자동 x = 답변; 자동 y =
&theAnswer;
```

IDE 편집기는 x의 추론된 유형이 int이고 y가 const임을 표시할 가능성이 높습니다.
정수*.

이것이 작동하려면 코드가 어느 정도 컴파일 가능한 상태에 있어야 합니다. IDE에서 이러한 종류의 정보를 제공할 수 있게 하는 것은 IDE 내부에서 실행되는 C++ 컴파일러(또는 최소한 프론트 엔드)이기 때문입니다. 해당 컴파일러가 코드를 구문 분석하고 유형 추론을 수행하기에 충분히 이해하지 못한다면 추론한 유형을 보여줄 수 없습니다.

int와 같은 간단한 유형의 경우 IDE의 정보는 일반적으로 괜찮습니다. 그러나 곧 보게 되겠지만 더 복잡한 유형이 포함되면 IDE에서 표시하는 정보가 특히 도움이 되지 않을 수 있습니다.

컴파일러 진단 컴파일러

가 추론한 유형을 표시하도록 하는 효과적인 방법은 컴파일 문제를 일으키는 방식으로 해당 유형을 사용하는 것입니다. 문제를 보고하는 오류 메시지는 문제를 일으키는 유형을 언급하는 것이 거의 확실합니다.

예를 들어 이전 예에서 x와 y에 대해 추론된 유형을 보고 싶다고 가정합니다. 먼저 정의하지 않은 클래스 템플릿을 선언합니다. 다음과 같이 잘 됩니다.

```
template<typename T> 클 // TD에 대해서만 선언;
래스 TD; // TD == "타입 디스플레이어"
```

이 템플릿을 인스턴스화하려고 하면 인스턴스화할 템플릿 정의가 없기 때문에 오류 메시지가 표시됩니다. x 및 y의 유형을 보려면 해당 유형으로 TD를 인스턴스화하십시오.

```
TD<decltype(x)> xType; // x 및 y 유형을 포함하는 // 오류 발생
TD<decltype(y)> yType;
```

내가 찾고 있는 정보를 찾는 데 도움이 되는 오류 메시지를 생성하는 경향이 있기 때문에 나는 variableNameType 형식의 변수 이름을 사용합니다. 위 코드의 경우 내 컴파일러 중 하나가 부분적으로 다음과 같이 진단 읽기를 실행합니다(우리가 찾는 유형 정보를 강조 표시했습니다).

오류: 집계 'TD<int> xType'에 불완전한 유형이 있으며
정의할 수 없다

오류: 집계 'TD<const int *> yType'의 유형이 불완전하여 정의할 수 없습니다.

다른 컴파일러는 동일한 정보를 제공하지만 다른 형식으로 제공합니다.

오류: 'xType'은 정의되지 않은 클래스 'TD<int>'를 사용합니다. 오류:
'yType'은 정의되지 않은 클래스 'TD<const int *>'를 사용합니다.

형식의 차이를 제외하고 테스트한 모든 컴파일러는 이 기술을 사용할 때 유용한 유형 정보가 포함된 오류 메시지를 생성합니다.

런타임 출력

유형 정보를 표시하는 printf 접근 방식(printf 사용을 권장하지 않음)은 런타임까지 사용할 수 없지만 출력 형식을 완전히 제어할 수 있습니다. 문제는 관심 있는 유형의 표시에 적합한 텍스트 표현을 만드는 것입니다. "땀 흘리지 마세요." "유형 ID와 std::type_info::name이 구조를 위한 것입니다."라고 생각합니다. x와 y에 대해 추론된 유형을 확인하기 위한 계속되는 탐구에서 다음과 같이 작성할 수 있다고 생각할 수 있습니다.

```
std::cout << typeid(x).name() << '\n'; std::cout << // x 및 y의 // 표시 유형
typeid(y).name() << '\n';
```

이 접근 방식은 x 또는 y와 같은 객체에서 typeid를 호출하면 std::type_info 객체가 생성되고 std::type_info에는 C 스타일 문자열(즉, const char *) 유형 이름의 표현.

std::type_info::name에 대한 호출은 적절한 반환을 보장하지 않지만 구현은 도움이 됩니다. 도움의 정도가 다릅니다. 예를 들어, GNU 및 Clang 컴파일러는 x의 유형이 "i"이고 y의 유형이 "PKi"라고 보고합니다. 이러한 결과는 이러한 컴파일러의 출력에서 "i"가 "int"를 의미하고 "PK"가 "konst const에 대한 포인터"를 의미한다는 것을 알게 되면 의미가 있습니다. (두 컴파일러 모두 이러한 " 맹글링된" 유형을 디코딩하는 도구인 c++filt를 지원합니다.) Microsoft 컴파일러는 x의 경우 "int" 및 y의 경우 "int const *"와 같이 덜 복잡한 출력을 생성합니다.

이러한 결과는 x 및 y 유형에 대해 정확하기 때문에 유형 보고 문제가 해결된 것으로 보고 싶은 유혹을 받을 수 있지만 조급해하지는 맙시다. 더 복잡한 예를 고려하십시오.

```
template<typename T> 무 // // 호출할 템플릿 함수
효 f(const T& param);
```



```
std::vector<위젯> createVec(); // 팩토리 함수
```



```
const 자동 vw = createVec(); // 초기화 vw w/공장 반환
```



```
if (!vw.empty())
    { f(&vw[0]); // f를 호출
    ...
}
```

사용자 정의 유형(위젯), STL 컨테이너(std::vector) 및 자동 변수(vw)를 포함하는 이 코드는 컴파일러에서 유형에 대한 가시성을 원할 수 있는 상황을 보다 잘 나타냅니다. 추론하고 있습니다. 예를 들어, 템플릿 유형 매개변수 T와 f의 함수 매개변수 param에 대해 어떤 유형이 유추되는지 아는 것이 좋습니다.

문제에서 typeid를 잃는 것은 간단합니다. 보고 싶은 유형을 표시하려면 f에 몇 가지 코드를 추가하기만 하면 됩니다.

```
template<typename T> 무
효 f(const T& param) { using
std::cout;
```

```

cout << "T =           " << typeid(T).이름() << '\n';           // T 표시
cout << "매개변수 =      " << typeid(param).이름() << '\n'; // 표시 // 매개변수의 // 유형
...
}

}

```

GNU 및 Clang 컴파일러에서 생성된 실행 파일은 다음 출력을 생성합니다.

```

T=          PK6Widget 매
개변수=PK6Widget

```

우리는 이미 이러한 컴파일러에서 PK가 "const에 대한 포인터"를 의미한다는 것을 알고 있으므로 유일한 미스터라는 숫자 6입니다. 그것은 바로 뒤에 오는 클래스 이름의 문자 수입니다(Widget). 따라서 이러한 컴파일러는 T와 param이 모두 const Widget* 유형임을 알려줍니다.

Microsoft의 컴파일러는 다음과 같이 동의합니다.

```

T=          클래스 위젯 const * param = 클래
스 위젯 const *

```

동일한 정보를 생성하는 세 개의 독립적인 컴파일러는 정보가 정확함을 나타냅니다. 그러나 더 자세히 보십시오. 템플릿 f에서 param의 선언된 유형은 const T&입니다. 그렇다면 T와 param이 같은 유형을 갖는 것이 이상하지 않습니까? 예를 들어 T가 int인 경우 param의 유형은 const int&여야 합니다. 동일한 유형이 아닙니다.

슬프게도 std::type_info::name의 결과는 신뢰할 수 없습니다. 이 경우, 예를 들어 세 컴파일러가 모두 param에 대해 보고하는 유형이 올바르지 않습니다. 게다가, std::type_info::name에 대한 사용은 형식이 값에 의한 매개변수로 템플릿 함수에 전달된 것처럼 처리되도록 요구하기 때문에 본질적으로 올바르지 않아야 합니다. **항목 1**에서 설명하는 것처럼 유형이 참조이면 해당 참조성을 무시하고 참조 제거 후 유형이 const(또는 휘발성)이면 해당 constness(또는 휘발성)도 무시된다는 의미입니다. 이것이 const Widget * const &인 param의 유형이 const Widget*으로 보고되는 이유입니다.

먼저 유형의 참조성이 제거된 다음 결과 포인터의 일관성이 제거됩니다.

마찬가지로 슬프게도 IDE 편집기에 의해 표시되는 유형 정보도 신뢰할 수 없거나 적어도 안정적으로 유용하지 않습니다. 이 동일한 예에서 내가 아는 한 IDE 편집기는 T의 유형을 다음과 같이 보고합니다(나는 이것을 만들지 않습니다).

```

상수
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<위젯, std::allocator<위젯
>::_Alloc>::value_type>::value_type *

```

동일한 IDE 편집기는 param의 유형을 다음과 같이 표시합니다.

```
const std::Simple_types<...>::value_type *const &
```

이것은 T의 유형보다 덜 위협적이지만 중간에 있는 "...는 "T 유형의 일부인 모든 내용을 생략하고 있습니다."라고 말하는 IDE 편집기의 방식임을 깨닫기 전까지는 혼란스럽습니다. 운이 좋으면 개발 환경이 이와 같은 코드에서 더 나은 작업을 수행합니다.

운보다 라이브러리에 더 의존하는 경향이 있다면 std::type_info::name 및 IDE가 실패할 수 있는 곳에서 Boost TypeIndex 라이브러리(종종 Boost.TypeIndex로 작성됨)가 성공하도록 설계되었다는 사실을 알게 되어 기쁩니다. . 라이브러리는 표준 C++의 일부가 아니지만 IDE나 TD와 같은 템플릿도 아닙니다. 또한 Boost 라이브러리([boost.com에서 사용 가능](#)) 크로스 플랫폼, 오픈 소스, 가장 편집증적인 기업 범무팀도 입맛에 맞도록 설계된 라이선스에 따라 사용할 수 있다는 것은 Boost 라이브러리를 사용하는 코드가 표준 라이브러리에 의존하는 코드만큼 이식성이 높다는 것을 의미합니다.

다음은 Boost.Type-Index를 사용하여 함수 f가 정확한 유형 정보를 생성하는 방법입니다.

```
#include <부스트/유형_색인.hpp>

template<typename T> 무
효 f(const T& param) { using
std::cout;

boost::typeindex::type_id_with_cvr 사용;

// T를 표시
cout << "T = "
    << type_id_with_cvr<T>().pretty_name() << '\n';

// param의 유형 표시 cout<
<< "param = <<
    type_id_with_cvr<decltype(param)>().pretty_name() << '\n';

...
}
```

이것이 작동하는 방식은 함수 템플릿 boost::typeindex:: type_id_with_cvr이 유형 인수(정보를 원하는 유형)를 취하고 const, volatile 또는 참조 한정자를 제거하지 않는다는 것입니다(따라서 "with_cvr" 템플릿 이름). 그 결과는 유형의 인간 친화적인 표현을 포함하는 std::string을 생성하는 pretty_name 멤버 함수가 있는 boost::typeindex::type_index 개체입니다.

f에 대한 이 구현에서는 typeid가 사용될 때 param에 대한 잘못된 유형 정보를 생성하는 호출을 다시 고려하십시오.

```
std::vector<위젯> createVec(); // 팩토리 함수

const 자동 vw = createVec(); // 초기화 vw w/공장 반환

if (!vw.empty()) { f(&vw[0]); // f를 호출
...
}
```

GNU 및 Clang의 컴파일러에서 Boost.TypeIndex는 다음과 같은 (정확한) 출력을 생성합니다.

```
티 = 위젯 const* param =
위젯 const* const&
```

Microsoft 컴파일러의 결과는 기본적으로 동일합니다.

```
티 = 클래스 위젯 const * param = 클래
스 위젯 const * const &
```

이러한 거의 균일성은 좋지만 IDE 편집기, 컴파일러 오류 메시지 및 Boost.TypeIndex와 같은 라이브러리는 컴파일러가 추론하는 유형을 파악하는 데 사용할 수 있는 도구일 뿐임을 기억하는 것이 중요합니다. 모두 도움이 될 수 있지만 결국 항목 **1-3의 유형 연역 정보를 이해하는 것을 대체할 수 없습니다.**

기억할 사항 • 추론된

유형은 종종 IDE 편집기, 컴파일러 오류 메시지 및 Boost TypeIndex 라이브러리를 사용하여 볼 수 있습니다.

- 일부 도구의 결과는 도움이 되지도 정확하지도 않을 수 있으므로 C++의 유형 연역 규칙에 대한 이해는 여전히 필수적입니다.

제 2 장

자동

개념적으로 `auto`는 단순할 수 있는 만큼 간단하지만 보기보다 더 미묘합니다.

이를 사용하면 타이핑을 절약할 수 있지만 수동 유형 선언을 악화시킬 수 있는 정확성 및 성능 문제도 방지할 수 있습니다. 게다가, `auto`의 타입 연역 결과 중 일부는 규정된 알고리즘을 충실히 준수하지만 프로그래머의 관점에서 볼 때 잘못된 것입니다. 이 경우 수동 유형 선언으로 되돌아가는 것이 가장 피하는 대안이기 때문에 `auto`를 올바른 답으로 안내하는 방법을 아는 것이 중요합니다.

이 짧은 장에서는 자동의 모든 기능을 다룹니다.

항목 5: 명시적 형식 선언 보다 `auto` 를 선호 합니다.

아, 단순한 기쁨

정수 `x`;

기다리다. 젠장. `x`를 초기화하는 것을 잊었으므로 그 값이 불확실합니다. 아마도. 실제로 0으로 초기화될 수 있습니다. 컨텍스트에 따라 다릅니다. 한숨을 쉬다.

괜찮아요. 반복자를 역참조하여 초기화할 지역 변수를 선언하는 간단한 즐거움으로 넘어갑시다.

```
template<typename It> void      // dwim에 대한 알고리즘("do what I mean") // b에서 e까
dwim(It b, It e) { while (b !=      지 범위의 모든 요소에 대해 //
e) {
```

```
유형 이름 std::iterator_traits<It>::value_type
curr값 = *b;
...
```

```
 } }
```

어. "typename std::iterator_traits<It>::value_type"은 반복자가 가리키는 값의 유형을 표현하기 위해? 진짜? 얼마나 즐거웠는지 기억을 차단했을 것입니다. 젠장. 잠깐, 내가 이미 말하지 않았습니까?

좋아요, 간단한 기쁨(3개): 유형이 클로저 유형인 지역 변수를 선언하는 기쁨입니다. 오, 맞아. 클로저의 유형은 컴파일러만 알고 있으므로 쓸 수 없습니다. 한숨을 쉬다. 젠장.

젠장, 젠장, 젠장! C++ 프로그래밍은 즐거운 경험이 아닙니다!

음, 예전에는 없었습니다. 그러나 C++11부터는 auto 덕분에 이러한 모든 문제가 사라집니다. auto 변수는 이나설라이저에서 유형이 추론되므로 초기화해야 합니다. 즉, 최신 C++ 고속도로에서 속도를 내면서 초기화되지 않은 변수 문제에 작별을 고할 수 있습니다.

정수 x1;	// 잠재적으로 초기화되지 않음
자동 x2;	// 오류! 이나설라이저 필요
자동 x3 = 0;	// 좋습니다, x의 값은 잘 정의되어 있습니다

상기 고속도로에는 역참조 반복자의 값을 갖는 지역 변수를 선언하는 것과 관련된 움푹 들어간 곳이 없습니다:

```
template<typename It> void           // 이전과
dwim(It b, It e) { while (b != e)
{ auto currValue = *b;
```

...

```
 }}
```

그리고 auto는 형식 추론을 사용하기 때문에(항목 2 참조) 컴파일러에게만 알려진 형식을 나타낼 수 있습니다.

```
자동 derefUPless =           // 비교 함수. [](const
std::unique_ptr<Widget>& p1, // 위젯용
const std::unique_ptr<Widget>& p2) // { return *p1 < *p2; };
// std::unique_ptrs
```

아주 멋져요. C++14에서는 람다 표현식에 대한 매개변수가 auto를 포함할 수 있기 때문에 온도가 더 떨어집니다.

```
자동 derefless =           // C++14 비교 // 함수
[](const auto& p1,
```

```
const auto& p2) { 반 // 포인터와 유사한 // 모든
    환 *p1 < *p2; }; 항목 이 가리키는 값
```

줄함에도 불구하고, 아마도 당신은 우리가 `std::function` 객체를 사용할 수 있기 때문에 클로저를 보유하는 변수를 선언하기 위해 `auto`가 정말로 필요하지 않다고 생각할 수도 있습니다. 그것은 사실입니다. 우리는 할 수 있지만 아마도 그것은 당신이 생각한 것이 아닐 것입니다. 그리고 아마도 지금 당신은 "`std::function` 객체가 무엇인가?"라고 생각하고 있을지도 모릅니다. 그래서 그것을 명확히 하자.

`std::function`은 함수 포인터의 개념을 일반화하는 C++11 표준 라이브러리의 템플릿입니다. 함수 포인터는 함수만 가리킬 수 있지만 `std::function` 객체는 호출 가능한 모든 객체, 즉 함수처럼 호출될 수 있는 모든 객체를 참조할 수 있습니다. 함수 포인터를 생성할 때 가리킬 함수의 유형을 지정해야 하는 것처럼(즉, 가리키고 싶은 함수의 서명) `std::function`을 생성할 때 참조할 함수 유형을 지정해야 합니다. 함수 자체. `std::function`의 템플릿 매개변수를 통해 이를 수행합니다.

예를 들어, 이 서명이 있는 것처럼 작동하는 호출 가능한 모든 객체를 참조할 수 있는 `func`라는 `std::function` 객체를 선언하려면,

```
bool(const std::unique_ptr<Widget>&, // C++11 서명
      const std::unique_ptr<Widget>&) // std::unique_ptr<Widget> // 비교 함수
```

당신은 이것을 쓸 것입니다 :

```
std::function<bool(const std::unique_ptr<Widget>&, const
                    std::unique_ptr<Widget>&)> func;
```

람다 식은 호출 가능한 객체를 생성하기 때문에 클로저는 `std::function` 객체에 저장할 수 있습니다. 즉, 다음과 같이 `auto`를 사용하지 않고 C++11 버전의 `derefUPLess`를 선언할 수 있습니다.

```
std::function<bool(const std::unique_ptr<Widget>&, const
                    std::unique_ptr<Widget>&)> derefUPLess
= [](const std::unique_ptr<Widget>& p1, const std::unique_ptr<Widget>&
      p2) { 반환 *p1 < *p2; };
```

구문상의 장황함을 제쳐두고 매개변수 유형을 반복해야 하는 경우에도 `std::function`을 사용하는 것은 `auto`를 사용하는 것과 같지 않다는 것을 인식하는 것이 중요합니다. 클로저를 보유하는 자동 선언 변수는 클로저와 유형이 동일하므로 클로저에 필요한 만큼만 메모리를 사용합니다. 클로저를 보유하는 `std::function` 선언 변수의 유형은 `std::function` 템플릿의 인스턴스화이며 주어진 서명에 대해 고정된 크기를 갖습니다. 이 크기는 저장하도록 요청된 클로저에 적합하지 않을 수 있으며, 이 경우 `std::function` 생성자는 클로저를 저장하기 위해 힙 메모리를 할당합니다. 그 결과는

`std::function` 객체는 일반적으로 자동 선언된 객체보다 더 많은 메모리를 사용합니다. 그리고 인라인을 제한하고 간접 함수 호출을 생성하는 구현 세부 사항 덕분에 `std::function` 객체를 통해 클로저를 호출하는 것이 자동 선언된 객체를 통해 호출하는 것보다 느릴 것이 거의 확실합니다. 즉, `std::function` 접근 방식은 일반적으로 자동 접근 방식보다 크고 느리며 메모리 부족 예외도 발생할 수 있습니다. 또한 위의 예에서 볼 수 있듯이 "auto"를 작성하는 것은 `std::function` 인스턴스화 유형을 작성하는 것보다 훨씬 적은 작업입니다. 클로저를 유지하기 위한 `auto` 와 `std::function` 간의 경쟁에서 `auto`는 거의 게임, 설정 및 일치입니다. (`std::bind`에 대한 호출 결과를 유지하기 위해 `std::function`보다 `auto`에 대해 비슷한 주장을 할 수 있지만, 항목 34에서는 `std::bind` 대신 람다를 사용하도록 최선을 다 합니다. 방법.)

`auto`의 장점은 초기화되지 않은 변수, 장황한 변수 선언 및 클로저를 직접 보유하는 기능을 피하는 것 이 상으로 확장됩니다. 하나는 내가 "타입 단축기"와 관련된 문제라고 부르는 것을 피할 수 있는 능력입니다. 다음은 아마도 본 적이 있을 것입니다. 아마도 다음과 같습니다.

```
표준::벡터<int> v;
...
부호 없는 sz = v.size();
```

`v.size()`의 공식 반환 유형은 `std::vector<int>::size_type`이지만 이를 알고 있는 개발자는 거의 없습니다. `std::vector<int>::size_type`은 `unsigned` 정수형으로 지정되어 있기 때문에 많은 프로그래머들이 `unsigned`로도 충분하다고 생각하고 위와 같은 코드를 작성한다. 이것은 몇 가지 흥미로운 결과를 초래 할 수 있습니다. 예를 들어 32비트 Windows에서는 `unsigned`와 `std::vector<int>::size_type`이 같은 크기이지만 64비트 Windows에서는 `unsigned`가 32비트인 반면 `std::vector<int>::size_type`은 64비트입니다. 즉, 32비트 Windows에서 작동하는 코드가 64비트 Windows에서 잘못 작동할 수 있으며, 32비트에서 64비트로 응용 프로그램을 이식할 때 누가 그런 문제에 시간을 할애하고 싶겠습니까?

`auto`를 사용하면 다음을 수행할 필요가 없습니다.

자동 sz = v.size(); // sz의 유형은 `std::vector<int>::size_type`입니다.

`auto` 사용의 저해에 대해 아직도 확신이 서지 않습니까? 그런 다음 다음 코드를 고려하십시오.

```
std::unordered_map<std::string, int> m;
...
for (const std::pair<std::string, int>& p : m) {
    ...
    // p로 원자를 한다
}
```

이것은 매우 합리적으로 보이지만 문제가 있습니다. 당신은 그것을 볼 수 있습니까?

무엇이 잘못되었는지 인식하려면 std::unordered_map의 핵심 부분이 const이므로 해시 테이블의 std::pair 유형(std::unordered_map이 있음)이 std::pair<가 아니라는 것을 기억해야 합니다. std::string, int>, std::pair <const std::string, int>입니다. 그러나 그것은 위의 루프에서 변수 p에 대해 선언된 유형이 아닙니다. 결과적으로 컴파일러는 std::pair<const std::string, int> 객체(즉, 해시 테이블에 있는 내용)를 std::pair<std::string, int> 객체로 변환하는 방법을 찾기 위해 노력할 것입니다. (p에 대해 선언된 유형). 그들은 m에 있는 각 객체를 복사하여 p가 바인딩하려는 유형의 임시 객체를 만든 다음 참조 p를 해당 임시 객체에 바인딩함으로써 성공할 것입니다. 각 루프 반복이 끝나면 임시 객체가 소멸됩니다. 이 루프를 작성했다면 이 동작에 놀랄 것입니다. 거의 확실히 참조 p를 m의 각 요소에 바인딩하려고 하기 때문입니다.

이러한 의도하지 않은 유형 불일치는 자동으로 제거될 수 있습니다.

```
for (const auto& p : m) {  
    ...  
    } // 이전과
```

이것은 더 효율적일 뿐만 아니라 입력하기도 더 쉽습니다. 게다가, 이 코드는 p의 주소를 취하면 m 내의 요소에 대한 포인터를 확실히 얻을 수 있다는 매우 매력적인 특성을 가지고 있습니다. auto를 사용하지 않는 코드에서는 임시 객체에 대한 포인터를 얻게 됩니다. 이 객체는 루프 반복이 끝나면 소멸됩니다.

마지막 두 가지 예 - std::vector<int>::size_type을 작성해야 할 때 unsigned를 작성하고 std::pair<const std>를 작성했어야 할 때 std::pair<std::string, int>를 작성: ::string, int> - 유형을 명시적으로 지정하면 원하지도 기대하지도 않는 암시적 변환이 발생할 수 있음을 보여줍니다. 대상 변수의 유형으로 auto를 사용하는 경우 선언하는 변수의 유형과 이를 초기화하는 데 사용된 표현식 유형 간의 불일치에 대해 걱정할 필요가 없습니다.

따라서 명시적 유형 선언보다 auto를 선호하는 몇 가지 이유가 있습니다. 그러나 자동은 완벽하지 않습니다. 각 자동 변수의 유형은 초기화 표현식에서 추론되며 일부 초기화 표현식에는 예상하지도 원하지도 않는 유형이 있습니다. 그러한 경우가 발생하는 조건과 이에 대해 할 수 있는 조치는 항목 2 와 6에서 논의되므로 여기서 다루지 않겠습니다. 대신, 나는 전통적인 유형 선언 대신 auto를 사용하는 것과 관련하여 가질 수 있는 다른 문제, 즉 결과 소스 코드의 가독성에 대해 관심을 돌릴 것입니다.

먼저 심호흡을 하고 긴장을 풀어주세요. `auto`는 선택사항이지 의무사항이 아닙니다. 전문적인 판단에 따라 명시적 유형 선언을 사용하여 코드가 더 명확하거나 유지 관리가 용이하거나 다른 방식으로 더 좋았으면 계속 사용할 수 있습니다. 그러나 C++는 프로그래밍 언어 세계에서 일반적으로 형식 유추로 알려진 것을 채택하는 데 새로운 지평을 열지 않는다는 점을 명심하십시오. 다른 정적으로 유형이 지정된 절차 언어(예: C#, D, Scala, Visual Basic)는 다양한 정적으로 유형이 지정된 기능 언어(예: ML, Haskell, OCaml, F# 등)는 말할 것도 없이 거의 동등한 기능을 가지고 있습니다. . 부분적으로는 변수가 명시적으로 입력되는 경우가 거의 없는 Perl, Python 및 Ruby와 같은 동적 형식 언어의 성공에 기인합니다. 소프트웨어 개발 커뮤니티는 유형 유추에 대한 광범위한 경험을 가지고 있으며 그러한 기술과 산업 강도의 대규모 코드 기반 생성 및 유지 관리에 모순이 없음을 입증했습니다.

일부 개발자는 `auto`를 사용하면 소스 코드를 한 눈에 보고 개체 유형을 결정할 수 있는 기능이 제거된다는 사실에 불안해합니다. 그러나 객체 유형을 표시하는 IDE의 기능은 종종 이 문제를 완화하며(심지어 [항목 4](#)에서 언급한 IDE 유형 표시 문제를 고려하더라도), 많은 경우에 객체 유형에 대한 다소 추상적인 관점도 마찬가지로 유용합니다. 정확한 유형으로. 예를 들어, 컨테이너, 카운터 또는 스마트 포인터의 종류를 정확히 알지 못해도 개체가 컨테이너, 카운터 또는 스마트 포인터라는 것을 아는 것으로 충분합니다.

잘 선택된 변수 이름을 가정하면 그러한 추상 유형 정보는 거의 항상 가까이에 있어야 합니다.

문제의 사실은 유형을 작성하는 것이 종종 정확성이나 효율성 또는 둘 다에서 미묘한 오류의 기회를 명시적으로 소개하는 것 이상을 하지 않는다는 것입니다. 또한 `auto` 유형은 초기화 표현식의 유형이 변경되면 자동으로 변경되며, 이는 `auto`를 사용하여 일부 리팩토링이 용이함을 의미합니다. 예를 들어 함수가 `int`를 반환하도록 선언되었지만 나중에 `long`이 더 낫다고 결정하면 함수 호출 결과가 `auto` 변수에 저장되어 있으면 다음에 컴파일할 때 호출 코드가 자동으로 업데이트됩니다. 결과가 명시적으로 `int`로 선언된 변수에 저장되는 경우 수정할 수 있도록 모든 호출 사이트를 찾아야 합니다.

기억할 사항 • `auto` 변수

수는 초기화되어야 하고 일반적으로 이식성 또는 효율성 문제로 이어질 수 있는 유형 불일치에 영향을 받지 않으며 리팩토링 프로세스를 용이하게 할 수 있으며 일반적으로 명시적으로 지정된 유형이 있는 변수보다 입력이 덜 필요합니다.

- 자동 유형 변수는 항목 2 및 6에 설명된 함정의 영향을 받습니다.

항목 6: 자동으로 원하지 않는 유형을 추론 할 때 명시적으로 유형이 지정된 이니셜라이저 관용구를 사용하십시오 .

항목 5 는 변수를 선언하기 위해 `auto`를 사용하는 것이 명시적으로 유형을 지정하는 것보다 많은 기술적 이점을 제공한다고 설명하지만, 때로는 `auto`의 유형 추론을 지그재그로 원할 때 지그재그로 만듭니다. 예를 들어 위젯을 사용하고 `std::vector<bool>`을 반환하는 함수가 있다고 가정합니다. 여기서 각 `bool`은 위젯이 특정 기능을 제공하는지 여부를 나타냅니다.

```
std::vector<bool> features(const Widget& w);
```

또한 비트 5가 위젯의 우선 순위가 높은지 여부를 표시한다고 가정합니다. 따라서 다음과 같은 코드를 작성할 수 있습니다.

```
위젯 w;
```

```
...
```

```
bool highPriority = 가능(w)[5]; // w가 높은 우선 순위입니까?
```

```
...
```

```
processWidget(w, highPriority);
```

// 우선 순위에 따라 // w를 처리합니다.

이 코드에는 아무런 문제가 없습니다. 그것은 잘 작동합니다. 그러나 `highPriority`의 명시적 유형을 `auto`로 바꾸는 것처럼 곁보기며 무해해 보이는 변경을 수행하면

```
auto highPriority = 가능(w)[5]; // w가 높은 우선 순위입니까?
```

상황이 바뀝니다. 모든 코드는 계속 컴파일되지만 동작은 더 이상 예측할 수 없습니다.

```
processWidget(w, highPriority);
```

// 정의되지 않은 동작!

주석에서 알 수 있듯이 `processWidget`에 대한 호출에는 정의되지 않은 동작이 있습니다.

하지만 왜? 대답은 아마 놀랄 것입니다. `auto`를 사용하는 코드에서 `highPriority` 유형은 더 이상 `bool`이 아닙니다. `std::vector<bool>`은 개념적으로 `bool`을 보유하지만 `std::vector<bool>`에 대한 `operator[]`는 컨테이너의 요소에 대한 참조를 반환하지 않습니다(이는 `std::vector::operator[]`가 반환하는 것입니다. `bool`을 제외한 모든 유형). 대신 `std::vector<bool>::reference` 유형의 객체를 반환합니다(`std::vector<bool>` 내부에 중첩된 클래스).

`std::vector<bool>::reference`가 존재하는 이유는 `std::vector<bool>`이 해당 `bool`을 `bool`당 1비트로 압축된 형태로 나타내도록 지정되었기 때문입니다. `std::vector<T>`에 대한 `operator[]`는 `T&`를 반환해야 하지만 C++에서는 비트에 대한 참조를 금지하기 때문에 `std::vector<bool>`의 `operator[]`에 문제가 발생합니다. 되돌릴 수 없는

`bool&, std::vector<bool>`에 대한 `operator[]`는 `bool&`처럼 작동하는 객체를 반환합니다. 이 작업이 성공하려면 `std::vector<bool>::reference` 객체를 `bool&s`가 있을 수 있는 모든 컨텍스트에서 기본적으로 사용할 수 있어야 합니다. 이 작업을 수행하는 `std::vector<bool>::reference`의 기능 중에는 `bool`로의 암시적 변환이 있습니다. (`bool&`이 아니라 `bool`입니다. `bool&`의 동작을 에뮬레이트하기 위해 `std::vector<bool>::reference`에서 사용하는 전체 기술 세트를 설명하려면 너무 멀리 떨어져 있어야 합니다. 암시적 변환은 더 큰 모자이크에서 하나의 돌일 뿐입니다.)

이 정보를 염두에 두고 원본 코드의 이 부분을 다시 살펴보세요.

```
bool highPriority = 기능(w)[5]; // highPriority의 // 유형을 명시적으로 선언
```

여기서 기능은 `operator[]`가 호출되는 `std::vector<bool>` 객체를 반환합니다. `operator[]`는 `std::vector<bool>::reference` 객체를 반환하며, 이는 암시적으로 `highPriority`를 초기화하는 데 필요한 `bool`로 변환됩니다. 따라서 높은 우선 순위는 예상대로 기능에 의해 반환된 `std::vector<bool>`의 비트 5 값으로 끝납니다.

`highPriority`에 대한 자동 선언에서 일어나는 일과 대조하십시오.

```
auto highPriority = 기능(w)[5]; // highPriority를 추론  
// 유형
```

다시, 기능은 `std::vector<bool>` 개체를 반환하고 다시 `operator[]`가 이 개체에서 호출됩니다. `operator[]`는 계속해서 `std::vector<bool>::reference` 객체를 반환하지만, `auto`는 이를 `highPriority`의 유형으로 추론하기 때문에 이제 변경 사항이 있습니다. `highPriority`에는 기능에서 반환된 `std::vector<bool>`의 비트 5 값이 전혀 없습니다.

값은 `std::vector<bool>::reference`가 구현되는 방식에 따라 다릅니다. 한 가지 구현은 이러한 객체가 참조된 비트를 보유하는 기계어에 대한 포인터와 해당 비트에 대한 해당 단어에 대한 오프셋을 포함하는 것입니다. 그러한 `std::vector<bool>::reference` 구현이 제자리에 있다고 가정하고 이것이 `highPriority`의 초기화에 대해 의미하는 바를 고려하십시오.

기능에 대한 호출은 임시 `std::vector<bool>` 객체를 반환합니다. 이 개체에는 이름이 없지만 이 토론의 목적을 위해 `temp`라고 합니다. `operator[]`는 `temp`에서 호출되고 반환되는 `std::vector<bool>::reference`에는 `temp`에 의해 관리되는 비트를 보유하는 데이터 구조의 단어에 대한 포인터와 비트에 해당하는 해당 단어에 대한 오프셋이 포함됩니다. 5. `highPriority`는 이 `std::vector<bool>::reference` 객체의 복사본이므로 `highPriority`도 `temp`의 단어에 대한 포인터와 비트 5에 해당하는 오프셋을 포함합니다. 명령문의 끝에서,

`temp` 는 임시 객체이기 때문에 소멸됩니다. 따라서 `highPriority`에는 매달린 포인터가 포함되어 있으며 이것이 `processWidget` 호출에서 정의되지 않은 동작의 원인입니다.

```
processWidget(w, highPriority); // 정의되지 않은 동작!
                                // highPriority는 // 덩글링 포인터를 포함합니다!
```

`std::vector<bool>::reference`는 프록시 클래스의 한 예입니다. 다른 유형의 동작을 에뮬레이트하고 보강할 목적으로 존재하는 클래스입니다. 프록시 클래스는 다양한 목적으로 사용됩니다.

`std::vector<bool>::reference`는 예를 들어 `std::vector<bool>`에 대한 `operator[]`가 비트에 대한 참조와 표준 라이브러리의 스마트 포인터 유형을 반환한다는 **한상**을 제공하기 위해 존재합니다. **ter 4)**는 리소스 관리를 원시 포인터에 접목하는 프록시 클래스입니다. 프록시 클래스의 유용성은 잘 확립되어 있습니다. 사실, 디자인 패턴 "프록시"는 소프트웨어 디자인 패턴 Pantheon의 가장 오래 된 구성원 중 하나입니다.

일부 프록시 클래스는 클라이언트에 명확하게 표시되도록 설계되었습니다. 예를 들어 `std::shared_ptr` 및 `std::unique_ptr`의 경우입니다. 다른 프록시 클래스는 다소 눈에 띄지 않게 작동하도록 설계되었습니다. `std::vector<bool>::reference`는 `std::bitset` 동포인 `std::bitset::ref`와 마찬가지로 "보이지 않는" 프록시의 예입니다.

에렌스.

또한 해당 캠프에는 표현식 템플릿으로 알려진 기술을 사용하는 C++ 라이브러리의 일부 클래스가 있습니다. 이러한 라이브러리는 원래 숫자 코드의 효율성을 향상시키기 위해 개발되었습니다. 예를 들어, `Matrix` 및 `Matrix` 객체 `m1, m2, m3` 및 `m4` 클래스가 주어지면 표현식

`행렬 합 = m1 + m2 + m3 + m4;`

`Matrix` 객체에 대한 `operator+`가 결과 자체 대신 결과에 대한 프록시를 반환하면 훨씬 더 효율적으로 계산할 수 있습니다. 즉, 두 `Matrix` 객체에 대한 `operator+`는 `Matrix` 객체 대신 `Sum<Matrix, Matrix>`와 같은 프록시 클래스의 객체를 반환합니다. `std::vector<bool>::reference` 및 `bool`의 경우와 마찬가지로 프록시 클래스에서 `Matrix`로의 암시적 변환이 발생하여 오른쪽 표현식에 의해 생성된 프록시 객체에서 합계의 초기화를 하용합니다. "="의 측면. (그 객체의 유형은 전통적으로 전체 초기화 표현식을 인코딩합니다. 즉, `Sum<Sum<Sum<Matrix, Matrix>, Matrix>`, `Matrix>`와 같은 것입니다. 이것은 확실히 클라이언트가 보호되어야 하는 유형입니다.)

일반적으로 "보이지 않는" 프록시 클래스는 `auto`에서 잘 작동하지 않습니다. 그러한 클래스의 객체는 종종 단일 명령문보다 오래 지속되도록 설계되지 않으므로 이러한 유형의 변수를 생성하는 것은 기본적인 라이브러리 설계 가정을 위반하는 경향이 있습니다. 그건

`std::vector<bool>::reference`의 경우, 그 가정을 위반하면 정의되지 않은 동작이 발생할 수 있음을 확인했습니다.

따라서 다음 형식의 코드를 피하고 싶습니다.

```
auto someVar = "보이지 않는" 프록시 클래스 유형의 표현;
```

그러나 프록시 개체가 사용 중임을 어떻게 알 수 있습니까? 그들을 사용하는 소프트웨어는 그들의 존재를 광고하지 않을 것입니다. 그것들은 적어도 개념적으로는 보이지 않아야 합니다! 그리고 일단 당신이 그것들을 찾았다면, 당신은 정말로 자동과 항목 5 가 그것에 대해 보여주는 많은 이점을 포기해야 합니까?

먼저 그들을 찾는 방법 질문을 합시다. "보이지 않는" 프록시 클래스는 일상적인 사용에서 프로그래머 레이더 아래로 날아가도록 설계되었지만, 이를 사용하는 라이브러리는 종종 그렇게 한다고 문서화합니다. 사용하는 라이브러리의 기본 설계 결정에 익숙해질수록 해당 라이브러리 내에서 프록시 사용으로 인해 눈이 멀어질 가능성이 줄어듭니다.

문서가 부족하면 헤더 파일이 공백을 채웁니다. 소스 코드가 프록시 개체를 완전히 숨길 수 있는 경우는 거의 없습니다. 일반적으로 클라이언트가 호출할 것으로 예상되는 함수에서 반환되므로 함수 서명은 일반적으로 존재를 반영합니다.

다음은 `std::vector<bool>::operator[]`에 대한 사양입니다. 예를 들면 다음과 같습니다.

<pre>네임스페이스 표준 { 템플릿 <클래스 할당자> 클래스 벡터 <bool, 할당자> { 공개: ... 클래스 참조 { ... }; 참조 연산자[](크기 유형 n); ... }; }</pre>	// C++ 표준에서
--	--

`std::vector<T>`에 대한 `operator[]`이 일반적으로 `T&`를 반환한다는 것을 알고 있다고 가정하면 이 경우 `operator[]`에 대한 비 전통적인 반환 유형은 프록시 클래스가 사용 중이라는 정보입니다. 사용 중인 인터페이스에 주의를 기울이면 종종 프록시 클래스의 존재가 드러날 수 있습니다.

실제로 많은 개발자는 복잡한 컴파일 문제를 추적하거나 잘못된 단위 테스트 결과를 디버그하려고 할 때만 프록시 클래스의 사용을 발견합니다.

그것들을 찾는 방법에 관계없이 `auto`가 프록시되는 유형 대신 프록시 클래스의 유형을 추론하는 것으로 결정되면 솔루션은 `auto`를 포기하는 것을 포함할 필요가 없습니다. 자동 자체가 문제가 아닙니다. 문제는 자동이 연역되지 않는다는 것입니다.

추론하려는 유형을 지정합니다. 해결책은 다른 유형 추론을 강제하는 것입니다.
그렇게 하는 방식을 저는 명시적으로 형식화된 이니셜라이저 관용구라고 부릅니다.

명시적으로 유형이 지정된 이니셜라이저 관용구에는 auto를 사용하여 변수를 선언하는 것이 포함되지만 auto가 추론하도록 하려는 유형으로 초기화 표현식을 캐스팅합니다. 예를 들어 highPriority를 부울로 만드는 데 사용할 수 있는 방법은 다음과 같습니다.

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

여기에서 features(w)[5]는 항상 그랬던 것처럼 std::vector<bool>::reference 객체를 계속 반환하지만 캐스트는 표현식의 유형을 bool로 변경하고 auto는 유형으로 추론합니다. 높은 우선 순위를 위해 런타임에 std::vector<bool>::operator[]에서 반환된 std::vector<bool>::reference 객체는 지원하는 bool로의 변환을 실행하고 해당 변환의 일부로 여전히 유효한 기능에서 반환된 std::vector<bool>에 대한 포인터는 역참조됩니다. 이는 우리가 이전에 겪었던 정의되지 않은 동작을 방지합니다. 인덱스 5는 다음과 같습니다.

포인터가 가리키는 비트에 적용되고 나오는 bool 값은 highPriority를 초기화하는 데 사용됩니다.

Matrix 예제의 경우 명시적으로 입력된 초기화 관용구는 다음과 같습니다.

```
자동 합계 = static_cast<매트릭스>(m1 + m2 + m3 + m4);
```

관용구의 적용은 프록시 클래스 유형을 생성하는 이니셜라이저에 국한되지 않습니다. 초기화 식에서 생성된 것과 다른 형식의 변수를 의도적으로 생성한다는 점을 강조하는 것도 유용할 수 있습니다. 예를 들어, 일부 허용 오차 값을 계산하는 함수가 있다고 가정합니다.

이중 calcEpsilon();	// 허용 오차 값 반환
-------------------	---------------

calcEpsilon은 분명히 double을 반환하지만 응용 프로그램에 float의 정밀도가 적절하고 float와 double 간의 크기 차이에 관심이 있다는 것을 알고 있다고 가정합니다. calcEpsilon의 결과를 저장하기 위해 float 변수를 선언할 수 있습니다.

float ep = calcEpsilon();	// 암시적으로 변환 // double → float
---------------------------	-------------------------------

그러나 이것은 "함수에 의해 반환된 값의 정밀도를 의도적으로 줄였습니다."라고 거의 알리지 않습니다. 그러나 명시적으로 형식화된 이니셜라이저 관용구를 사용하는 선언은 다음을 수행합니다.

```
자동 ep = static_cast<float>(calcEpsilon());
```

의도적으로 정수 값으로 저장하는 부동 소수점 표현식이 있는 경우에도 유사한 추론이 적용됩니다. 임의의 액세스 반복자를 사용하여 컨테이너에 있는 요소의 인덱스를 계산해야 한다고 가정합니다(예: std::vector, std::deque,

또는 `std::array`), 원하는 요소가 컨테이너의 시작 부분에서 얼마나 멀리 떨어져 있는지 나타내는 0.0과 1.0 사이의 `double`이 제공됩니다. (0.5는 컨테이너의 중간을 나타냅니다.) 또한 결과 인덱스가 `int`에 맞을 것이라고 확신한다고 가정합니다. 컨테이너가 `c`이고 `double`이 `d`이면 인덱스를 다음과 같이 계산할 수 있습니다.

정수 인덱스 = `d * c.size()`;

그러나 이것은 의도적으로 오른쪽의 `double`을 `int`로 변환한다는 사실을 흐리게 합니다. 명시적으로 유형이 지정된 이나셜라이저 관용구는 모든 것을 투명하게 만듭니다.

자동 인덱스 = `static_cast<int>(d * c.size())`;

기억할 사항 • "보이지 않

는" 프록시 유형은 `auto`가 초기에 대해 "잘못된" 유형을 추론하게 할 수 있습니다.

표현을 읽는다.

- 명시적으로 유형이 지정된 이나셜라이저 관용구는 `auto`가 원하는 유형을 추론하도록 합니다.
가질 것.

최신 C++로 이동

유명한 기능과 관련하여 C++11 및 C++14는 자랑할 것이 많습니다. 자동, 스마트 포인터, 의미 이동, 람다, 동시성 - 각각은 매우 중요하므로 한 장을 할애합니다. 이러한 기능을 마스터하는 것이 중요하지만 효과적인 현대 C++ 프로그래머가 되려면 일련의 작은 단계도 필요합니다. 각 단계는 C++98에서 최신 C++로의 여정에서 발생하는 특정 질문에 대한 답변입니다. 객체 생성에 괄호 대신 중괄호를 사용해야 하는 경우는 언제입니까? 별칭 선언이 `typedef`보다 나은 이유는 무엇입니까? `constexpr`은 `const`와 어떻게 다릅니까? `const` 멤버 함수와 스레드 안전성 사이의 관계는 무엇입니까? 목록은 계속됩니다. 그리고 이 장에서 하나씩 답을 제시합니다.

항목 7: 객체를 생성할 때 () 와 {} 를 구별 하세요.

관점에 따라 C++11에서 개체 초기화를 위한 구문 선택은 부끄러움이나 혼란스러운 혼란을 구현합니다. 일반적으로 초기화 값은 괄호, 등호 또는 중괄호로 지정할 수 있습니다.

정수 `x(0);` // 이나설라이저는 괄호 안에 있습니다.

정수 `y = 0;` // 이나설라이저는 "="를 따릅니다.

정수 `z[0];` // 이나설라이저는 중괄호 안에 있습니다.

많은 경우 등호와 중괄호를 함께 사용할 수도 있습니다.

정수 `z = { 0 };` // 이나설라이저는 "="와 중괄호를 사용합니다.

이 항목의 나머지 부분에서는 일반적으로 등호 더하기 중괄호 구문을 무시할 것입니다. C++에서는 일반적으로 중괄호만 있는 버전과 동일하게 취급하기 때문입니다.

"혼란스러운 혼란" 로비는 초기화에 등호를 사용한다는 점을 지적합니다.

종종 C++ 초보자가 할당이 진행되고 있다고 생각하도록 오도합니다.

그렇지는 않지만. int와 같은 내장 유형의 경우 차이점은 학문적이지만 사용자 정의 유형의 경우 초기화와 할당을 구별하는 것이 중요합니다.

다른 함수 호출이 포함됩니다.

```
위젯 w1;           // 기본 생성자 호출
```

```
위젯 w2 = w1;      // 할당이 아닙니다. 콜 컨структор
```

```
w1 = w2;           // 할당; 복사 연산자 호출 =
```

여러 초기화 구문을 사용하더라도 C++98이

원하는 초기화를 표현할 방법이 없었습니다. 예를 들어 불가능했습니다.

특정 집합을 보유하는 STL 컨테이너를 생성해야 함을 직접 나타냅니다.

값(예: 1, 3, 5).

여러 초기화 구문의 혼란을 해결하고

모든 초기화 시나리오를 다루지는 않지만 C++11에서는 균일한 초기화를 도입합니다.

최소한 개념적으로 어디서나 사용할 수 있는 단일 초기화 구문

모든 것을 표현합니다. 중괄호를 기반으로 하므로 중괄호라는 용어를 선호합니다.

초기화. "균일한 초기화"는 아이디어입니다. "중괄호 초기화"는 구문입니다.

건설하다.

중괄호 초기화를 사용하면 이전에는 표현할 수 없었던 것을 표현할 수 있습니다. 중괄호를 사용하여 컨테이너의 초기 내용을 확인하는 것은 쉽습니다.

```
표준::벡터<int> v{ 1, 3, 5 }; // v의 초기 내용은 1, 3, 5입니다.
```

중괄호를 사용하여 비정적 데이터에 대한 기본 초기화 값을 지정할 수도 있습니다.

회원. C++11의 새로운 기능인 이 기능은 "=" 초기화 구문과 공유됩니다.

세금(괄호 제외):

```
클래스 위젯 {
```

```
...
```

사적인:

```
정수 x[ 0 ]; 정수           // 좋습니다. x의 기본값은 0입니다.
```

```
y = 0; 정수               // 역시 좋다
```

```
z(0); };                  // 오류!
```

반면에 복사할 수 없는 객체(예: std::atomic - 항목 40 참조)는

중괄호 또는 괄호를 사용하여 초기화되지만 "="를 사용하지 않음:

```
표준::원자<int> ai1{ 0 };           // 좋아
```

표준::원자<int> ai2(0); // 좋아

표준::원자<int> ai3 = 0; // 오류!

따라서 중괄호 초기화를 "균일"이라고 부르는 이유를 쉽게 이해할 수 있습니다. C++의 초기화 표현식을 지정하는 세 가지 방법, 중괄호만 사용할 수 있습니다. 어디.

중괄호 초기화의 새로운 기능은 암시적 축소 변환을 금지한다는 것입니다. 빌트인 타입 중에서 중괄호 이나셜라이저의 표현식 값이 다음이 아닌 경우 초기화되는 객체의 유형으로 표현 가능하도록 보장되지만 코드는 엮다:

이중 x, y, z;

...

정수 합계1{ x + y + z }; // 오류! 두 배의 합
// int로 표현할 수 없음

괄호와 "="를 사용한 초기화는 축소 변환을 확인하지 않습니다. 너무 많은 레거시 코드를 깨뜨릴 수 있기 때문입니다.

정수 합계2(x + y + z); // OK(표현식 값
// int로 잘림)

정수 합계3 = x + y + z; // 마찬가지

중괄호 초기화의 또 다른 주목할만한 특징은 C++에 대한 내성입니다. 가장 성가신 구문 분석. C++ 규칙의 부작용은 다음과 같이 구문 분석될 수 있습니다. 선언은 하나로 해석되어야 하며, 가장 성가신 구문이 가장 자주 영향을 받습니다. 개발자가 객체를 기본적으로 구성하고 싶지만 실수로 대신 함수를 선언합니다. 문제의 근본은 인수가 있는 생성자, 다음과 같이 할 수 있습니다.

위젯 w1(10); // 인수 10으로 위젯 ctor 호출

그러나 유사한 방법을 사용하여 인수가 0인 위젯 생성자를 호출하려고 하면 구문에서 객체 대신 함수를 선언합니다.

위젯 w2(); // 가장 성가신 구문 분석! 함수를 선언
// 위젯을 반환하는 w2라는 이름!

매개변수 목록에 중괄호를 사용하여 함수를 선언할 수 없으므로 기본적으로 중괄호를 사용하여 객체를 구성하면 이 문제가 발생하지 않습니다.

위젯 w3{}; // 인수 없이 위젯 ctor를 호출합니다.

따라서 중괄호 초기화에 대해 할 말이 많습니다. 가장 다양한 컨텍스트에서 사용할 수 있는 구문이며 암시적 축소 변환을 방지하며 C++의 가장 성가신 구문 분석에 면역입니다. 선함의 삼위일체! 그렇다면 왜 이 항목에 "중괄호 초기화 구문 선호"와 같은 제목이 없습니까?

중괄호 초기화의 단점은 이에 수반되는 때때로 놀라운 동작입니다. 이러한 동작은 중괄호 이니셜라이저, std::initializer_lists 및 생성자 오버로드 해결 간의 비정상적으로 얹힌 관계에서 비롯됩니다.

그들의 상호 작용은 한 가지 일을 해야 하는 것처럼 보이지만 실제로는 다른 일을 하는 코드로 이어질 수 있습니다. 예를 들어, **항목 2** 는 자동 선언된 변수에 중괄호 이니셜라이저가 있을 때 추론된 유형이 std::initializer_list라고 설명합니다. 동일한 이니셜라이저를 사용하여 변수를 선언하는 다른 방법이 더 직관적인 유형을 생성하더라도 결과적으로 auto를 더 좋아할수록 중괄호 초기화에 대한 열정이 줄어들 것입니다.

생성자 호출에서 std::initializer_list 매개변수가 포함되지 않는 한 괄호와 중괄호는 동일한 의미를 갖습니다.

클래스 위젯 { 공개:

```
위젯(int i, bool b);           // ctor가 선언하지 않음 //
위젯(int i, double d);         std::initializer_list 매개변수
...
};

위젯 w1(10, true);            // 첫 번째 ctor를 호출합니다.

위젯 w2{10, true};            // 첫 번째 ctor도 호출

위젯 w3(10, 5.0);             // 두 번째 ctor 호출

위젯 w4{10, 5.0};             // 두 번째 ctor도 호출
```

그러나 하나 이상의 생성자가 std::initializer_list 유형의 매개 변수를 선언하는 경우 중괄호 초기화 구문을 사용하는 호출은 std::initializer_lists를 사용하는 오버로드를 강력하게 선호합니다. 강하게. 컴파일러가 중괄호 초기화를 사용하여 std::initializer_list를 사용하는 생성자에 대한 호출을 구성할 수 있는 방법이 있으면 컴파일러는 해당 해석을 사용합니다. 위의 Widget 클래스가 std::initializer_list<long double>을 취하는 생성자로 보강된 경우, 예를 들어,

클래스 위젯 { 공개:

```
위젯(int i, bool b);           // 이전과 같이 //
위젯(int i, double d);         이전과 같이
```

```
위젯(std::initializer_list<long double> il); // 추가됨
```

```
...  
};
```

위젯 w2 및 w4는 새 생성자를 사용하여 구성됩니다.
 std::initializer_list 요소의 유형(long double)은
 non-std::initializer_list 생성자, 두 인수 모두에 대해 더 나쁜 일치!
 바라보다:

위젯 w1(10, true);	// 괄호를 사용하고 이전과 같이 // 첫 번째 ctor를 호출합니다.
위젯 w2{10, true};	// 중괄호를 사용하지만 지금은 호출 // std::initializer_list ctor // (10 및 true는 long double로 변환)
위젯 w3(10, 5.0);	// 괄호를 사용하고 이전과 같이 // 두 번째 ctor 호출
위젯 w4{10, 5.0};	// 중괄호를 사용하지만 지금은 호출 // std::initializer_list ctor // (10과 5.0은 long double로 변환)

일반적으로 복사 및 이동 구성도 다음으로 하이제킹될 수 있습니다.

std::initializer_list 생성자:

클래스 위젯 {	
공공의:	
위젯(int i, bool b); // 이전과	
위젯(int i, double d); // 이전과	
위젯(std::initializer_list<long double> il); // 이전과	
연산자 float() const;	// 전환하다
...	// 부동
};	
위젯 w5(w4);	// 괄호를 사용하고 복사 ctor를 호출합니다.
위젯 w6{w4};	// 중괄호 사용, 호출 // std::initializer_list ctor // (w4는 float로 변환하고 float // long double로 변환)

```
위젯 w7(std::move(w4));           // 팔호를 사용하고 move ctor를 호출합니다.
```

```
위젯 w8{std::move(w4)};          // 중괄호 사용, 호출
                                  // std::initializer_list ctor
                                  // (w6과 같은 이유로)
```

중괄호 이니셜라이저와 생성자를 일치시키려는 컴파일러의 결정
`std::initializer_lists`는 너무 강력하여 `std::ini`가 가장 잘 일치하더라도 우선합니다.
`tializer_list` 생성자를 호출할 수 없습니다. 예를 들어:

```
클래스 위젯 {
    공공의:
        위젯(int i, bool b);           // 이전과
        위젯(int i, double d);         // 이전과

    위젯(std::initializer_list<bool> il); // 요소 유형은
                                              // 이제 부울
    ...
};                                         // 암시적 없음
                                            // 변환 함수

위젯 w{10, 5.0};                         // 오류! 축소 전환이 필요합니다.
```

여기서 컴파일러는 처음 두 생성자를 무시합니다(두 번째 생성자는
 두 인수 유형 모두에서 정확히 일치)를 사용하여 생성자를 호출하려고 시도합니다.
`std::initializer_list<bool>`. 해당 생성자를 호출하면 변환이 필요합니다.
 bool에 대한 `int(10)` 및 `double(5.0)`. 두 전환 모두 좁혀질 것입니다.
 (`bool`은 두 값 모두를 정확하게 나타낼 수 없음) 축소 변환이 금지됩니다.
 중괄호 이니셜라이저 내부에 있으므로 호출이 유효하지 않고 코드가 거부됩니다.

중괄호 이니셜라이저의 인수 유형을 다음으로 변환할 방법이 없는 경우에만
`std::initializer_list`의 유형은 컴파일러가 일반 오버로드에 대해 풀백합니까?
 해결. 예를 들어 `std::initializer_list<bool>` 구성을 대체하면
`std::initializer_list<std::string>`을 취하는 토록해서 비 `std::initializer_list` 생성자는 다시 후
 보가 됩니다.
 int와 bool을 `std::strings`로 변환하는 방법:

```
클래스 위젯 {
    공공의:
        위젯(int i, bool b);           // 이전과
        위젯(int i, double d);         // 이전과

    // std::initializer_list 요소 유형은 이제 std::string입니다.
    위젯(std::initializer_list<std::string> il);
    ...
                                              // 암시적 없음
```

```

};

// 변화 함수

위젯 w1(10, true); // 팔호를 사용하지만 여전히 첫 번째 ctor를 호출합니다.

위젯 w2{10, true}; // 중괄호를 사용하여 이제 첫 번째 ctor를 호출합니다.

위젯 w3(10, 5.0); // 팔호를 사용하고 여전히 두 번째 ctor를 호출합니다.

위젯 w4{10, 5.0}; // 중괄호를 사용하여 이제 두 번째 ctor를 호출합니다.

```

이로써 중괄호 이니셜라이저와 생성자에 대한 조사가 거의 끝나갈 무렵
 오버로딩, 하지만 해결해야 하는 흥미로운 엣지 케이스가 있습니다. 가정하다
 기본 구성을 지원하는 객체를 구성하기 위해 빈 중괄호 세트를 사용합니다.
 또한 std::initializer_list 구성을 지원합니다. 당신의 빈 무엇
 중괄호 의미? "인수 없음"을 의미하는 경우 기본 구성을 얻습니다.
 "빈 std::initializer_list"를 의미합니다. std::ini에서 구성을 가져옵니다.
 요소가 없는 tializer_list.

규칙은 기본 구성을 얻는 것입니다. 빈 중괄호는 인수가 없음을 의미합니다.
 빈 std::initializer_list:

```

클래스 위젯 {
  공공의:
    위젯(); // 기본 ctor

    위젯(std::initializer_list<int> il); // std::이니셜라이저
                                                // _리스트 ctor

    ...
};

// 암시적 없음
// 변화 함수

위젯 w1; // 기본 ctor 호출

위젯 w2{}; // 기본 ctor도 호출

위젯 w3(); // 가장 성가신 구문 분석! 함수를 선언합니다!

```

빈 std::ini로 std::initializer_list 생성자를 호출하려는 경우
 tializer_list, 빈 중괄호를 생성자 인수로 만들어 수행합니다.
 팔호 안에 빈 중괄호를 넣거나 자신이 무엇인지 구분하는 중괄호
 통과:

```

위젯 w4({}); // std::initializer_list ctor를 호출합니다.
                // 빈 목록으로

위젯 w5{}{}; // 마찬가지

```

이 시점에서 중괄호 초기화자, `std::initializer_lists` 및 생성자 오버로딩에 대한 모호한 규칙이 머릿속에 떠돌아다니는 상황에서 이 정보가 일상적인 프로그래밍에서 얼마나 중요한지 의아해할 수 있습니다. 직접적으로 영향을 받는 클래스 중 하나가 `std::vector`이기 때문에 생각보다 많습니다. `std::vector`에는 컨테이너의 초기 크기와 각 초기 요소가 가져야 하는 값을 지정할 수 있는 비 `std::initializer_list` 생성자가 있습니다. 컨테이너의 초기 값을 지정합니다. 숫자 유형의 `std::vector`(예: `std::vector<int>`)를 만들고 생성자에 두 개의 인수를 전달하면 해당 인수를 괄호로 묶는지 중괄호로 묶는지에 따라 엄청난 차이가 납니다. :

```
표준::벡터<int> v1(10, 20); // non-std::initializer_list 사용 // ctor: 10개 요소 생성 //
std::vector, // 모든 요소의 값은 20
```

```
표준::벡터<int> v2{10, 20}; // std::initializer_list ctor 사용:
// 2-요소 std::vector 생성, // 요소 값은 10 및
20
```

그러나 `std::vector` 및 괄호, 중괄호 및 생성자 오버로딩 해결 규칙의 세부 사항에서 뒤로 물려나자. 이 논의에서 두 가지 주요 시사점을 얻을 수 있습니다. 먼저 클래스 작성자는 오버로드된 생성자 집합에 `std::initializer_list`를 사용하는 하나 이상의 함수가 포함되어 있는 경우 중괄호 초기화를 사용하는 클라이언트 코드에서 `std::initializer_list` 오버로드만 볼 수 있음을 알아야 합니다. 결과적으로 호출된 오버로드가 클라이언트가 괄호 또는 중괄호를 사용하는지 여부에 영향을 받지 않도록 생성자를 설계하는 것이 가장 좋습니다. 다시 말해서, `std::vector` 인터페이스 디자인에서 현재 오류로 간주되는 것에서 배우고 이를 방지하도록 클래스를 디자인하십시오.

`std::initializer_list` 생성자가 없는 클래스가 있고 하나를 추가하면 중괄호 초기화를 사용하는 클라이언트 코드가 비 `std::initializer_list` 생성자로 해석되는 호출이 이제 새로운 생성자로 해석된다는 것을 의미합니다. 가능. 물론 이러한 종류의 일은 오버로드 집합에 새 함수를 추가할 때마다 발생할 수 있습니다. 이전 오버로드 중 하나로 해결하는 데 사용된 호출이 새 오버로드를 호출하기 시작할 수 있습니다. `std::initializer_list` 생성자 오버로드와의 차이점은 `std::initializer_list` 오버로드가 다른 오버로드와 경쟁할 뿐만 아니라 다른 오버로드가 거의 고려되지 않는 지점까지 덮어쓴다는 것입니다. 따라서 이러한 오버로드는 신중하게 추가해야 합니다.

두 번째 교훈은 클래스 클라이언트로서 객체를 생성할 때 괄호와 중괄호 중에서 신중하게 선택해야 한다는 것입니다. 대부분의 개발자는 결국 한 가지 유형을 선택합니다.

필요한 경우에만 다른 구분 기호를 사용합니다. 기본적으로 중괄호를 사용하는 사람들은 타의 추종을
불허하는 적용 범위, 축소 변환 금지, C++의 가장 성가신 구문 분석에 대한 내성에 매료됩니다. 그러한
사람들은 어떤 경우(예: 주어진 크기와 초기 요소 값으로 std::vector 생성)에 괄호가 필요하다는 것
을 이해합니다. 반면에 go parentheses-go 군중은 괄호를 기본 인수 구분 기호로 받아들입니다.

그들은 C++98 구문 전통과의 일관성, auto-deduced-a-std::initializer_list 문제의 회피, 그리고
그들의 객체 생성 호출이 std:: initializer_list 생성자. 그들은 때때로 중괄호만이 할 수 있음을 인정합
니다(예: 특정 값으로 컨테이너를 생성할 때). 어느 쪽 접근 방식이 다른 접근 방식보다 낫다는 데에는
합의가 없으므로 하나를 선택하여 일관되게 적용하는 것이 좋습니다.

템플릿 작성자인 경우 개체 생성을 위한 괄호와 중괄호 사이의 긴장이 특히 답답할 수 있습니다. 일반
적으로 어느 것을 사용해야 하는지 알 수 없기 때문입니다. 예를 들어, 임의의 수의 인수에서 임의 유형
의 개체를 생성한다고 가정합니다. 가변 템플릿은 이것을 개념적으로 간단하게 만듭니다.

```
template<typename T,                                     // 생성할 객체 유형 // 사용할 인수 유형
         typename... Ts> 무
효 doSomeWork(Ts&&... params) {
    ...
    params에서 로컬 T 객체 생성...
    ...
}
```

의사 코드 줄을 실제 코드로 바꾸는 두 가지 방법이 있습니다(std::forward에 대한 정보는 [항목 25](#)
참조).

```
T localObject(std::forward<Ts>(params)...);           // 괄호 사용
```

```
T localObject{std::forward<Ts>(params)...};           // 중괄호 사용
```

따라서 다음 호출 코드를 고려하십시오.

```
표준::벡터<int> v;
...
doSomeWork<std::vector<int>>(10, 20);
```

`doSomeWork`가 `localObject`를 생성할 때 괄호를 사용하는 경우 결과는 10개 요소가 있는 `std::vector`입니다. `doSomeWork`가 중괄호를 사용하는 경우 결과는 2개의 요소가 있는 `std::vector`입니다. 어느 것이 맞습니까? `doSomeWork`의 저자는 알 수 없습니다. 발신자만 할 수 있습니다.

이것이 바로 표준 라이브러리 함수 `std::make_unique` 및 `std::make_shared`가 직면한 문제입니다([항목 21](#) 참조). 이러한 함수는 내부적으로 괄호를 사용하고 이 결정을 인터페이스의 일부로 문서화하여 문제를 해결합니다.¹

기억해야 할 사항 • 중

괄호 초기화는 가장 널리 사용되는 초기화 구문으로, 축소 변환을 방지하고 C++의 가장 성가신 구문 분석에 영향을 받지 않습니다. • 생성자 오버로드 해결 중에 다른 생성자가 더 나은 일치를 제공하더라도 가능한 경우 중괄호 이나설라이저가 `std::initializer_list` 매개변수와 일치합니다. • 괄호와 중괄호 중 선택이 중요한 차이를 만들 수 있는 예는 두 개의 인수로 `std::vector<numeric type>` 을 만드는 것입니다.

- 템플릿 내에서 개체 생성을 위해 괄호와 중괄호 중에서 선택하는 것은 어려울 수 있습니다.

항목 8: 0 과 NULL 보다 nullptr 을 선호하십시오 .

여기 거래가 있습니다. 리터럴 0은 포인터가 아니라 `int`입니다. C++가 포인터만 사용할 수 있는 컨텍스트에서 0을 보고 있는 자신을 발견하면 마지못해 0을 널 포인터로 해석하지만 이는 대체 위치입니다. C++의 기본 정책은 0이 포인터가 아니라 `int`라는 것입니다.

실제로는 NULL도 마찬가지입니다. 구현이 `int`(예: `long`) 이외의 정수 유형을 NULL에 제공할 수 있기 때문에 NULL의 경우 세부 정보에 약간의 불확실성이 있습니다. 일반적이지는 않지만 여기서 문제는 NULL의 정확한 유형이 아니기 때문에 0도 NULL도 포인터 유형을 갖지 않기 때문에 실제로 중요하지 않습니다.

¹ 템플릿에서 생성된 함수에 괄호 또는 중괄호를 사용해야 하는지 여부를 호출자가 결정할 수 있도록 하는 보다 유연한 디자인이 가능합니다. 자세한 내용은 [Andrzej의 C++ 블로그](#) 2013년 6월 5일 항목을 참조하십시오. "[직관적인 인터페이스 - 1부](#)."

C++98에서 이것의 주된 의미는 포인터와 통합 유형에 대한 오버로딩이 놀라움으로 이어질 수 있다는 것이었습니다. 이러한 오버로드에 0 또는 NULL을 전달하면 포인터 오버로드가 호출되지 않습니다.

무효 f(int); 무효 f(bool); 무효 f(무 효*);	<i>// f의 세 가지 오버로드</i>
f(0);	<i>// f(void*)가 아닌 f(int)를 호출합니다.</i>
f(NULL);	<i>// 컴파일되지 않을 수 있지만 일반적으로 // f(int)를 호출합니다. f(void*)를 호출하지 않음</i>

f(NULL)의 동작에 대한 불확실성은 NULL 유형과 관련하여 구현에 부여된 여유를 반영합니다. NULL이 예를 들어 0L(즉, long으로 0)로 정의되면 long에서 int로, long에서 bool로, 0L에서 void*로의 변환이 동등하게 양호한 것으로 간주되기 때문에 호출이 모호합니다. 그 호출에 대한 흥미로운 점은 소스 코드의 명백한 의미("나는 f를 널 포인터로 호출합니다.")와 실제 의미("나는 f를 어떤 종류의 정수로 호출합니다. 널 포인터"). 이 반직관적인 동작은 C++98 프로그래머가 포인터 및 정수 형식에 대한 오버로드를 방지하기 위한 지침으로 이어졌습니다. 이 지침은 이 항목의 조언에도 불구하고 nullptr이 더 나은 선택임에도 불구하고 일부 개발자는 계속 0과 NULL을 사용할 가능성이 높기 때문에 C++11에서 여전히 유효합니다.

nullptr의 장점은 정수 형식이 없다는 것입니다. 솔직히 포인터 타입도 없지만 모든 타입의 포인터라고 생각하시면 됩니다. nullptr의 실제 유형은 std::nullptr_t이고, 훌륭하게 순환하는 정의에서 std::nullptr_t는 nullptr 유형으로 정의됩니다. std::nullptr_t 유형은 암시적으로 모든 원시 포인터 유형으로 변환되며, 이것이 nullptr이 모든 유형의 포인터인 것처럼 작동하게 만드는 이유입니다.

nullptr을 사용하여 오버로드된 함수 f를 호출하면 void* 오버로드(즉, 포인터 오버로드)가 호출됩니다. nullptr은 정수로 볼 수 없기 때문입니다.

`f(nullptr);` *// f(void*) 오버로드 호출*

따라서 0 또는 NULL 대신 nullptr을 사용하면 과부하 해결 놀라움을 피할 수 있지만 이것이 유일한 아점은 아닙니다. 또한 특히 자동 변수가 관련된 경우 코드 명확성을 향상시킬 수 있습니다. 예를 들어 코드 베이스에서 다음과 같은 문제가 발생했다고 가정해 보겠습니다.

`자동 결과 = findRecord(/* 인수 */);`

`if (결과 == 0) {`

```
...
}
```

findRecord가 반환하는 내용을 알지 못하는 경우(또는 쉽게 찾을 수 없는 경우) 결과가 포인터 유형인지 정수 유형인지 명확하지 않을 수 있습니다. 결국, 0 (어떤 결과가 테스트 대상인지) 어느 쪽이든 갈 수 있습니다. 다음이 표시되면 반면,

```
자동 결과 = findRecord( /* 인수 */ );
```

```
if (결과 == nullptr) {
    ...
}
```

모호함이 없습니다. 결과는 포인터 유형이어야 합니다.

nullptr은 템플릿이 그림에 들어갈 때 특히 밝게 빛납니다. 당신이 적절한 뮤텍스가 있을 때만 호출되어야 하는 일부 함수가 있습니다. 잠금. 각 함수는 다른 종류의 포인터를 사용합니다.

```
int      f1(std::shared_ptr<위젯> spw); // 다음 경우에만 호출
double f2(std::unique_ptr<위젯> upw); // 적절한
bool f3(위젯* pw); // 뮤텍스가 잠겨 있습니다.
```

nullptr 포인터를 전달하려는 호출 코드는 다음과 같습니다.

```
std::mutex f1m, f2m, f3m;                                // f1, f2 및 f3에 대한 뮤텍스

MutexGuard 사용 =                                         // C++11 형식 정의; 항목 9 참조
    std::lock_guard<std::mutex>;
...

{
    멱스가드 g(f1m); 자동                                // f1에 대한 뮤텍스 잠금
    결과 = f1(0);                                         // 0을 nullptr로 f1에 전달
    // 뮤텍스 잠금 해제

    ...
}

{
    멱스가드 g(f2m); 자동                                // f2에 대한 뮤텍스 잠금
    결과 = f2(NULL);                                     // NULL을 nullptr로 f2에 전달
    // 뮤텍스 잠금 해제

    ...
}

{
```

```
    멀스가드 g(f3m); 자동 결           // f3에 대한 뮤텍스 잠금
    과 = f3(nullptr); // nullptr을 null ptr로 f3에 전달 } // 뮤텍스 잠금 해제
```

이 코드의 처음 두 호출에서 nullptr을 사용하지 못한 것은 슬픈 일이지만 코드가 작동하고 그 점이 중요합니다. 그러나 호출 코드에서 반복되는 패턴(잠금 뮤텍스, 함수 호출, 뮤텍스 잠금 해제)은 슬프기만 합니다. 짜증나네요.

이러한 종류의 소스 코드 중복은 템플릿이 피하도록 설계된 것 중 하나이므로 패턴을 템플릿화해 보겠습니다.

```
템플릿<유형명 FuncType, 유형명 MuxType,
        유형명 PtrType>

auto lockAndCall(FuncType func,
                 MuxType 및 뮤텍스,
                 PtrType ptr) -> decltype(func(ptr))
{
    MuxGuard g(뮤텍스); 반환
    함수(ptr);
}
```

이 함수의 반환 유형(auto ... -> decltype(func(ptr)))이 머리를 긁적거리게 하는 경우에는 고개를 끄덕이고 무슨 일이 일어나고 있는지 설명하는 항목 3으로 이동하십시오. C++14에서 반환 유형은 간단한 decltype(auto)으로 축소될 수 있습니다.

```
template<유형명 FuncType, 유형명
         MuxType, 유형명 PtrType>
decltype(auto)
lockAndCall(FuncType func,                                // C++14
           MuxType 및 뮤텍스,
           포인트 유형 포인트)
{
    MuxGuard g(뮤텍스); 반환
    함수(ptr);
}
```

lockAndCall 템플릿(어느 버전이든)이 주어지면 호출자는 다음과 같은 코드를 작성할 수 있습니다.

```
자동 결과1 = 잠금 및 호출(f1, f1m, 0);           // 오류!
```

```
...
```

```
자동 결과2 = 잠금 및 호출(f2, f2m, NULL);          // 오류!
```

```
...
```

자동 결과3 = 잠금 및 호출(f3, f3m, nullptr); // 좋아

글쎄, 그들은 그것을 작성할 수 있지만 주석에서 알 수 있듯이 세 가지 경우 중 두 가지 경우 코드가 컴파일되지 않습니다. 첫 번째 호출의 문제는 lockAnd Call에 0이 전달되면 템플릿 유형 추론이 시작되어 유형을 파악한다는 것입니다. 0의 유형은 이전이었고 항상 int일 것이므로 lockAndCall에 대한 이 호출의 인스턴스화 내부에 있는 매개변수 ptr의 유형입니다. 불행히도 이것은 lockAndCall 내부의 func 호출에서 int가 전달되고 f1이 기대하는 std::shared_ptr<Widget> 매개변수와 호환되지 않는다는 것을 의미합니다. lockAndCall에 대한 호출에서 전달된 0은 널 포인터를 나타내기 위한 것 이지만 실제로 전달된 것은 평범한 int였습니다. 이 int를 std::shared_ptr <Widget>으로 f1에 전달하는 것은 유형 오류입니다. 템플릿 내부에서 std:: shared_ptr<Widget>이 필요한 함수에 int 가 전달되고 있기 때문에 0으로 lockAndCall에 대한 호출이 실패합니다.

NULL과 관련된 호출에 대한 분석은 본질적으로 동일합니다. NULL이 lockAndCall에 전달되면 매개 변수 ptr에 대해 정수 유형이 추론되고 int 또는 int와 유사한 유형인 ptr이 f2에 전달되면 유형 오류가 발생합니다. 이는 std::unique_ptr<Widget> .

대조적으로, nullptr을 포함하는 호출은 문제가 없습니다. nullptr이 lockAndCall에 전달되면 ptr 의 유형은 std::nullptr_t로 추론됩니다. ptr이 f3에 전달되면 std::nullptr_t가 모든 포인터 유형으로 암시적으로 변환되기 때문에 std::nullptr_t에서 Widget*으로 암시적으로 변환됩니다.

템플릿 유형 추론이 0과 NULL에 대해 "잘못된" 유형을 추론한다는 사실(즉, 널 포인터에 대한 표현으로서의 풀백 의미가 아니라 실제 유형)은 널 포인터를 참조하려고 합니다. nullptr을 사용하면 템플릿에 특별한 문제가 없습니다.

nullptr이 0과 NULL에 취약한 과부하 해결 놀라움을 겪지 않는다는 사실과 함께 이 경우는 철통처럼 보입니다. 널 포인터를 참조하려면 0이나 NULL이 아닌 nullptr을 사용하십시오.

기억해야 할 사항

- 0 및 NULL보다 nullptr을 선호합니다.
- 정수 및 포인터 유형에 대한 오버로드를 방지합니다.

항목 9: `typedef`보다 별칭 선언을 선호 합니다.

나는 우리가 STL 컨테이너를 사용하는 것이 좋은 생각이라는 데 동의할 수 있다고 확신하며, [Item 18](#) 이 `std::unique_ptr`을 사용하는 것이 좋은 생각이라는 것을 당신에게 확신시키기를 바랍니다. `std::unique_ptr<std::unordered_map<std::string, std::string>>`"을 두 번 이상 사용합니다. 생각만 해도 수근관 증후군의 위험이 높아집니다.

그러한 의학적 비극을 피하는 것은 쉽습니다. `typedef` 도입:

```
typedef
    std::unique_ptr<std::unordered_map<std::string, std::string>>
UPtrMapSS;
```

그러나 `typedef`은 C++98입니다. 물론 C++11에서 작동하지만 C++11은 별칭 선언도 제공합니다.

`UPtrMapSS` 사용 =
`std::unique_ptr<std::unordered_map<std::string, std::string>>;`

`typedef`과 alias 선언이 정확히 같은 일을 한다는 점을 감안할 때, 둘 중 하나를 선호하는 확실한 기술적 이유가 있는지 궁금해하는 것이 합리적입니다.

있습니다. 그러나 그것에 도달하기 전에 많은 사람들이 함수 포인터와 관련된 유형을 다룰 때 별칭 선언을 삼키기가 더 쉽다고 생각한다는 점을 언급하고 싶습니다.

```
// FP는 int와 // const std::string&을 취하고 아무 것도 반환하지 않는 함수에 대한 포인터와 동의어입니다.
typedef void (*FP)(int, const std::string&); // 형식 정의
```

```
// 위와 같은 의미로 FP = void (*)(int,
const std::string&); // 별칭 // 선언
```

물론 두 형식 모두 질식하기 쉽지 않으며 어쨌든 함수 포인터 유형에 대한 동의어를 처리하는 데 많은 시간을 할애하는 사람은 거의 없습니다. 따라서 이것이 `typedef`보다 별칭 선언을 선택해야 하는 설득력 있는 이유는 아닙니다.

그러나 강력한 이유가 존재합니다. 바로 템플릿입니다. 특히 별칭 선언은 템플릿화될 수 있지만(이 경우 별칭 템플릿이라고 함) `typedef`은 템플릿화할 수 없습니다.

이것은 C++11 프로그래머에게 C++98에서 템플릿화된 구조체 내부에 중첩된 `typedef`과 함께 해킹되어야 했던 것을 표현하기 위한 간단한 메커니즘을 제공합니다. 예를 들어 사용자 지정 할당자 `MyAlloc`을 사용하는 연결 목록에 대한 동의어를 정의하는 것을 고려하십시오. 별칭 템플릿을 사용하면 케이크 조각이 됩니다.

```
템플릿<유형이름 T> // MyAllocList<T>
MyAllocList 사용 = std::list<T, MyAlloc<T>>; // 동의어입니다 // 표준::목록<T,
// MyAlloc<T>>

MyAllocList<위젯> lw; // 클라이언트 코드
```

`typedef`를 사용하면 케이크를 처음부터 만들어야 합니다.

```
템플릿<유형이름 T> // MyAllocList<T>::유형
struct MyAllocList { // 동의어입니다.
    typedef std::list<T, MyAlloc<T>> 유형; // 표준::목록<T,
}; // MyAlloc<T>>
MyAllocList<위젯>::유형 lw; // 클라이언트 코드
```

더 나빠진다. 생성 목적으로 템플릿 내부에서 `typedef`를 사용하려면 템플릿 매개변수에 의해 지정된 유형의 객체를 보유하는 연결 목록을 사용하면 `typedef` 이름 앞에 `typename`이 있어야 합니다.

```
템플릿<유형이름 T>
클래스 위젯 {
    private: // MyAllocList<T>
        유형 이름 MyAllocList <T>::유형 목록; // 데이터 멤버로
        ...
};

// 위젯<T> 포함
```

여기서 `MyAllocList<T>::type`은 템플릿 유형에 의존하는 유형을 나타냅니다. 매개변수(`T`). 따라서 `MyAllocList<T>::type`은 종속 유형이며 C++의 많은 사랑스러운 규칙은 종속 유형의 이름 앞에 유형이 와야 한다는 것입니다. 이를.

MyAllocList가 별칭 템플릿으로 정의되면 typename에 대한 이러한 필요성이 사라집니다(번거로운 "::type" 접미사):

템플릿<유형이름 T>
MyAllocList 사용 = std::list<T, MyAlloc<T>>; // 이전과

```
템플릿<유형이름 T>
클래스 위젯 {
    사적인:
        MyAllocList<T> 목록;
        ...
    };
    // "유형 이름" 없음,
    // "::유형" 없음
```

`MyAllocList<T>`(즉, 별칭 템플릿 사용)가 종속적으로 보일 수 있습니다.
템플릿 매개변수 `T`에서 `MyAllocList<T>::type`으로 (즉, 중첩 유형 사용)

def), 하지만 당신은 컴파일러가 아닙니다. 컴파일러가 위젯 템플릿을 처리하고 MyAllocList<T>를 사용하는 경우(즉, 별칭 템플릿 사용), MyAllocList가 별칭 템플릿이기 때문에 MyAllocList<T>가 유형의 이름임을 알고 있습니다. 유형. 따라서 MyAllocList<T>는 비종속 유형이며 유형 이름 지정자는 필수도 허용도 아닙니다.

컴파일러는 위젯 템플릿에서 MyAllocList<T>::type(즉, 중첩된 typedef 사용)을 볼 때 반면에 MyAllocList의 특수화가 있을 수 있기 때문에 그것이 유형의 이름을 지정하는지 확신할 수 없습니다. MyAllocList<T>::type이 유형이 아닌 다른 것을 참조하는 위치를 아직 보지 못했습니다. 미친 소리처럼 들리지만 이러한 가능성에 대해 컴파일러를 비난하지 마십시오. 그러한 코드를 생성하는 것으로 알려진 것은 인간입니다.

예를 들어, 어떤 잘못 인도된 영혼은 다음과 같은 것을 만들어 냈을 수 있습니다.

```
클래스 와인 { ... };

template<typename T> // MyAllocList 특수화 // T가 Wine일 때
class MyAllocList<Wine> { private:
    열거형 클래스 WineType // "enum class"에 대한 정보는 // 항목
    { 화이트, 레드, 로즈 }; 10 을 참조하십시오 .

    와인 종류 유형; // 이 클래스에서 type은 // 데이터 멤버
    ...           입니다!
};
```

보시다시피 MyAllocList<Wine>::type은 유형을 참조하지 않습니다. 위젯이 Wine으로 인스턴스화되는 경우 위젯 템플릿 내부의 MyAllocList<T>::type은 유형이 아니라 데이터 멤버를 참조합니다. 그런 다음 위젯 템플릿 내에서 MyAllocList<T>::type이 유형을 참조하는지 여부는 솔직히 T가 무엇인지에 따라 달라지며, 이것이 컴파일러가 typename 앞에 선행하여 그것이 유형임을 주장하는 이유입니다.

템플릿 메타프로그래밍(TMP)을 해본 적이 있다면 템플릿 유형 매개변수를 가져와서 수정된 유형을 생성해야 하는 필요성에 거의 확실하게 부딪쳤을 것입니다. 예를 들어, 어떤 유형의 T가 주어지면 T에 포함된 모든 const 또는 참조 한정자를 제거할 수 있습니다. 예를 들어 const std::string&을 std::string으로 바꿀 수 있습니다. 또는 유형에 const를 추가하거나 lvalue 참조로 바꾸고 싶을 수도 있습니다. 예를 들어 Widget을 const Widget으로 또는 Widget&으로 바꾸십시오. (TMP를 해본 적이 없다면 그것은 너무 나쁜 것입니다. 왜냐하면 진정으로 효과적인 C++ 프로그래머가 되고 싶다면 최소한 C++의 이 측면에 대한 기본 사항에 익숙해야 하기 때문입니다.)

항목 23 및 27에서 방금 언급한 유형 변환의 종류를 포함하여 TMP가 작동하는 예를 볼 수 있습니다.)

C++11은 `<type_traits>` 헤더 내부의 템플릿 모음인 유형 특성의 형태로 이러한 종류의 변환을 수행 할 수 있는 도구를 제공합니다. 해당 헤더에는 수십 개의 유형 특성이 있으며 모두 유형 변환을 수행 하는 것은 아니지만 예측 가능한 인터페이스를 제공하는 특성이 있습니다. 변환을 적용하려는 유형 `T` 가 주어지면 결과 유형은 `std::transformer <T>::type`입니다. 예를 들어:

<code>std::remove_const<T>::type</code> <code>std::remove_reference<T>::type</code> <code>std::add_lvalue_reference<T>::type</code>	<code>// const T에서 T를 산출합니다.</code> <code>// T& 및 T&&에서 T를 산출합니다.</code> <code>// T에서 T& 산출</code>
---	--

주석은 이러한 변환이 수행하는 작업을 요약한 것일 뿐이므로 너무 문자 그대로 받아들이지 마십시오. 프로젝트에 사용하기 전에 정확한 사양을 찾아봐야 합니다.

여기서 내 동기는 어쨌든 유형 특성에 대한 자습서를 제공하는 것이 아닙니다. 오히려 이러한 변환을 적용하려면 각 사용 끝에 `::type`을 작성해야 합니다. 템플릿 내부의 유형 매개변수에 적용하는 경우(실제 코드에서 거의 항상 사용하는 방식임) 유형 이름을 사용할 때마다 먼저 사용해야 합니다. 이러한 두 가지 구문 속도 범프의 이유는 C++11 유형 특성이 템플릿화된 구조체 내에서 중첩된 `typedef`로 구현되기 때문입니다. 맞아요, 그것들은 제가 여러분에게 앤리어스 템플릿보다 열등하다고 확신시키려 했던 유형 동의어 기술을 사용하여 구현되었습니다!

그것에 대한 역사적 이유가 있지만 표준화 위원회가 별칭 템플릿이 더 나은 방법이라는 것을 뒤늦게 인정하고 C++14에 그러한 템플릿을 포함했기 때문에 견너될 것입니다. 모든 C++11 유형 변환에 대해. 별칭에는 다음과 같은 공통 형식이 있습니다. 각 C++11 변환에 대해

`std::transformation<T>::type` 에는 `std::transformation_t`라는 해당 C++14 별칭 템플릿이 있습니다. 예를 통해 내가 의미하는 바를 명확히 알 수 있습니다.

<code>std::remove_const<T>::type</code> <code>std::remove_const_t<T></code> <code>std::remove_reference<T>::type</code> <code>std::remove_reference_t<T></code> <code>std::add_lvalue_reference<T>::type</code> <code>std::add_lvalue_reference_t<T></code>	<code>// C++11: const T → T</code> <code>// C++14에 해당</code> <code>// C++11: T&/T&& → T</code> <code>// C++14에 해당</code> <code>C++11: T → T&</code> <code>// C++14에 해당</code>
--	--

C++11 구문은 C++14에서 유효하지만 왜 사용하려는지 모르겠습니다. C++14에 액세스할 수 없는 경우에도 별칭 템플릿을 직접 작성하는 것은 어린아이의 장난입니다. C++11 언어 기능만 필요하며 어린이도 할 수 있습니다.

패턴을 흉내내죠? C++14 표준의 전자 사본에 액세스할 수 있는 경우 복사 및 붙여넣기만 하면 되므로 훨씬 더 쉽습니다. 여기에서 시작하겠습니다.

`remove_const_t`를 사용

하는 템플릿 `<class T> = typename remove_const<T>::type;`

`remove_reference_t`를

사용하는 템플릿 `<class T> = typename remove_reference<T>::type;`

`add_lvalue_reference_t`

를 사용하는 템플릿 `<class T> = typename`

`add_lvalue_reference<T>::type;`

보다? 이보다 더 쉬울 수는 없습니다.

기억해야 할 사항

- `typedef`는 템플릿화를 지원하지 않지만 별칭 선언은 지원합니다. • 별칭 템플릿은 "`::type`" 접미사를 피하고 템플릿에서 "`typename`" `typedef`를 참조하는 데 종종 접두사가 필요합니다.
- C++14는 모든 C++11 유형 특성 변환에 대한 별칭 템플릿을 제공합니다.

항목 10: 범위 가 지정되지 않은 열거형보다 범위가 지정된 열거형 을 선호 합니다.

일반적으로 중괄호 안에 이름을 선언하면 해당 이름의 가시성이 중괄호로 정의된 범위로 제한됩니다. C++98 스타일 열거형에서 선언된 열거자는 그렇지 않습니다. 이러한 열거자의 이름은 열거형을 포함하는 범위에 속하며, 이는 해당 범위의 다른 어떤 것도 동일한 이름을 가질 수 없음을 의미합니다.

열거형 색상 { 검정, 흰색, 빨강 }; // 검정, 흰색, 빨강은 // Color와 같은 범위에 있습니다.

자동 흰색 = 거짓;

// 오류! 흰색 이미 // 이 범위에서 선언됨

이러한 열거자 이름이 열거형 정의를 포함하는 범위로 누출된다는 사실은 이러한 종류의 열거형에 대한 공식 용어인 `unscoped`를 발생시킵니다. 새로운 C++11 대응 항목인 범위가 지정된 열거형 은 다음과 같은 방식으로 이름을 누출하지 않습니다.

열거형 클래스 Color { 검정, 흰색, 빨강 }; // 검정, 흰색, 빨강 // 범위는 Color

자동 흰색 = 거짓;

// 좋아, 다른 건 없어

```
// 범위 내 "흰색"

색상 c = 흰색; // 오류! 명명된 열거자가 없습니다.
// "white"는 이 범위에 있습니다.

색상 c = 색상::흰색; // 좋아

자동 c = 색상::흰색; // 또한 괜찮습니다(그리고 그에 따라
// 항목 5의 조언과 함께)
```

범위가 지정된 열거형은 "열거형 클래스"를 통해 선언되기 때문에 때때로 다음과 같이 참조됩니다.
열거형 클래스.

범위가 지정된 열거형이 제공하는 네임스페이스 오염의 감소는 다음을 수행하기에 충분한 이유입니다.
범위가 지정되지 않은 형제보다 더 선호하지만 범위가 지정된 열거형에는 두 번째로 매력적인 항목이 있습니다.
장점: 열거자가 훨씬 더 강력하게 형식화됩니다. unsco-를 위한 열거자
ped 열거형은 암시적으로 정수형으로 변환됩니다(그리고 거기에서 부동 소수점으로
유형). 따라서 다음과 같은 의미 오류는 완전히 유효합니다.

```
열거형 색상 { 검정, 흰색, 빨강 }; // 범위가 지정되지 않은 열거형
```

```
표준::벡터<표준::크기_t> 프라임 팩터 // 함수. 돌아오는
(표준::크기_t x); // x의 소인수
```

```
색상 c = 빨간색;
...
```

```
if (c < 14.5) { // Color를 double(!)과 비교합니다.

자동 요소 = // 소인수 계산
PrimeFactors(c); // 색상(!)
...
}
```

그러나 "enum" 다음에 간단한 "class"를 던져 범위가 지정되지 않은 enum을 변환합니다.
범위가 지정된 것으로, 그것은 매우 다른 이야기입니다. 암시적 변환이 없습니다.
범위가 지정된 열거형의 열거자에서 다른 유형으로:

```
열거형 클래스 Color { 검정, 흰색, 빨강 }; // 이제 열거형의 범위가 지정되었습니다.
```

```
색상 c = 색상::빨간색; // 이전과 같지만
... // 범위 한정자와 함께

if (c < 14.5) { // 오류! 비교할 수 없다
// 색상 및 이중
```

```

자동 요소 =
    PrimeFactors(c);
...
}

// 오류! // std::size_t를 기대하는 함수에
Color를 전달할 수 없습니다.

```

솔직히 Color에서 다른 유형으로의 변환을 수행하려면 유형 시스템을 원하는 대로 비틀기 위해 항상 하던 대로 하십시오. 캐스트를 사용하십시오.

```

if (static_cast<더블>(c) < 14.5) {
    ...
}

자동 요인 =
    PrimeFactors(static_cast<std::size_t>(c)); // 컴파일
}

// 이상한 코드이지만 // 유
효함

```

범위가 지정된 열거형은 범위가 지정되지 않은 열거형에 비해 세 번째 이점이 있는 것처럼 보일 수 있습니다. 범위가 지정된 열거형은 앞으로 선언될 수 있기 때문입니다. 즉, 열거자를 지정하지 않고 이름을 선언할 수 있습니다.

```

열거형 색상; // 오류!

```

이것은 오해의 소지가 있습니다. C++11에서 범위가 지정되지 않은 열거형도 앞으로 선언될 수 있지만 약간의 추가 작업 후에만 가능합니다. 작업은 C++의 모든 열거형이 컴파일러에 의해 결정되는 통합 기본 유형을 갖는다는 사실에서 비롯됩니다. Color와 같은 범위가 지정되지 않은 열거형의 경우,

```
열거형 색상 { 검정, 흰색, 빨강 };
```

컴파일러는 표현할 값이 세 개뿐이므로 기본 유형으로 char를 선택할 수 있습니다. 그러나 일부 열거형에는 훨씬 더 큰 값 범위가 있습니다. 예:

```

열거형 상태 { 양호 = 0, 실때 = 1, 불완
    전 = 100, 손상 =
    200, 미정 = 0xFFFFFFFF };

```

여기서 표현되는 값의 범위는 0에서 0xFFFFFFFF입니다. 특이한 기계(문자가 32비트 이상으로 구성된 경우)를 제외하고 컴파일러는 상태 값의 표현을 위해 char보다 큰 정수 유형을 선택해야 합니다.

메모리를 효율적으로 사용하기 위해 컴파일러는 열거자 값의 범위를 나타내기에 충분한 열거형에 대해 가장 작은 기본 유형을 선택하려고 합니다. 어떤 경우에는 컴파일러가 크기 대신 속도를 최적화하고 이 경우 허용되는 가장 작은 기본 유형을 선택하지 않을 수 있지만 크기에 대해 최적화할 수 있기를 확실히 원합니다. 이를 가능하게 하기 위해 C++98은 열거형 정의만 지원합니다(여기서 모든 열거자가 나열됨). 열거형 선언은 허용되지 않습니다. 이를 통해 컴파일러는 열거형이 사용되기 전에 각 열거형에 대한 기본 유형을 선택할 수 있습니다.

그러나 열거형을 앞으로 선언할 수 없다는 점에는 단점이 있습니다. 가장 주목할만한 것은 아마도 컴파일 의존성이 증가일 것입니다. Status 열거형을 다시 고려하십시오.

```
열거형 상태 { 양호 = 0, 실패 = 1, 불
    완전 = 100, 손
    상 = 200, 미정 =
    0xFFFFFFFF };
```

이것은 시스템 전체에서 사용될 가능성이 있는 일종의 열거형으로 시스템의 모든 부분이 의존하는 헤더 파일에 포함됩니다. 새로운 상태 값이 도입되면,

```
열거 상태 { 양호 = 0, 실패 = 1, 불완
    전 = 100, 손상
    = 200, 감사 = 500, 미정 =
    0xFFFFFFFF };
```

단일 하위 시스템(아마도 단일 함수만!)만이 새 열거자를 사용하더라도 전체 시스템을 다시 컴파일해야 할 가능성이 높습니다. 이것은 사람들이 싫어하는 종류입니다. 그리고 C++11에서 열거형을 정방향 선언하는 기능이 제거하는 종류입니다. 예를 들어, 다음은 범위가 지정된 열거형과 하나를 매개변수로 사용하는 함수의 완벽하게 유효한 선언입니다.

```
열거형 클래스 상태; // 포워드 선언
```

```
무효 continueProcessing(상태 s); // fwd로 선언된 열거형 사용
```

이러한 선언을 포함하는 헤더는 Status의 정의가 수정된 경우 다시 컴파일할 필요가 없습니다. 또한 상태가 수정되었지만(예: 감사된 열거자를 추가하기 위해) continueProcessing의 동작은 영향을 받지 않습니다(예):

continueProcessing은 감사를 사용하지 않음), continueProcessing의 구현도 다시 컴파일할 필요가 없습니다.

그러나 컴파일러가 enum을 사용하기 전에 enum의 크기를 알아야 하는 경우 C++98의 enum은 알 수 없는데 C++11의 enum은 어떻게 정방향 선언에서 벗어날 수 있습니까? 대답은 간단합니다. 범위가 지정된 열거형의 기본 유형은 항상 알려져 있으며 범위가 지정되지 않은 열거형의 경우 지정할 수 있습니다.

기본적으로 범위가 지정된 열거형의 기본 유형은 int입니다.

열거형 클래스 상태; // 기본 유형은 int입니다.

기본값이 적합하지 않으면 재정의할 수 있습니다.

열거형 클래스 상태: std::uint32_t; // 기본 유형
// 상태는 std::uint32_t입니다 //
(<cstdint>에서)

어느 쪽이든 컴파일러는 범위가 지정된 열거형에서 열거자의 크기를 알고 있습니다.

범위가 지정되지 않은 열거형의 기본 유형을 지정하려면 범위가 지정된 열거형과 동일한 작업을 수행하고 결과가 앞으로 선언될 수 있습니다.

열거형 색상: std::uint8_t; // 범위가 지정되지 않은 열거형에 대한 fwd
decl; // 기본 유형은 // std::uint8_t입니다.

기본 유형 사양은 열거형 정의를 따를 수도 있습니다.

열거형 클래스 상태: std::uint32_t { 양호 = 0, 실패 = 1, 불완전함 =
100, 손상됨 =
200, 감사됨 = 500, 미정
= 0xFFFFFFFF };

범위가 지정된 열거형이 네임스페이스 오염을 피하고 무의미한 암시적 유형 변환에 민감하지 않다는 사실을 고려할 때 범위가 지정되지 않은 열거형이 유용할 수 있는 상황이 적어도 한 번 있다는 사실을 듣고 놀랄 수 있습니다. C++11의 std::tuples 내의 필드를 참조할 때입니다. 예를 들어, 소셜 네트워킹 웹사이트에서 사용자의 이름, 이메일 주소 및 평판 값에 대한 값을 보유하는 튜플이 있다고 가정합니다.

UserInfo = // 별칭을 입력합니다. 항목 9 참조 // 이
std::tuple<std::string,

```
표준::문자열, 표           // 이메일 //
준::크기_t>;               평판
```

주석이 튜플의 각 필드가 나타내는 것을 나타내지만, 별도의 소스 파일에서 다음과 같은 코드를 접할 때는 별로 도움이 되지 않을 것입니다.

```
사용자 정보 ulInfo;           // 튜플 타입의 객체
```

```
...
```

```
자동 값 = std::get<1>(ulInfo); // 필드 1의 값 가져오기
```

프로그래머로서 추적해야 할 사항이 많습니다. 필드 1이 사용자의 이메일 주소에 해당한다는 사실을 기억해야 할까요? 나는 그렇게 생각하지 않는다. 범위가 지정되지 않은 열거형을 사용하여 이름을 필드 번호와 연결하면 다음이 필요하지 않습니다.

예제:

```
열거형 UserInfoFields { uiName, uiEmail, uiReputation };
```

```
사용자 정보 ulInfo;           // 이전과
```

```
...
```

```
자동 값 = std::get<uiEmail>(ulInfo); // 아, // 이메일 필드의 값을 얻습니다.
```

이 작업을 수행하는 것은 UserInfoFields에서 std::get에 필요한 유형인 std::size_t로의 암시적 변환입니다.

범위가 지정된 열거형이 있는 해당 코드는 훨씬 더 장황합니다.

```
열거형 클래스 UserInfoFields { uiName, uiEmail, uiReputation };
```

```
사용자 정보 ulInfo;           // 이전과
```

```
...
```

```
자동 val =
```

```
std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
(ulInfo);
```

열거자를 취하고 해당 std::size_t 값을 반환하는 함수를 작성하여 자세한 정보를 줄일 수 있지만 약간 까다롭습니다. std::get은 템플릿이고 제공하는 값은 템플릿 인수(괄호가 아닌 꺪쇠 괄호 사용에 주의)이므로 열거자를 std::size_t로 변환하는 함수는 실행 중에 결과를 생성해야 합니다. 편집. 항목 15 에서 설명하는 것처럼 이는 constexpr 함수여야 함을 의미합니다.

실제로 모든 종류의 열거형과 함께 작동해야 하므로 실제로 constexpr 함수 템플릿이어야 합니다. 일 반화를 하려면

반환 유형도 일반화하십시오. std::size_t를 반환하는 대신 열거형의 기본 유형을 반환합니다. std::underlying_type 유형 특성을 통해 사용할 수 있습니다. (유형 특성에 대한 정보는 항목 9 를 참조하십시오.) 마지막으로 예외를 생성하지 않을 것임을 알고 있기 때문에 noexcept로 선언합니다 (항목 14 참조). 결과는 임의의 열거자를 사용하고 해당 값을 컴파일 시간 상수로 반환할 수 있는 함수 템플릿 toUType입니다.

```
template<typename E>
constexpr typename std::underlying_type<E>::type toUType(E 열거자)
    noexcept {

    반품
    static_cast<유형 이름
        std::underlying_type<E>::type>(열거자);
}
```

C++14에서 toUType은 typename std::underlying_type<E>::type을 더 세련된 std::underlying_type_t로 대체하여 단순화할 수 있습니다(항목 9 참조).

```
template<typename E> // C++14
constexpr std::underlying_type_t<E> toUType(E 열거
    자) noexcept
{
    return static_cast<std::underlying_type_t<E>>(열거자);
}
```

더 세련된 자동 반환 유형(항목 3 참조)은 C++14에서도 유효합니다.

```
template<typename E> // C++14
constexpr auto toUType(E 열거
    자) noexcept {
    return static_cast<std::underlying_type_t<E>>(열거자);
}
```

작성 방법에 관계없이 toUType을 사용하면 다음과 같이 튜플의 필드에 액세스할 수 있습니다.

```
자동 값 = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

범위가 지정되지 않은 열거형을 사용하는 것보다 작성하는 것이 더 중요하지만 열거자와 관련된 네임스페이스 오염 및 부주의한 변환도 방지합니다. 대부분의 경우 디지털 통신의 최첨단 기술이 2400-2400 년대로 거슬러 올라가는 열거 기술의 함정을 피하는 능력에 대해 몇 개의 추가 문자를 입력하는 것이 합리적인 가격이라고 결정할 수 있습니다. 보 모뎀.

기억해야 할 사항

- C++98 스타일 열거형은 이제 범위가 지정되지 않은 열거형으로 알려져 있습니다. • 범위가 지정된 열거형의 열거자는 열거형 내에서만 볼 수 있습니다. 그들은 변환 캐스트로만 다른 유형에.
- 범위가 지정된 열거형과 범위가 지정되지 않은 열거형 모두 기본 유형의 사양을 지원합니다. 범위가 지정된 열거형의 기본 기본 유형은 int입니다. 범위가 지정되지 않은 열거형에는 기본 기본 유형이 없습니다.
- 범위가 지정된 열거형은 항상 앞으로 선언될 수 있습니다. 범위가 지정되지 않은 열거형은 선언이 기본 형식을 지정하는 경우에만 앞으로 선언될 수 있습니다.

항목 11: 정의되지 않은 `private`보다 삭제된 함수를 선호하라 것.

다른 개발자에게 코드를 제공하고 그들이 특정 함수를 호출하지 못하도록 하려면 일반적으로 함수를 선언하지 않습니다. 함수 선언이 없고 호출할 함수가 없습니다. 쉬워요. 그러나 때때로 C++는 함수를 선언하고 클라이언트가 해당 함수를 호출하지 못하도록 하려면 peasy가 더 이상 쉽지 않습니다.

상황은 "특수 멤버 함수", 즉 필요할 때 C++에서 자동으로 생성하는 멤버 함수에서만 발생합니다. 항목 17에서는 이러한 함수에 대해 자세히 설명하지만 지금은 복사 생성자와 복사 할당 연산자에 대해서만 걱정할 것입니다. 이 장은 주로 C++11의 더 나은 방법으로 대체된 C++98의 일반적인 방법에 대해 설명합니다. C++98에서 멤버 함수의 사용을 억제하려면 거의 항상 복사 생성자입니다, 할당 연산자 또는 둘 다.

이러한 함수의 사용을 방지하기 위한 C++98 접근 방식은 정의하지 않고 `private`로 선언하는 것입니다. 예를 들어, C++ 표준 라이브러리의 `iostreams` 계층 구조의 기본 근처에는 클래스 템플릿 `basic_ios`가 있습니다. 모든 `istream` 및 `ostream` 클래스는 이 클래스에서 (간접적으로) 상속합니다. `istream`과 `ostream`을 복사하는 것은 바람직하지 않습니다. 왜냐하면 그러한 작업이 무엇을 해야 하는지가 명확하지 않기 때문입니다. 예를 들어, `istream` 객체는 입력 값의 스트림을 나타내며, 그 중 일부는 이미 읽었을 수 있고 일부는 잠재적으로 나중에 읽을 수 있습니다. `istream`이 복사되는 경우 이미 읽은 모든 값과 미래에 읽을 모든 값을 복사해야 합니까? 그러한 질문을 처리하는 가장 쉬운 방법은 존재하지 않도록 정의하는 것입니다. 스트림 복사를 금지하는 것이 바로 그 일입니다.

istream 및 ostream 클래스를 복사할 수 없도록 렌더링하려면 C++98에서 basic_ios가 다음과 같이 지정됩니다(주석 포함).

```
템플릿 <클래스 charT, 클래스 특성 = char_traits<charT>> 클래스 basic_ios : 공개
ios_base { 공개:
```

...

개인:

```
basic_ios(const basic_ios&); // 정의되지 않음 basic_ios& operator=(const
basic_ios&); // 정의되지 않음 };
```

이러한 함수를 private로 선언하면 클라이언트가 해당 함수를 호출할 수 없습니다. 고의적으로 정의하지 못한다는 것은 아직 액세스 권한이 있는 코드(예: 클래스의 멤버 함수 또는 친구)가 이를 사용하는 경우 함수 정의 누락으로 인해 연결이 실패한다는 의미입니다.

C++11에는 본질적으로 동일한 목적을 달성하는 더 좋은 방법이 있습니다. "= delete"를 사용하여 복사 생성자와 복사 할당 연산자를 삭제된 함수로 표시합니다.

다음은 C++11에 지정된 basic_ios의 동일한 부분입니다.

```
템플릿 <클래스 charT, 클래스 특성 = char_traits<charT>> 클래스 basic_ios : 공개
ios_base { 공개:
```

...

```
basic_ios(const basic_ios&) = 삭제; basic_ios&
operator=(const basic_ios&) = 삭제;
```

...

};

이러한 함수를 삭제하는 것과 비공개로 선언하는 것의 차이점은 다른 어떤 것보다 유행의 문제처럼 보일 수 있지만 여기에는 생각하는 것보다 더 큰 내용이 있습니다. 삭제된 함수는 어떤 식으로든 사용할 수 없으므로 멤버 및 친구 함수에 있는 코드라도 basic_ios 개체를 복사하려고 하면 컴파일에 실패합니다. 이는 링크 타임까지 부적절한 사용이 진단되지 않는 C++98 동작에 비해 개선된 것입니다.

규칙에 따라 삭제된 함수는 비공개가 아닌 공개로 선언됩니다. 거기에는 이유가 있습니다. 클라이언트 코드가 멤버 함수를 사용하려고 하면 C++는 삭제된 상태 전에 액세스 가능성을 확인합니다. 클라이언트 코드가 삭제된 private 함수를 사용하려고 할 때 일부 컴파일러는 함수의 액세스 가능성이 실제로 사용할 수 있는지 여부에 영향을 미치지 않음에도 불구하고 함수가 private인 것에 대해서만 불평합니다. private-and-not-defined 멤버를 대체하기 위해 레거시 코드를 수정할 때 이것을 염두에 둘 가치가 있습니다.

새 기능을 공개하면 일반적으로 삭제된 기능이 있습니다.

더 나은 오류 메시지가 나타납니다.

삭제된 기능의 중요한 아점은 모든 기능이 삭제될 수 있다는 것입니다.

멤버 함수만 비공개일 수 있습니다. 예를 들어, 정수를 취하고 그것이 행운의 숫자인지 여부를 반환하는 비멤버 함수가 있다고 가정합니다.

```
bool isLucky(정수);
```

C++의 C 유산은 모호하게 보일 수 있는 거의 모든 유형을 의미합니다.

숫자는 암시적으로 int로 변환되지만 컴파일되는 일부 호출은 그렇지 않을 수 있습니다. 의미:

```
if (isLucky('a')) ... // "는 행운의 숫자입니까?
```

```
if (isLucky(true)) ... // 사실이다"?
```

```
if (isLucky(3.5)) ... // 3으로 잘라야 할까요?  
// 운을 확인하기 전에?
```

행운의 숫자가 정말로 정수여야 한다면, 우리는 다음과 같은 호출을 방지하고 싶습니다. 컴파일.

이를 수행하는 한 가지 방법은 원하는 형식에 대해 삭제된 오버로드를 만드는 것입니다. 필터링:

```
bool isLucky(정수); // 원래 함수
```

```
bool isLucky(char) = 삭제; // 문자 거부
```

```
bool isLucky(bool) = 삭제; // 부울 거부
```

```
bool isLucky(더블) = 삭제; // double을 거부하고  
// 부동
```

(더블과 플로트 둘 다

거절당했다는 사실이 당신을 놀라게 할 수도 있지만, 당신이 그것을 기억하고 나면 놀라움은 사라질 것입니다.

float를 int로 또는 double로 변환하는 것 중에서 선택이 주어지면 C++는

더블로 변환. 따라서 float를 사용하여 isLucky를 호출하면 double이 호출됩니다.

int가 아닌 오버로드. 글쎄, 그것은 노력할 것입니다. 과부하가 삭제된다는 사실은 호출이 컴파일되지 않도록 합니다.)

삭제된 기능은 사용할 수 없지만 프로그램의 일부입니다. 이와 같이,

과부하 해결 중에 고려됩니다. 그렇기 때문에 삭제된

위의 함수 선언에서 isLucky에 대한 바람직하지 않은 호출은 거부됩니다.

```
if (isLucky('a')) ... // 오류! 삭제된 함수 호출
```

```

if (isLucky(true)) ...           // 오류!
if (isLucky(3.5f)) ...           // 오류!

```

삭제된 함수가 수행할 수 있는 또 다른 트릭(비공개 멤버 함수는 수행할 수 없음)은 비활성화되어야 하는 템플릿 인스턴스화의 사용을 방지하는 것입니다. 예를 들어 내장 포인터와 함께 작동하는 템플릿이 필요하다고 가정합니다 ([4장의](#) 조언에도 불구하고 원시 포인터보다 스마트 포인터를 선호함).

```

template<typename T> 무
효 processPointer(T* ptr);

```

포인터의 세계에는 두 가지 특별한 경우가 있습니다. 하나는 역참조, 증가 또는 감소 등의 방법이 없기 때문에 `void*` 포인터입니다. 다른 하나는 일반적으로 개별 문자에 대한 포인터가 아니라 C 스타일 문자열에 대한 포인터를 나타내기 때문에 `char*` 포인터입니다. 이러한 특별한 경우는 종종 특별한 처리를 요구하며, `processPointer` 템플릿의 경우 적절한 처리가 이러한 유형을 사용하는 호출을 거부하는 것이라고 가정하겠습니다. 즉, `void*` 또는 `char*` 포인터로 `processPointer`를 호출할 수 없어야 합니다.

그것은 쉽게 시행됩니다. 해당 인스턴스를 삭제하기만 하면 됩니다.

```

template<>
무효 processPointer<void>(void*) = 삭제;

template<>
무효 processPointer<char>(char*) = 삭제;

```

이제 `void*` 또는 `char*`을 사용하여 `processPointer`를 호출하는 것이 유효하지 않은 경우 `const void*` 또는 `const char*`로 호출하는 것도 유효하지 않을 수 있으므로 일반적으로 이러한 인스턴스화도 삭제해야 합니다.

```

template<>
무효 processPointer<const 무효>(const 무효*) = 삭제;

template<>
무효 processPointer<const char>(const char*) = 삭제;

```

그리고 정말로 철저하게 하고 싶다면 `const volatile void*` 및 `const volatile char*` 오버로드도 삭제하고 다른 표준 문자 유형에 대한 포인터에 대한 오버로드 작업을 시작하게 됩니다.

`std::wchar_t`, `std::char16_t` 및 `std::char32_t`.

흥미롭게도 클래스 내부에 함수 템플릿이 있고 `private`로 선언하여 일부 인스턴스화를 비활성화하려는 경우(클래식 C++98 규칙) 멤버 함수를 제공할 수 없기 때문에 그렇게 할 수 없습니다. 템플릿 전 문화

기본 템플릿과 다른 액세스 수준. 예를 들어 processPointer가 위젯 내부의 멤버 함수 템플릿이고 void* 포인터에 대한 호출을 비활성화하려는 경우 컴파일되지는 않지만 C++98 접근 방식이 됩니다.

클래스 위젯 { 공개:

```
...
template<typename T> 무
효 processPointer(T* ptr) { ... }

개인: 템플릿
<> 무효 // 오류!
processPointer< 무효>(무효*);

};
```

문제는 템플릿 전문화를 클래스 범위가 아닌 네임스페이스 범위에서 작성해야 한다는 것입니다. 삭제된 기능에는 다른 액세스 수준이 필요하지 않으므로 이 문제가 발생하지 않습니다. 클래스 외부에서 삭제할 수 있습니다(따라서 네임스페이스 범위에서).

클래스 위젯 { 공개:

```
...
template<typename T> 무
효 processPointer(T* ptr) { ... }

...
};

템플릿<> 무효 // 아직
위젯::processPointer< 무효>( 무효*) = 삭제; // 공공의,
// 하지
만 // 삭제됨
```

진실은 함수를 private로 선언하고 정의하지 않는 C++98 관행은 실제로 C++11의 삭제된 함수가 실제로 달성하는 것을 달성하려는 시도라는 것입니다. 에뮬레이션으로서 C++98 접근 방식은 실제만큼 좋지 않습니다. 클래스 외부에서 작동하지 않고 클래스 내부에서 항상 작동하지 않으며 작동할 때 링크 타임까지 작동하지 않을 수 있습니다. 따라서 삭제된 기능을 고수하십시오.

기억할 사항 • 정의

되지 않은 비공개 함수보다 삭제된 함수를 선호합니

다. • 비회원 함수 및 템플릿을 포함한 모든 함수는 삭제할 수 있습니다.
인스턴스화.

항목 12: 재정의하는 함수 재정의를 선언하십시오 .

C++의 객체 지향 프로그래밍 세계는 클래스, 상속 및 가상 함수를 중심으로 이루어집니다. 이 세상에서 가장 기본적인 아이디어는 파생 클래스의 가상 함수 구현이 기본 클래스의 구현을 재정의한다는 것입니다. 가상 기능 재정의가 얼마나 쉽게 잘못될 수 있는지 깨닫는 것은 낙담합니다. 이 부분은 머피의 법칙이 단순히 준수되어야 하는 것이 아니라 존중되어야 한다는 생각으로 설계된 것과 같습니다.

"재정의"는 "오버로딩"과 매우 유사하지만 완전히 관련이 없기 때문에 가상 함수 재정의는 기본 클래스 인터페이스를 통해 파생 클래스 함수를 호출할 수 있게 하는 것임을 분명히 하겠습니다.

```

클래스 기반 { 공개:
    가상 무효
        doWork();                                // 기본 클래스 가상 함수
        ...
    };
}

클래스 파생: 공개 기본 { 공개: 가상 무효
    doWork();                                // Base::doWork를 재정의합니다. //
    ...
};

std::unique_ptr<베이스> upb =           // 기본 클래스 포인터 생성
    std::make_unique<Derived>(); // 파생 클래스 객체로; // std::make_unique에 대한 정
                                    // 보는 항목 21 을 참조 하세요.
    ...
    upb->doWork();                         // base를 통해 doWork를 호출합니다. //
                                              // class ptr; 파생 클래스 // 함수가 호출됨
}

```

재정의가 발생하려면 몇 가지 요구 사항이 충족되어야 합니다.

- 기본 클래스 함수는 가상이어야 합니다.
- 기본 및 파생 함수 이름은 동일해야 합니다.
소멸자).
- 기본 및 파생 함수의 매개변수 유형은 동일해야 합니다.
- 기본 및 파생 함수의 상수는 동일해야 합니다.
- 기본 및 파생 함수의 반환 유형과 예외 사양은 호환되어야 합니다.

C++98의 일부이기도 한 이러한 제약 조건에 C++11은 다음을 하나 더 추가합니다.

- 함수의 참조 한정자는 동일해야 합니다. 멤버 함수 참조 한정자는 C++11의 덜 알려진 기능 중 하나이므로 들어 본 적이 없더라도 놀라지 마십시오. 그들은 멤버 함수의 사용을 lvalue만 또는 rvalue로만 제한하는 것을 가능하게 합니다. 멤버 함수를 사용하기 위해 가상일 필요는 없습니다.

클래스 위젯 { 공개:

...

무효 doWork() &; // doWork의 이 버전은 // *이 값이 lvalue인 경우에
만 적용됩니다.

무효 doWork() && }; // doWork의 이 버전은 // *이 값이 rvalue일 때만
적용됩니다.

...

위젯 makeWidget(); // 팩토리 함수(rvalue 반환)

위젯 w; // 일반 객체(lvalue)

...

w.doWork(); // lvalue에 대해 Widget::doWork를 호출합니다. //
(예: Widget::doWork &)

makeWidget().doWork(); // rvalue에 대해 Widget::doWork를 호출합니다. // (예:
Widget::doWork &&)

참조 한정자가 있는 멤버 함수에 대해서는 나중에 자세히 설명하겠지만 지금은 기본 클래스의 가상 함수에 참조 한정자가 있는 경우 해당 함수의 파생 클래스 재정의가 정확히 동일한 참조를 가져야 한다는 점에 유의하십시오.

예선. 그렇지 않은 경우 선언된 함수는 파생 클래스에 계속 존재하지만 기본 클래스의 어떤 것도 재정의하지 않습니다.

재정의에 대한 이러한 모든 요구 사항은 작은 실수가 큰 차이를 만들 수 있음을 의미합니다. 재정의 오류가 포함된 코드는 일반적으로 유효하지만 그 의미는 의도한 것과 다릅니다. 그러므로 당신이 원가를 잘못했을 때 알려주는 컴파일러에 의존할 수 없습니다. 예를 들어 다음 코드는 완전히 합법적이고 언뜻 보기에는 합리적으로 보이지만 기본 클래스 함수에 연결된 단일 파생 클래스 함수가 아닌 가상 함수 재정의를 포함하지 않습니다. 각각의 경우에 문제를 식별할 수 있습니까? 즉, 각 파생 클래스 함수가 동일한 이름을 가진 기본 클래스 함수를 재정의하지 않는 이유는 무엇입니까?

```
클래스 베이스 { 공
개: 가상 무효
    mf1() const; 가상 무효 mf2(int x);
    가상 무효 mf3() &; 무효 mf4()
    const; };
```

```
클래스 파생: 공개 기본 { 공개: 가상 무효
    mf1(); 가상 무효 mf2(부호 없는 int x);
    가상 무효 mf3() &&; 무효 mf4()
    const; };
```

도움이 필요해?

- mf1은 Base에서는 const로 선언되지만 Derived에서는 선언되지 않습니다.
- mf2는 Base에서 int를 사용하지만 Derived에서는 unsigned int를 사용합니다.
- mf3은 Base에서 lvalue로 한정되지만 Derived에서는 rvalue로 한정됩니다.
- mf4는 Base에서 가상으로 선언되지 않습니다.

"이봐, 실제로 이런 것들은 컴파일러 경고를 불러일으킬 것이므로 걱정할 필요가 없다."라고 생각할 수도 있습니다. 사실일 수도 있습니다. 하지만 그렇지 않을 수도 있습니다. 내가 확인한 두 개의 컴파일러에서 코드는 불만 없이 수락되었으며 모든 경고가 활성화된 상태였습니다. (다른 컴파일러는 일부 문제에 대한 경고를 제공했지만 전부는 아닙니다.)

파생 클래스 재정의를 선언하는 것은 올바르게 하는 데 중요하지만 잘못되기 쉽기 때문에 C++11에서는 파생 클래스 함수가

기본 클래스 버전을 재정의해야 합니다. 재정의를 선언합니다. 이것을 위의 예제에 적용하면 다음과 같은 파생 클래스가 생성됩니다.

```
클래스 파생: 공개 기본 { 공개: 가상 무효
mf1() 재정의; 가상 무효 mf2(unsigned
int x) 재정의; 가상 무효 mf3() && 재정의;
가상 무효 mf4() const 재정의; };
```

물론 이것은 컴파일되지 않을 것입니다. 왜냐하면 이런 식으로 작성될 때 컴파일러는 모든 재정의 관련 문제에 대해 kvetch할 것이기 때문입니다. 그것이 바로 당신이 원하는 것이며, 이것이 모든 재정의 함수 재정의를 선언해야 하는 이유입니다.

컴파일을 수행하는 override를 사용하는 코드는 다음과 같습니다(목표가 Derived의 모든 함수가 Base의 가상 함수를 재정의하는 것이라고 가정).

```
클래스 베이스 { 공
개: 가상 무효
mf1() const; 가상 무효 mf2(int x);
가상 무효 mf3() &; 가상 무효 mf4()
const; };
```

```
클래스 파생: 공개 Base { 공개: 가상 무효
mf1() const 재정의; 가상 무효 mf2(int x)
재정의; 가상 무효 mf3() 및 재정의; 무효 mf4() const
재정의; };
```

// "가상"을 추가하는 것은 괜찮습니다
다. // 하지만 필수는 아닙니다.

이 예제에서 작업을 수행하는 일부는 Base에서 mf4 virtual을 선언하는 것과 관련이 있습니다. 대부분의 재정의 관련 오류는 파생 클래스에서 발생하지만 기본 클래스에서도 잘못될 수 있습니다.

모든 파생 클래스 재정의에 재정의를 사용하는 정책은 컴파일러가 예정 재정의가 아무 것도 재정의하지 않을 때 알려주는 것 이상을 수행할 수 있습니다. 또한 기본 클래스에서 가상 함수의 서명을 변경하려는 경우 결과를 측정하는 데 도움이 될 수 있습니다. 파생 클래스가 모든 곳에서 재정의를 사용하는 경우 서명을 변경하고 시스템을 재컴파일하고 얼마나 많은 손상을 입었는지(즉, 얼마나 많은 파생 클래스가 컴파일에 실패했는지) 확인한 다음 서명 변경이 문제의 가치가 있는지 여부를 결정할 수 있습니다. 재정의가 없으면 포괄적인 단위 테스트가 있어야 합니다.

기본 클래스 함수를 재정의해야 하지만 컴파일러 진단을 이끌어낼 필요는 없습니다.

C++에는 항상 키워드가 있었지만 C++11에는 override와 final의 두 가지 컨텍스트 키워드가 도입되었습니다. 2 이러한 키워드는 특정 컨텍스트에서만 예약되어 있다는 특성이 있습니다. override의 경우 멤버 함수 선언의 끝에서 발생하는 경우에만 예약된 의미를 갖습니다. 즉, 이미 이를 재정의를 사용하는 레거시 코드가 있는 경우 C++11에 대해 변경할 필요가 없습니다.

```
클래스 경고 { 공개:
    ...
    무효 재정의(); // C++98의 잠재적인 레거시 클래스
    ...
};

그것이 오버라이드에 대한 모든 것이지만 멤버 함수 참조 한정자에 대해 할 말이 전부는 아닙니다. 나는 나중에 그들에 대해 더 많은 정보를 제공하겠다고 약속했고 지금은 나중입니다.
```

lvalue 인수만 허용하는 함수를 작성하려면 const가 아닌 lvalue 참조 매개변수를 선언합니다.

```
무효 doSomething(위젯& w); // lvalue 위젯만 허용
rvalue 인수만 허용하는 함수를 작성하려면 rvalue 참조 매개변수를 선언합니다.
```

```
무효 doSomething(위젯&& w); // rvalue 위젯만 허용
멤버 함수 참조 한정자는 단순히 멤버 함수가 호출되는 객체에 대해 동일한 구분을 그리는 것을 가능하게 합니다(예: *this). 이것은 멤버 함수 선언의 끝에 있는 const와 정확히 유사합니다. 이는 멤버 함수가 호출되는 객체(즉, *this)가
```

상수

참조 한정 멤버 함수의 필요성은 일반적이지 않지만 발생할 수 있습니다.
예를 들어 Widget 클래스에 std::vector 데이터 멤버가 있고 클라이언트에 직접 액세스할 수 있는 접근자 함수를 제공한다고 가정합니다.

```
클래스 위젯 { 공개:
```

2 가상 함수에 final을 적용하면 파생 클래스에서 함수가 재정의되는 것을 방지할 수 있습니다. final은 클래스에도 적용될 수 있으며, 이 경우 해당 클래스를 기본 클래스로 사용할 수 없습니다.

```
DataType 사용 = std::vector<double>;
...
// "사용"에 대한 // 정보
는 항목 9 를 참조하십시오 .
```

```
DataType& data() { 반환 값; }
...
```

사적인:
데이터 유형 값; };

이것은 오늘날 빛을 발하는 가장 캡슐화된 디자인은 아니지만 이를 제쳐두고 이 클라이언트 코드에서 어떤 일이 일어나는지 고려하십시오.

```
위젯 w;
...
```

```
자동 vals1 = w.data(); // w.values를 vals1에 복사
```

Widget::data의 반환 유형은 lvalue 참조(정확하게는 std::vector<double>&)이고 lvalue 참조는 lvalue로 정의되어 있으므로 lvalue에서 vals1을 초기화합니다. 따라서 vals1은 주석에서 알 수 있듯이 w.values에서 생성된 복사본입니다.

이제 위젯을 생성하는 팩토리 함수가 있다고 가정합니다.

```
위젯 makeWidget();
```

makeWidget에서 반환된 위젯 내부의 std::vector로 변수를 초기화하려고 합니다.

```
자동 vals2 = makeWidget().data(); // 내부에 값 복사
// 위젯을 vals2로
```

다시 Widgets::data는 lvalue 참조를 반환하고 다시 lvalue 참조는 lvalue이므로 새 객체(vals2)는 위젯 내부의 값에서 복사 생성됩니다. 그러나 이번에는 Widget이 makeWidget(즉, rvalue)에서 반환된 임시 객체이므로 내부에 std::vector를 복사하는 것은 시간 낭비입니다.

이동하는 것이 좋지만 데이터가 lvalue 참조를 반환하기 때문에 C++ 규칙에 따라 컴파일러는 복사본에 대한 코드를 생성해야 합니다. ("마치 규칙"으로 알려진 것을 통해 최적화할 수 있는 약간의 여지가 있지만 컴파일러가 이를 활용하는 방법을 찾는 데 의존하는 것은 어리석은 일입니다.)

필요한 것은 데이터가 rvalue 위젯에서 호출될 때 결과도 rvalue여야 함을 지정하는 방법입니다. 참조 한정자를 사용하여 lvalue 및 rvalue 위젯에 대한 데이터를 오버로드하면 다음이 가능합니다.

```

클래스 위젯 { 공개:
    DataType 사용 =
        std::vector<double>;
    ...

    DataType& 데이터() && { 반
        환 값; }

    DataType 데이터() && { 반
        환 std::move(값); }

    ...
}

사적인:
    데이터 유형 값; };

```

// lvalue 위젯의 경우 // lvalue를
반환합니다.

// rvalue 위젯의 경우 // rvalue
를 반환합니다.

데이터 오버로드와 다른 반환 유형을 확인하십시오. lvalue 참조 오버로드는 lvalue 참조(즉,
lvalue)를 반환하고 rvalue 참조 오버로드는 임시 개체(즉, rvalue)를 반환합니다. 이것은 이제
클라이언트 코드가 우리가 원하는 대로 작동한다는 것을 의미합니다.

```

자동 vals1 = w.data();
// lvalue 오버로드를 호출합니다.
// Widget::data, copy- // vals1
생성

자동 vals2 = makeWidget().data(); // rvalue 오버로드를 호출합니다.
// Widget::data, move- //
vals2 생성

```

이것은 확실히 좋지만 이 해피엔딩의 따뜻한 빛이 이 아이템의 진정한 요점에서 당신을 산만하게
하지 않도록 하십시오. 요점은 기본 클래스의 가상 함수를 재정의하려는 파생 클래스의 함수를 선
언할 때마다 해당 함수 재정의를 선언해야 한다는 것입니다.

기억해야 할 사항

- 재정의하는 함수 재정의를 선언합니다. • 멤버
함수 참조 한정자를 사용하면 lvalue 및 rvalue 개체(*this)를 다르게 처리할 수 있습니다.

항목 13: 반복자 보다 const_iterator 를 선호하십시오.

const_iterators는 const에 대한 포인터와 동일한 STL입니다. 수정할 수 없는 값을 가리킵니다. 가능하면 항상 const를 사용하는 표준 관행은 반복자가 필요할 때마다 const_iterators를 사용해야 하지만 반복자가 가리키는 대상을 수정할 필요는 없다고 지시합니다.

이는 C++98의 경우 C++11의 경우와 동일하지만 C++98의 const_iterators는 중간 지원만 했습니다. 만들기가 쉽지 않았고, 한번 갖게 되면 사용할 수 있는 방법이 제한적이었습니다. 예를 들어, std::vector<int>에서 1983년(프로그래밍 언어의 이름으로 "C++"이 "C with Classes"로 대체됨)이 처음으로 검색된 다음 값 1998(첫 번째 ISO C++ 표준이 채택된 연도) 해당 위치에서. 벡터에 1983이 없으면 삽입은 벡터의 끝으로 가야 합니다. C++98에서 반복자를 사용하면 쉬웠습니다.

```
std::vector<int> 값;
...
std::vector<int>::iterator it =
    std::find(values.begin(), values.end(), 1983); values.insert(it,
1998);
```

그러나 이 코드는 반복자가 가리키는 것을 수정하지 않기 때문에 여기서 반복자는 실제로 적절한 선택이 아닙니다. const_iterator를 사용하도록 코드를 수정하는 것은 간단해야 하지만 C++98에서는 전혀 그렇지 않았습니다. 다음은 개념적으로 건전하지만 여전히 올바르지 않은 접근 방식입니다.

typedef std::vector<int>::iterator IterT; // typedef std::vector<int>::const_iterator
를 입력합니다. ConstIterT; // 정의

```
std::vector<int> 값;
...
ConstIterT ci =
    std::find(static_cast<ConstIterT>(values.begin()), // 캐스트
              static_cast<ConstIterT>(values.end())); // 캐스트 1983);

values.insert(static_cast<IterT>(ci), 1998); // 컴파일되지
                                                않을 수 있습니다. // 아래 참조
```

물론 `typedef`는 필수는 아니지만 코드의 캐스트를 작성하기 쉽게 만듭니다. (별칭 선언을 사용하라는 항목 9 의 조언을 따르는 대신 `typedef`를 표시하는 이유가 궁금하다면 이 예제가 C++98 코드를 보여주고 별칭 선언이 C++11의 새로운 기능이기 때문입니다.)

값이 비 `const` 컨테이너이고 C++98에서는 비 `const` 컨테이너에서 `const_iterator`를 가져오는 간단한 방법이 없었기 때문에 `std::find` 호출 시 캐스트가 존재합니다. 다른 방법으로 `const_iterator`를 얻을 수 있기 때문에 캐스트가 꼭 필요한 것은 아니지만(예: 값을 참조에 대한 참조 변수에 바인딩 한 다음 코드에서 값 대신 해당 변수를 사용할 수 있음) 한 가지 방법은 또는 `const_iterator`를 `non-const` 컨테이너의 요소로 가져오는 프로세스에는 약간의 왜곡이 포함됩니다.

`const_iterator`가 있으면 문제가 더 악화되는 경우가 많습니다. C++98에서 삽입(및 삭제) 위치는 반복자에 의해서만 지정될 수 있기 때문입니다. `const_iterators`는 허용되지 않았습니다. 그렇기 때문에 위의 코드에서 `const_iterator`(`std::find`에서 매우 조심스럽게 가져옴)를 `iterator`로 캐스팅했습니다. 삽입하기 위해 `const_iterator`를 전달하면 컴파일되지 않습니다.

솔직히 말해서 내가 보여준 코드는 컴파일되지 않을 수도 있습니다. 왜냐하면 `static_cast`가 아니라 `const_iterator`에서 `iterator`로 이식 가능한 변환이 없기 때문입니다. `reinterpret_cast`로 알려진 시맨틱 망치도 이 작업을 수행할 수 없습니다. (그것은 C++98 제한이 아닙니다. C++11에서도 마찬가지입니다. `const_iterators`는 아무리 많이 보여도 반복자로 변환하지 않습니다.) 반복자를 생성하는 이식 가능한 방법이 몇 가지 있습니다. `const_iterators`가 수행하는 지점이지만 명확하지 않고 보편적으로 적용할 수 없으며 이 책에서 논의할 가치가 없습니다. 게다가, 나는 지금쯤 내 요점이 분명하기를 바랍니다. `const_iterators`는 C++98에서 너무 많은 문제를 겪었고, 거의 그럴 가치가 없었습니다. 하루가 끝나면 개발자는 가능할 때마다 `const`를 사용하지 않고 실용적일 때마다 사용하며 C++98에서 `const_iterators`는 그다지 실용적이지 않았습니다.

C++11에서 변경된 모든 것. 이제 `const_iterator`는 얻기도 쉽고 사용하기도 쉽습니다. 컨테이너 멤버 함수 `cbegin` 및 `cend`는 `const_iterator`를 생성하며, 비 `const` 컨테이너에 대해서도 반복자를 사용하여 위치를 식별(예: 삽입 및 지우기)하는 STL 멤버 함수는 실제로 `const_iterator`를 사용합니다. C++11에서 반복자를 사용하여 `const_iterator`를 사용하는 원래 C++98 코드를 수정하는 것은 정말 간단합니다.

```
std::vector<int> 값; // 이전과  
...  
자동 = // cbegin 사용
```

```
std::find(values.cbegin(), values.cend(), 1983); // 그리고 센드
values.insert(it, 1998);
```

이제 실용적인 `const_iterators`를 사용하는 코드입니다!

`const_iterators`에 대한 C++11의 지원이 약간 부족한 유일한 상황은 최대한 일반적인 라이브러리 코드를 작성하려는 경우입니다. 이러한 코드는 일부 컨테이너 및 컨테이너와 유사한 데이터 구조가 멤버가 아닌 비멤버 함수로 시작 및 종료(+ `cbegin`, `cend`, `rbegin` 등)를 제공한다는 점을 고려합니다. 예를 들어 내장 배열의 경우이며 인터페이스가 자유 함수로만 구성된 일부 타사 라이브러리의 경우이기도 합니다. 따라서 최대한 일반적인 코드는 멤버 버전의 존재를 가정하지 않고 비멤버 함수를 사용합니다.

예를 들어 작업한 코드를 `findAnd`로 일반화할 수 있습니다.

다음과 같이 템플릿을 삽입합니다.

```
template<typename C, typename V> 무효
findAndInsert(C& 컨테이너, const V& targetVal,
              const V& insertVal)
// 컨테이너에서 // targetVal의 //
// 첫 번째 항목을 찾은 다음 //
// insertVal을 // 삽입합니다.

{ std::cbegin 사용; std::cend
  를 사용하여;

  auto it = std::find(cbegin(container), // 비멤버 cbegin // 비멤버 cend
                      cend(컨테이너),
                      targetVal);

  container.insert(그것, insertVal); }
```

이것은 C++14에서 잘 작동하지만 슬프게도 C++11에서는 작동하지 않습니다. C++11은 표준화 중 감독을 통해 비멤버 함수 `begin` 및 `end`를 추가했지만 `cbegin`, `cend`, `rbegin`, `rend`, `crbegin` 및 `crend`를 추가하지 못했습니다. C++14는 이러한 간과를 수정합니다.

C++11을 사용 중이고 최대한 일반적인 코드를 작성하고 사용 중인 라이브러리 중 비멤버 `cbegin` 및 친구들을 위한 누락된 템플릿을 제공하지 않는 경우 자신의 구현을 쉽게 함께 던질 수 있습니다. 예를 들어, 다음은 비멤버 `cbegin`의 구현입니다.

```
템플릿 <클래스 C> auto
cbegin(const C& 컨테이너)->decltype(std::begin(컨테이너)) {
```

```
반환 std::begin(컨테이너); // 아래 설명 참조
}
```

비회원 `cbegin`이 회원 `begin`을 호출하지 않는 것을 보고 놀랍지 않습니까? 나도 마찬가지였다. 하지만 논리를 따르라. 이 `cbegin` 템플릿은 컨테이너와 같은 데이터 구조 C를 나타내는 모든 유형의 인수를 허용하고 `const`에 대한 참조 매개변수인 컨테이너를 통해 이 인수에 액세스합니다. C가 기존의 컨테이너 유형(예: `std::vector<int>`)인 경우 컨테이너는 해당 컨테이너의 `const` 버전에 대한 참조가 됩니다 (예: `const std::vector<int>&`). `const` 컨테이너에서 비멤버 시작 함수(C++11에서 제공)를 호출하면 `const_iterator`가 생성되고 해당 반복자는 이 템플릿이 반환하는 것입니다. 이런 식으로 구현하는 것의 장점은 시작 멤버 함수(컨테이너의 경우 C++11의 비멤버 시작 호출)를 제공하지만 `cbegin` 멤버를 제공하지 못하는 컨테이너에서도 작동한다는 것입니다. 따라서 시작만 지원하는 컨테이너에서 이 비멤버 `cbegin`을 사용할 수 있습니다.

이 템플릿은 C가 내장 배열 유형인 경우에도 작동합니다. 이 경우 컨테이너는 `const` 배열에 대한 참조가 됩니다. C++11은 배열의 첫 번째 요소에 대한 포인터를 반환하는 배열에 대해 비멤버 `begin`의 특수 버전을 제공합니다. `const` 배열의 요소는 `const`이므로 `const` 배열에 대해 비멤버 `begin`이 반환하는 포인터는 `const`에 대한 포인터이고 `const`에 대한 포인터는 실제로 배열에 대한 `const_iterator`입니다. (템플릿이 내장 배열에 대해 어떻게 특화될 수 있는지에 대한 통찰력을 얻으려면 배열에 대한 참조 매개변수를 사용하는 템플릿의 유형 추론에 대한 항목 1의 논의를 참조하십시오.)

그러나 기본으로 돌아가십시오. 이 항목의 요점은 가능하면 `const_iterators`를 사용하도록 권장하는 것입니다. 의미가 있을 때마다 `const`를 사용하는 근본적인 동기는 C++11보다 이전에 있었지만 C++98에서는 반복자로 작업할 때 실용적이지 않았습니다. C++11에서는 매우 실용적이며 C++14는 C++11이 남긴 미완성 비즈니스를 정리합니다.

기억해야 할 사항

- 반복자보다 `const_iterator`를 선호합니다.
- 최대한 일반적인 코드에서는 비멤버 버전의 `begin`, `end`, `rbegin` 등을 해당 멤버 함수에 사용합니다.

항목 14: 예외를 내보내지 않을 경우 함수 noexcept를 선언 하십시오.

C++98에서 예외 사양은 다소 변덕스러운 짐승이었습니다. 함수가 방출할 수 있는 예외 유형을 요약 해야 했기 때문에 함수의 구현이 수정된 경우 예외 사양도 수정이 필요할 수 있습니다. 호출자가 원래 예외 사양에 종속될 수 있으므로 예외 사양을 변경하면 클라이언트 코드가 손상될 수 있습니다. 컴파일러는 일반적으로 함수 구현, 예외 사양 및 클라이언트 코드 간의 일관성을 유지하는 데 도움을 주지 않았습니다. 대부분의 프로그래머는 궁극적으로 C++98 예외 사양이 문제가 될 가치가 없다고 결정했습니다.

C++11 작업 중에 함수의 예외 발생 동작에 대한 진정으로 의미 있는 정보는 포함 여부에 대한 합의가 나타났습니다. 검정이든 흰색이든, 함수는 예외를 발생시키거나 발생하지 않을 것이라고 보장했습니다.

이 이분법은 C++98을 본질적으로 대체하는 C++11 예외 사양의 기초를 형성합니다. (C++98 스타일 예외 사양은 계속 유효하지만 더 이상 사용되지 않습니다.) C++11에서 무조건 noexcept는 예외를 내보내지 않도록 보장하는 함수를 위한 것입니다.

함수를 그렇게 선언해야 하는지 여부는 인터페이스 디자인의 문제입니다. 함수의 예외 발생 동작은 클라이언트의 주요 관심사입니다. 호출자는 함수의 noexcept 상태를 쿼리할 수 있으며 이러한 쿼리의 결과는 호출 코드의 예외 안전성이나 효율성에 영향을 줄 수 있습니다. 따라서 함수가 noexcept인지 여부는 멤버 함수가 const인지 여부만큼 중요한 정보입니다. 예외를 내보내지 않을 것임을 알면서도 noexcept 함수를 선언하지 않는 것은 단순히 인터페이스 사양이 좋지 않은 것입니다.

그러나 예외를 생성하지 않는 함수에 noexcept를 적용하는 추가 인센티브가 있습니다. 컴파일러가 더 나은 목적 코드를 생성할 수 있도록 합니다. 그 이유를 이해하려면 함수가 예외를 내보내지 않을 것이라고 말하는 C++98과 C++11 방식의 차이점을 조사하는 것이 도움이 됩니다. 호출자에게 예외를 받지 않겠다고 약속하는 함수 f를 고려하십시오. 이를 표현하는 두 가지 방법은 다음과 같습니다.

```
int f(int x) throw(); // f에서 예외 없음: C++98 스타일
```

```
int f(int x) 예외 없음; // f: C++11 스타일에서 예외 없음
```

런타임에 예외가 f를 벗어나면 f의 예외 사양이 위반됩니다. C++98 예외 사양을 사용하면 호출 스택이 f의 호출자에게 해제되고 여기에 관련되지 않은 일부 작업 후에 프로그램 실행이 종료됩니다. C++11 예외 사양을 사용하면 런타임 동작이 약간 다릅니다. 스택은 프로그램 실행이 종료되기 전에만 해제될 수 있습니다.

호출 스택을 해제하는 것과 가능한 한 해제하는 것의 차이는 코드 생성에 놀라울 정도로 큰 영향을 미칩니다. noexcept 함수에서 옵티마이저는 예외가 함수 밖으로 전파되는 경우 런타임 스택을 해제할 수 없는 상태로 유지할 필요가 없으며 예외가 함수를 떠나는 경우 noexcept 함수의 객체가 생성의 역순으로 파괴되도록 보장해서는 안 됩니다.. .

"throw()" 예외 사양이 있는 함수는 예외 사양이 전혀 없는 함수와 마찬가지로 최적화 유연성이 부족합니다. 상황은 다음과 같이 요약될 수 있습니다.

RetType 함수(params) noexcept; // 가장 최적화 가능

RetType 함수(매개변수) throw(); // 덜 최적화됨

RetType 함수(매개변수); // 덜 최적화됨

이것만으로도 예외를 생성하지 않는다는 것을 알 때마다 함수 noexcept를 선언할 충분한 이유가 됩니다.

일부 기능의 경우 케이스가 더 강력합니다. 이동 작업이 그 대표적인 예입니다. std::vector<Widget>을 사용하는 C++98 코드 기반이 있다고 가정합니다. 위젯은 때때로 push_back을 통해 std::vector에 추가됩니다.

표준::벡터<위젯> vw;

...

위젯 w;

...

// w로 작업

vw.push_back(w); // vw에 w 추가

...

이 코드가 제대로 작동하고 C++11용으로 수정하는 데 관심이 없다고 가정합니다.

그러나 C++11의 이동 의미 체계가 이동 가능 유형이 포함될 때 레거시 코드의 성능을 향상시킬 수 있다는 사실을 이용하고 싶습니다. 따라서 위젯이 직접 작성하거나 자동 생성 조건이 충족되는지 확인하여 위젯에 이동 작업이 있는지 확인합니다(항목 17 참조).

새 요소가 std::vector에 추가되면 std::vector에 공간이 부족할 수 있습니다. 즉, std::vector의 크기가 해당 용량과 같을 수 있습니다. 그런 일이 발생하면 std::vector는 더 큰 새 메모리 청크를 할당하여

요소를 제거하고 기존 메모리 청크에서 새 청크로 요소를 전송합니다. C++98에서는 이전 메모리에서 새 메모리로 각 요소를 복사한 다음 이전 메모리에 있는 개체를 삭제하여 전송을 수행했습니다. 이 접근 방식을 통해 `push_back`은 강력한 예외 안전 보장을 제공할 수 있었습니다. 요소를 복사하는 동안 예외가 발생하면 `std::vector`의 상태가 변경되지 않은 상태로 유지됩니다. 새 메모리에 성공적으로 복사되었습니다.

C++11에서 자연스러운 최적화는 `std::vector` 요소의 복사를 이동으로 바꾸는 것입니다. 불행히도 이렇게 하면 `push_back`의 예외 안전 보장을 위반할 위험이 있습니다. n 개의 요소가 이전 메모리에서 이동되었고 요소 $n+1$ 을 이동하는 예외가 발생하면 `push_back` 작업을 완료할 때까지 실행할 수 없습니다. 그러나 원래의 `std::vector`가 수정되었습니다. n 개의 요소가 이동되었습니다. 각 개체를 원래 메모리로 다시 이동하려고 하면 예외가 발생할 수 있으므로 원래 상태로 복원하는 것이 불가능할 수 있습니다.

이는 레거시 코드의 동작이 `push_back`의 강력한 예외 안전 보장에 의존할 수 있기 때문에 심각한 문제입니다. 따라서 C++11 구현은 이동 작업이 예외를 내보내지 않는다는 사실이 알려져 있지 않는 한 `push_back` 내부의 복사 작업을 이동으로 자동으로 바꿀 수 없습니다. 이 경우 복사본을 이동하는 것이 안전하고 유일한 부작용은 성능이 향상됩니다.

`std::vector::push_back`은 "가능하면 이동하지만 필요한 경우 복사" 전략을 활용하며 표준 라이브러리에서 그렇게 하는 유일한 기능은 아닙니다. C++98에서 강력한 예외 안전 보장을 자랑하는 다른 함수(예: `std::vector::reserve`, `std::deque::insert` 등)도 같은 방식으로 작동합니다. 이러한 모든 함수는 이동 작업이 예외를 내보내지 않는 것으로 알려진 경우에만 C++98의 복사 작업 호출을 C++11의 이동 작업 호출로 대체합니다. 그러나 함수가 이동 작업이 예외를 생성하지 않는지 어떻게 알 수 있습니까? 대답은 분명합니다. 작업이 `noexcept`로 선언되었는지 확인합니다.

상

스왑 기능은 `noexcept`가 특히 바람직한 또 다른 경우를 포함합니다. 스왑은 많은 STL 알고리즘 구현의 핵심 구성 요소이며 일반적으로 복사 할당 연산자에도 사용됩니다. 광범위한 사용은 `noexcept`가 제공하는 최적화를 특히 가치 있게 만듭니다. 흥미롭게도 표준 라이브러리의 스왑이 `noexcept`인지 여부는 때때로 사용자가

3 검사는 일반적으로 원형 교차로입니다. `std::vector::push_back`과 같은 함수는 `std::move_if_noexcept`를 호출합니다.

`std::move`는 유형의 이동 생성자가 `noexcept`인지 여부에 따라 조건부로 `rvalue`([항목 23 참조](#))로 변환하는 변형입니다. 차례로 `std::move_if_noexcept`는 `std::is_nothrow_move_constructible`을 참조하고 이 유형 특성([항목 9 참조](#))의 값은 이동 생성자가 `noexcept`(또는 `throw()`) 지정을 가지고 있는지 여부에 따라 컴파일러에 의해 설정됩니다.

정의된 스왑은 예외가 아닙니다. 예를 들어 표준 라이브러리의 배열 및 std::pair 스왑 선언은 다음과 같습니다.

```
템플릿 <class T, size_t N> 무효 스왑
(T&a)[N], // 보다
T (&b)[N]) noexcept(noexcept(swap(*a, *b))); // 아래에

템플릿 <클래스 T1, 클래스 T2> 구조체 쌍 {
    ...
    무효 스왑(쌍& p) noexcept(noexcept(swap(first, p.first)) && noexcept(swap(second,
        p.second)));
    ...
};
```

이러한 함수는 조건부로 noexcept입니다. noexcept인지 여부는 noexcept 절 내부의 표현식이 noexcept인지 여부에 따라 다릅니다. 예를 들어 Widget의 두 배열이 주어지면 배열의 개별 요소를 교체하는 것이 noexcept인 경우, 즉 Widget에 대한 교체가 noexcept인 경우에만 교체가 noexcept입니다. 따라서 위젯 스왑의 작성자는 위젯의 스왑 배열이 noexcept인지 여부를 결정합니다.

이는 차례로 Widget 배열의 배열과 같은 다른 스왑이 예외가 없는지 여부를 결정합니다. 마찬가지로 위젯을 포함하는 두 개의 std::pair 객체를 교체하는 것이 noexcept인지 여부는 위젯에 대한 교체가 noexcept인지 여부에 따라 다릅니다. 상위 수준 데이터 구조를 교체하는 것은 일반적으로 하위 수준 구성 요소를 교체하는 것이 noexcept인 경우에만 noexcept가 될 수 있다는 사실이 가능하면 언제든지 noexcept 교체 기능을 제공하도록 동기를 부여해야 합니다.

지금쯤이면 noexcept가 제공하는 최적화 기회에 대해 흥분하셨기를 바랍니다. 아아, 나는 당신의 열정을 진정시켜야합니다. 최적화도 중요하지만 정확성이 더 중요합니다. 이 항목의 시작 부분에서 noexcept는 함수 인터페이스의 일부이므로 장기적으로 noexcept 구현을 커밋할 의사가 있는 경우에만 noexcept 함수를 선언해야 한다고 언급했습니다. 함수 noexcept를 선언하고 나중에 그 결정을 후회한다면, 당신의 선택은 암울합니다.

함수 선언에서 noexcept를 제거할 수 있으므로(즉, 인터페이스 변경) 클라이언트 코드가 손상될 위험이 있습니다. 예외가 이스케이프할 수 있도록 구현을 변경할 수 있지만 원래의(지금은 잘못된) 예외 사양을 유지할 수 있습니다. 그렇게 하면 예외가 함수를 떠나려고 하면 프로그램이 종료됩니다. 또는 처음부터 구현을 변경하려는 열망을 촉발시킨 모든 것을 버리고 기존 구현을 포기할 수 있습니다.

이러한 옵션 중 어느 것도 매력적이지 않습니다.

문제는 대부분의 기능이 예외 중립적이라는 것입니다. 이러한 함수는 자체적으로 예외를 발생시키지 않지만 호출하는 함수는 예외를 발생시킬 수 있습니다. 그 때

발생하면 예외 중립 기능을 사용하면 방출된 예외가 호출 체인의 위쪽에 있는 처리기로 전달될 수 있습니다. 예외 중립 함수는 예외가 아닙니다. "그냥 통과" 예외를 내보낼 수 있기 때문입니다. 따라서 대부분의 함수에는 noexcept 지정이 적절하게 부족합니다.

그러나 일부 함수에는 예외가 발생하지 않는 자연스러운 구현이 있으며, 특히 이동 작업과 스왑과 같은 몇 가지 기능에 대해서는 noexcept가 상당한 결과를 가져올 수 있으므로 가능하면 noexcept 방식으로 구현하는 것이 좋습니다.⁴ 함수는 예외를 절대로 내보내지 않아야 한다고 정직하게 말할 수 있습니다. 확실히 noexcept를 선언해야 합니다.

일부 함수에는 자연스러운 noexcept 구현이 있다고 말했습니다.

noexcept 선언을 허용하기 위해 함수 구현을 왜곡하는 것은 꼬리를 훈드는 것입니다. 말 앞에 수레를 놓고 있다. 나무를 위해 숲을 보는 것이 아니다. ... 좋아하는 비유를 선택하십시오. 간단한 함수 구현이 예외를 생성할 수 있는 경우(예: throw할 수 있는 함수를 호출하여) 호출자로부터 이를 숨기기 위해 건너뛸 수 있습니다(예: 모든 예외를 포착하고 상태 코드 또는 특수 반환 값)은 함수 구현을 복잡하게 할 뿐만 아니라 일반적으로 호출 사이트에서도 코드를 복잡하게 만듭니다. 예를 들어 호출자는 상태 코드 또는 특수 반환 값을 확인해야 할 수 있습니다. 이러한 복잡성(예: 추가 분기, 명령어 캐시에 더 많은 압력을 가하는 더 큰 기능 등)의 런타임 비용은 noexcept를 통해 달성하고자 하는 속도 향상을 초과할 수 있으며 이해하고 유지하기가 더 어렵습니다. 그것은 형편없는 소프트웨어 엔지니어링 일 것입니다.

일부 기능의 경우 noexcept가 매우 중요하며 기본적으로 그렇게 되어 있습니다. C++98에서는 메모리 할당 해제 기능(예: operator delete 및 operator delete[])과 소멸자가 예외를 내보내는 것을 허용하지 않는 스타일로 간주되었으며 C++11에서는 이 스타일 규칙이 다음으로 업그레이드되었습니다. 언어 규칙. 기본적으로 모든 메모리 할당 해제 함수와 모든 소멸자(사용자 정의 및 컴파일러 생성 모두)는 암시적으로 noexcept입니다. 따라서 noexcept를 선언할 필요가 없습니다.

(그렇게 하는 것은 아무 것도 해를 끼치지 않으며 단지 비관습적입니다.) 소멸자가 암시적으로 noexcept가 아닌 유일한 경우는 클래스의 데이터 멤버(상속된 멤버 및 다른 데이터 멤버 내부에 포함된 멤버 포함)가 명시적으로 다음을 나타내는 유형일 때입니다. 소멸자가 예외를 내보낼 수 있음을 나타냅니다(예: "noexcept(false)"로 선언).

이러한 소멸자는 흔하지 않습니다. 표준 라이브러리에는 없고

⁴ 표준 라이브러리의 컨테이너에 대한 이동 작업에 대한 인터페이스 사양에는 예외가 없습니다. 그러나 구현자는 표준 라이브러리 기능에 대한 예외 사양을 강화할 수 있으며 실제로는 최소한 일부 컨테이너 이동 작업이 noexcept로 선언되는 것이 일반적입니다. 그 관행은 이 항목의 조언을 예시합니다. 예외가 발생하지 않도록 컨테이너 이동 작업을 작성할 수 있다는 것을 알게 된 구현자는 표준에서 요구하지 않더라도 종종 작업 noexcept를 선언합니다.

표준 라이브러리에서 사용 중인 객체의 소멸자(예: 컨테이너에 있거나 알고리즘에 전달되었기 때문에)는 예외를 내보내고 프로그램의 동작은 정의되지 않습니다.

일부 라이브러리 인터페이스 디자이너는 계약이 넓은 기능과 좁은 계약이 있는 기능을 구별합니다. 계약이 넓은 함수에는 전제 조건이 없습니다. 이러한 함수는 프로그램 상태에 관계없이 호출될 수 있으며 호출자가 전달하는 인수에 제약을 가하지 않습니다.⁵ 넓은 계약을 가진 함수는 정의되지 않은 동작을 나타내지 않습니다.

넓은 계약이 없는 기능은 좁은 계약을 가집니다. 이러한 기능의 경우 전제 조건이 위반되면 결과가 정의되지 않습니다.

넓은 계약으로 함수를 작성하고 예외를 내보내지 않을 것임을 알고 있다면 이 항목의 조언을 따르고 noexcept를 선언하는 것은 쉽습니다. 계약이 좁은 기능의 경우 상황이 더 까다롭습니다. 예를 들어, std::string 매개변수를 사용하는 f 함수를 작성하고 f의 자연적 구현에서 예외가 발생하지 않는다고 가정합니다. 이는 f가 noexcept로 선언되어야 함을 시사합니다.

이제 f에 전제 조건이 있다고 가정합니다. std::string 매개변수의 길이는 32자를 초과하지 않아야 합니다. f가 길이가 32보다 큰 std::string으로 호출되는 경우 정의에 따른 전제 조건 위반으로 인해 정의되지 않은 동작이 발생하기 때문에 동작이 정의되지 않습니다. f는 이 전제 조건을 확인할 의무가 없습니다. 함수가 전제 조건이 충족된다고 가정할 수 있기 때문입니다. (호출자는 그러한 가정이 유효한지 확인할 책임이 있습니다.) 전제 조건이 있더라도 f noexcept를 선언하는 것이 적절해 보입니다.

```
무효 f(const std::string& s) noexcept; // 전제 조건: //
s.length() <= 32
```

그러나 f의 구현자가 전제 조건 위반을 확인하기로 선택했다고 가정합니다.

확인은 필수는 아니지만 금지된 것도 아니며 전제 조건을 확인하는 것은 예를 들어 시스템 테스트 중에 유용할 수 있습니다. throw된 예외를 디버깅하는 것은 일반적으로 정의되지 않은 동작의 원인을 추적하는 것보다 쉽습니다.

그러나 테스트 하니스 또는 클라이언트 오류 처리기가 이를 감지할 수 있도록 전제 조건 위반을 어떻게 보고해야 할까요? 간단한 접근 방식은 "사전 조건이 위반되었습니다" 예외를 throw하는 것이지만 f가 noexcept로 선언되면 불가능합니다. 예외를 던지면 프로그램이 종료됩니다. 이를 위해-

⁵ "프로그램 상태에 관계없이" 및 "제약 없음"은 동작이 이미 정의되지 않은 프로그램을 합법화하지 않습니다. 예를 들어, std::vector::size는 넓은 계약을 가지고 있지만 std::vector로 캐스팅한 임의의 메모리 청크에서 호출하는 경우 합리적으로 동작할 필요는 없습니다. 캐스트의 결과는 정의되지 않았으므로 캐스트를 포함하는 프로그램에 대한 동작 보장이 없습니다.

아들, 넓은 계약과 좁은 계약을 구별하는 라이브러리 디자이너는 일반적으로 넓은 계약이 있는 기능을 제외하지 않습니다.

마지막으로 컴파일러는 일반적으로 함수 구현과 예외 사양 간의 불일치를 식별하는 데 도움이 되지 않는다는 이전 관찰에 대해 자세히 설명하겠습니다. 완벽하게 합법적인 다음 코드를 고려하십시오.

```
무효 설정(); 무효 정  
리(); // 다른 곳에 정의된 함수

무효 doWork() noexcept { 설정  
(); // 수행할 작업 설정  
... // 실제 작업을 수행

    대청소(); } // 정리 작업 수행
```

여기에서 `doWork`는 `noexcept`가 아닌 함수 설정 및 정리를 호출하더라도 `noexcept`로 선언됩니다. 이것은 모순되는 것처럼 보이지만 그렇게 선언되지 않았음에도 예외를 내보내지 않는 설정 및 정리 문서일 수 있습니다. `non-noexcept` 선언에는 타당한 이유가 있을 수 있습니다. 예를 들어, 그것들은 C로 작성된 라이브러리의 일부일 수 있습니다. (std 네임스페이스로 이동된 C 표준 라이브러리의 함수조차도 예외 사양이 없습니다. 예를 들어 `std::strlen`은 `noexcept`로 선언되지 않습니다.) 또는 C++98 예외 사양을 사용하지 않기로 결정했고 아직 C++11용으로 수정되지 않은 C++98 라이브러리의 일부여야 합니다.

`noexcept` 함수가 `noexcept` 보장이 없는 코드에 의존하는 정당한 이유가 있기 때문에 C++에서는 이러한 코드를 허용하고 컴파일러는 일반적으로 이에 대한 경고를 발행하지 않습니다.

기억해야 할 사항

- `noexcept`는 함수 인터페이스의 일부이며 이는 호출자가 그것에 의존하십시오.
- `noexcept` 함수는 `non-noexcept` 함수보다 더 최적화할 수 있습니다. • `noexcept`는 이동 작업, 스왑, 메모리 할당 해제 기능 및 소멸자에 특히 유용합니다.
- 대부분의 함수는 `noexcept`보다는 예외 중립적입니다.

항목 15: 가능하면 constexpr 을 사용하세요 .

C++11에서 가장 혼란스러운 새 단어에 대한 상이 있다면 constexpr이 아마도 수상할 것입니다. 객체에 적용할 때는 본질적으로 const의 강화된 형식이지만 기능에 적용할 때는 완전히 다른 의미를 갖습니다. constexpr이 표현하고자 하는 것과 일치할 때 분명히 사용하고 싶기 때문에 혼란을 없애는 것은 문제의 가치가 있습니다.

개념적으로 constexpr은 상수일 뿐만 아니라 컴파일 중에 알려진 값을 나타냅니다. 개념은 이야기의 일부일 뿐이지만 constexpr이 기능에 적용될 때 상황이 이것이 시사하는 것보다 더 미묘한 차이가 있기 때문입니다. 놀라운 결말을 막지 않도록 지금은 constexpr 함수의 결과가 const라고 가정할 수 없으며 컴파일 중에 해당 값을 알고 있다는 것을 당연하게 여길 수도 없습니다. 아마도 가장 흥미롭게도 이러한 것들은 가능입니다. constexpr 함수는 const 또는 컴파일 중에 알려진 결과를 생성할 필요가 없다는 것이 좋습니다!

하지만 constexpr 객체부터 시작하겠습니다. 이러한 객체는 실제로 const이며 실제로 컴파일 시간에 알려진 값을 갖습니다. (기술적으로 번역하는 동안 값이 결정되며 번역은 컴파일뿐만 아니라 링크까지 포함됩니다. 그러나 C++용 컴파일러나 링커를 작성하지 않는 한 이것은 사용자에게 영향을 미치지 않으므로 값이 constexpr 객체는 컴파일 중에 결정되었습니다.)

컴파일하는 동안 알려진 값은 권한이 있습니다. 예를 들어, 읽기 전용 메모리에 배치할 수 있으며, 특히 임베디드 시스템 개발자에게 이는 상당히 중요한 기능이 될 수 있습니다. 더 광범위하게 적용할 수 있는 것은 상수이고 컴파일 중에 알려진 적분 값이 C++에서 적분 상수 표현식이 필요한 컨텍스트에서 사용될 수 있다는 것입니다. 이러한 컨텍스트에는 배열 크기의 지정, 통합 템플릿 인수(std::array 객체의 길이 포함), 열거자 값, 정렬 지정자 등이 포함됩니다. 이런 종류의 일에 변수를 사용하려면 반드시 constexpr로 선언해야 합니다. 그러면 컴파일러가 컴파일 시간 값이 있는지 확인하기 때문입니다.

```
정수 sz; // 비 constexpr 변수
...
constexpr 자동 배열크기1 = sz; // 오류! sz의 값은 // 컴파일 시 알 수
                               없음
표준::배열<int, sz> 데이터1; // 오류! 같은 문제
constexpr 자동 배열크기2 = 10; // 좋아요, 10은
```

// 컴파일 타임 상수

```
std::array<int, arraySize2> data2; // 좋아, arraySize2
// constexpr입니다.
```

const 객체는 컴파일 중에 알려진 값으로 초기화할 필요가 없기 때문에 const는 constexpr과 동일한 보장을 제공하지 않습니다.

정수 sz;	// 이전과
--------	--------

...

const 자동 배열 크기 = sz;	// 좋아요, arraySize는 // sz의 const 복사본입니다.
----------------------	--

std::array<int, arraySize> 데이터;	// 오류! arraySize의 값 // 컴파일 시 알 수 없음
---------------------------------	--

간단히 말해서 모든 constexpr 객체는 const이지만 모든 const 객체가 constexpr인 것은 아닙니다. 컴파일러가 컴파일 타임 상수가 필요한 컨텍스트에서 사용할 수 있는 값을 변수에 포함하도록 하려면 도달할 도구는 const가 아니라 constexpr입니다.

constexpr 객체에 대한 사용 시나리오는 constexpr 함수가 포함될 때 더 흥미로워집니다. 이러한 함수는 컴파일 시간 상수와 함께 호출될 때 컴파일 시간 상수를 생성합니다. 런타임까지 알려지지 않은 값으로 호출되면 런타임 값을 생성합니다. 이것은 그들이 무엇을 할 지 모르는 것처럼 들릴지 모르지만 그것에 대해 생각하는 것은 잘못된 방법입니다. 그것을 보는 올바른 방법은 다음과 같습니다.

- constexpr 함수는 컴파일 시간 상수를 요구하는 컨텍스트에서 사용할 수 있습니다. 이러한 컨텍스트에서 constexpr 함수에 전달하는 인수의 값이 컴파일 중에 알려진 경우 결과는 컴파일 중에 계산됩니다. 컴파일하는 동안 인수 값을 알 수 없는 경우 코드가 거부됩니다.
- constexpr 함수가 컴파일 중에 알려지지 않은 하나 이상의 값으로 호출되면 일반 함수처럼 작성하여 런타임에 결과를 계산합니다. 이것은 동일한 작업을 수행하기 위해 두 개의 함수가 필요하지 않다는 것을 의미합니다. 하나는 컴파일 시간 상수를 위한 것이고 다른 하나는 다른 모든 값을 위한 것입니다. constexpr 함수가 모든 작업을 수행합니다.

다양한 방법으로 실행할 수 있는 실험의 결과를 저장하기 위해 데이터 구조가 필요하다고 가정합니다. 예를 들어 조명 수준은 팬 속도 및 온도와 같이 실험 과정에서 높거나 낮거나 꺼질 수 있습니다. 실험과 관련된 n개의 환경 조건이 있는 경우 각 조건에는 세 가지 가능성성이 있습니다.

ble 상태에서 조합의 수는 $3n$ 입니다. 따라서 모든 조건 조합에 대한 실험 결과를 저장하려면 $3n$ 값을 위한 충분한 공간이 있는 데이터 구조가 필요합니다.

각 결과가 int이고 컴파일 중에 n 이 알려져 있거나 계산할 수 있다고 가정하면 std::array가 합리적인 데이터 구조 선택이 될 수 있습니다. 그러나 컴파일하는 동안 $3n$ 을 계산하는 방법이 필요합니다. C++ 표준 라이브러리는 우리가 필요로 하는 수학적 기능인 std::pow를 제공하지만, 우리의 목적을 위해 두 가지 문제가 있습니다. 첫째, std::pow는 부동 소수점 유형에서 작동하며 적분 결과가 필요합니다. 둘째, std::pow는 constexpr이 아니므로(즉, 컴파일 시간 값으로 호출할 때 컴파일 시간 결과를 반환하는 것이 보장되지 않음) std::array의 크기를 지정하는 데 사용할 수 없습니다.

다행히도 우리는 필요한 포우를 쓸 수 있습니다. 잠시 후 이를 수행하는 방법을 보여주겠지만 먼저 선언 및 사용 방법을 살펴보겠습니다.

```
constexpr // pow는 절대 던지지 않는 constexpr
int pow(int base, int exp) noexcept { // 함수입니다.

    ...
    // impl은 아래에 있습니다.

}

constexpr 자동 numConds = 5; // 조건 #

std::array<int, pow(3, numConds)> 결과; // 결과에는 //  $3^{\text{numConds}}$  // 요소가 있습니다.
                                            // 있습니다.
```

pow 앞의 constexpr은 pow가 const 값을 반환한다고 말하지 않고 base와 exp가 컴파일 시간 상수인 경우 pow의 결과가 컴파일 시간 상수로 사용될 수 있다고 말합니다. base 및/또는 exp가 컴파일 타임 상수가 아닌 경우 pow의 결과는 런타임에 계산됩니다. 즉, pow는 std::array의 크기를 컴파일 타임 계산하는 것과 같은 작업을 수행하기 위해 호출될 수 있을 뿐만 아니라 다음과 같은 런타임 컨텍스트에서도 호출될 수 있습니다.

```
자동 베이스 = readFromDB("베이스"); 자동 // 이 값을 // 런타임에 가져옵
exp = readFromDB("자수");           // 니다.

자동 baseToExp = pow(기본, 특급); // pow 함수 호출 // 런타임에
```

constexpr 함수는 컴파일 시간 값으로 호출될 때 컴파일 시간 결과를 반환할 수 있어야 하므로 구현에 제한이 적용됩니다.
제한 사항은 C++11과 C++14에서 다릅니다.

C++11에서 constexpr 함수는 하나의 실행 가능한 명령문만 포함할 수 있습니다. 그것은 두 가지 트릭을 사용할 수 있기 때문에 그것보다 더 제한적으로 들립니다.

`constexpr` 함수의 표현력을 당신이 생각하는 것 이상으로 확장합니다.

첫째, 조건부 "?:" 연산자를 `if-else` 문 대신 사용할 수 있고, 둘째, 루프 대신 재귀를 사용할 수 있습니다. 따라서 `pow`는 다음과 같이 구현할 수 있습니다.

```
constexpr int pow(int base, int exp) noexcept {
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

이것은 작동하지만, 하드 코어 함수형 프로그래머를 제외하고는 누구도 이를 예쁘다고 생각할 것이라고 상상하기 어렵습니다. C++14에서는 `constexpr` 함수에 대한 제한이 상당히 완화되어 다음 구현이 가능합니다.

```
constexpr int pow(int base, int exp) noexcept { // C++14
    자동 결과 = 1; for (int i
    = 0; i < exp; ++i) 결과 *= 기본;
    반환 결과;
}
```

`constexpr` 함수는 리터럴 유형을 가져오고 반환하는 것으로 제한되며, 이는 본질적으로 컴파일 중에 결정된 값을 가질 수 있는 유형을 의미합니다. C++11에서는 `void`을 제외한 모든 내장 유형이 적합하지만 생성자 및 기타 멤버 함수가 `constexpr`일 수 있으므로 사용자 정의 유형도 리터럴일 수 있습니다.

```
클래스 포인트 { 공
개: constexpr 포인트
트(더블 xVal = 0, 더블 yVal = 0) noexcept: x(xVal), y(yVal) {}
```

```
constexpr 이중 xValue() const noexcept { 반환 x; } constexpr 이중 yValue()
const noexcept { 반환 y; }
```

```
무효 setX(더블 newX) noexcept { x = newX; } 무효 setY(더블
newY) noexcept { y = newY; }
```

```
개인: 더블
x, y; };
```

여기에서 `Point` 생성자는 `constexpr`로 선언될 수 있습니다. 왜냐하면 컴파일 중에 전달된 인수가 알려진 경우 `con-`

structed Point는 컴파일 중에도 알 수 있습니다. 따라서 초기화된 포인트는 constexpr이 될 수 있습니다.

`constexpr` 포인트 p1(9.4, 27.7);

// 괜찮음, "실행" constexpr // 컴파일 중 ctor

`constexpr` 포인트 p2(28.8, 5.3);

// 역시 좋다

유사하게, getter xValue 및 yValue는 constexpr일 수 있습니다. 왜냐하면 컴파일 중에 알려진 값으로 Point 객체에서 호출되는 경우(예: constexpr Point 객체), 데이터 멤버 x 및 y의 값은 컴파일 중에 알 수 있기 때문입니다. -. 이를 통해 Point의 getter를 호출하는 constexpr 함수를 작성하고 이러한 함수의 결과로 constexpr 객체를 초기화할 수 있습니다.

`constexpr`

포인트 중간점(const Point& p1, const Point& p2) noexcept {

```
return { (p1.xValue() + p2.xValue()) / 2, (p1.yValue() +
    p2.yValue()) / 2 }; // constexpr 호출
} // 멤버 함수
```

`constexpr` 자동 중간 = 중간점(p1, p2);

// constexpr 초기화 //
constexpr 함수의 // 결과가 있는 객체

이것은 매우 흥미진진합니다. 이는 초기화가 생성자, getter 및 비멤버 함수에 대한 호출을 포함하지만 객체 mid를 읽기 전용 메모리에서 생성할 수 있음을 의미합니다! 이는 템플릿에 대한 인수 또는 열거자의 값을 지정하는 표현식에서 mid.xValue() * 10과 같은 표현식을 사용할 수 있음을 의미합니다! 런타임이 흐려지기 시작하고 전통적으로 런타임에 수행된 일부 계산이 컴파일 시간으로 마이그레이션될 수 있습니다. 마이그레이션에 참여하는 코드가 많을수록 소프트웨어가 더 빨리 실행됩니다. (그러나 컴파일은 더 오래 걸릴 수 있습니다.)

C++11에서는 두 가지 제한 사항으로 인해 Point의 멤버 함수 setX 및 setY가 constexpr로 선언되지 않습니다. 첫째, 작동하는 개체를 수정하고 C++11에서 constexpr 멤버 함수는 암시적으로 const입니다. 둘째, void 반환 유형이 있으며 void는 C++11에서 리터럴 유형이 아닙니다. 이러한 제한은 모두 C++14에서 해제되므로 C++14에서는 Point의 setter도 constexpr이 될 수 있습니다.

6 Point::xValue는 double을 반환하므로 mid.xValue() * 10의 형식도 double입니다. 부동 소수점 형식은 템플릿을 인스턴스화하거나 열거자 값을 지정하는 데 사용할 수 없지만 정수 형식을 생성하는 더 큰 식의 일부로 사용할 수 있습니다. 예를 들어 static_cast<int>(mid.xValue() * 10)는 템플릿을 인스턴스화하거나 열거자 값을 지정하는 데 사용할 수 있습니다.

클래스 포인트 { 공개:

...

`constexpr` 무효 `setX(이중 newX) noexcept { x = newX; }` // C++14

`constexpr` 무효 `setY(이중 newY) noexcept { y = newY; }` // C++14

...

};

이를 통해 다음과 같은 기능을 작성할 수 있습니다.

```
// 원점에 대한 p의 반사 반환(C++14) constexpr Point reflection(const Point& p)
noexcept {

    포인트 결과;                                // const가 아닌 Point 생성

    결과.setX(-p.xValue()); 결과.setY(-
        p.y값());                                // x 및 y 값을 설정합니다.

    반환 결과;                                  // 그것의 복사본을 반환
}
```

클라이언트 코드는 다음과 같을 수 있습니다.

```
constexpr 포인트 p1(9.4, 27.7); constexpr           // 위와 같이
포인트 p2(28.8, 5.3); constexpr 자동 중간
= 중간점(p1, p2);

constexpr 자동 ReflectedMid = 반사(중           // ReflectedMid의 값은 // (-19.1
    간);                                         -16.5)이고 // 컴파일 중에 알려짐
```

이 항목의 조언은 가능할 때마다 `constexpr`을 사용하는 것입니다. 이제 `constexpr` 객체와 `constexpr` 함수 모두 비 `constexpr` 객체 및 함수보다 더 넓은 범위의 컨텍스트에서 사용할 수 있기를 바랍니다. 가능할 때마다 `constexpr`을 사용하면 객체와 기능이 사용될 수 있는 상황의 범위를 최대화할 수 있습니다.

`constexpr`은 객체 또는 함수 인터페이스의 일부라는 점에 유의하는 것이 중요합니다. `constexpr`은 "C++에서 상수 표현식이 필요한 컨텍스트에서 사용할 수 있습니다."라고 선언합니다. 객체 또는 함수 `constexpr`을 선언하면 클라이언트가 이를 사용할 수 있습니다.

그러한 맥락. 나중에 `constexpr` 사용이 실수였다고 판단하고 이를 제거하면 임의로 많은 양의 클라이언트 코드가 컴파일을 중지할 수 있습니다.

(디버깅이나 성능 조정을 위해 함수에 I/O를 추가하는 간단한 작업은 I/O 문이 일반적으로 `constexpr` 함수에서 허용되지 않기 때문에 이러한 문제로 이어질 수 있습니다.) "가능한 경우 항상 `constexpr` 사용"의 일부는 당신이 그것을 적용하는 객체와 기능에 부과하는 제약에 장기적으로 혼신하려는 당신의 의지입니다.

기억해야 할 사항

- `constexpr` 객체는 `const`이며 동안 알려진 값으로 초기화됩니다.
편집.
- `constexpr` 함수는 다음과 같이 호출될 때 컴파일 타임 결과를 생성할 수 있습니다.
컴파일 중에 값이 알려진 인수.
- `constexpr` 객체와 함수는 더 넓은 범위의 컨텍스트에서 사용될 수 있습니다.
비 `constexpr` 객체 및 기능보다.
- `constexpr`은 객체 또는 함수 인터페이스의 일부입니다.

항목 16: `const` 멤버 함수를 스레드로부터 안전하게 보호하세요.

수학적 영역에서 작업하는 경우 다항식을 나타내는 클래스를 사용하는 것이 편리할 수 있습니다. 이 클래스 내에서 다항식의 근(들), 즉 다항식이 0으로 평가되는 값을 계산하는 함수를 갖는 것이 유용할 것입니다. 이러한 함수는 다항식을 수정하지 않으므로 `const`로 선언하는 것이 자연스럽습니다.

```
클래스 다항식 { 공개: 사용
RootsType =
    std::vector<double>;           // 값을 보유하는 데이터 구조 // 다항식이 0으로 평
                                    // 가되는 곳 // ("사용"에 대한 정보는 항목 9 참조)
...
RootsType 루트() const;
...
};

}

다항식의 근을 계산하는 것은 비용이 많이 들 수 있으므로 필요하지 않은 경우 수행하고 싶지 않습니다. 그리고 만약 우리가 그것을 해야 한다면, 우리는 확실히 그것을 두 번 이상 하고 싶지 않습니다. 따라서 다음을 계산해야 하는 경우 다항식의 루트를 캐시합니다.
```

캐시된 값을 반환하기 위해 루트를 구현합니다. 기본 접근 방식은 다음과 같습니다.

```
클래스 다항식 { 공개: 사용
RootsType =
    std::vector<double>;

RootsType 루트() const {
    if (!rootsAreValid) { // 캐시가 유효하지 않은 경우
        ...
        // 루트 계산, // rootVals
        //에 저장
        rootAreValid = 참;
    }

    rootVals를 반환합니다.
}

개인: 변경
가능한 bool rootAreValid{ false }; 변경 가능한 // 이니셜라이저에 대한 정보는 항
RootsType rootVals{}; }; 목 7 을 참조하십시오 .
```

개념적으로 루트는 작동하는 다항식 개체를 변경하지 않지만 캐싱 활동의 일부로 rootVals 및 rootAreValid를 수정해야 할 수도 있습니다.

이것이 mutable의 고전적인 사용 사례이며 이것이 이러한 데이터 멤버에 대한 선언의 일부인 이유입니다.

이제 두 개의 스레드가 동시에 다항식 개체에서 루트를 호출한다고 상상해 보십시오.

다항식 p;

...

/----- 스레드 1 ----- */*

자동 rootOfP = p.roots();

/----- 스레드 2 ----- */*

자동 valsGivingZero = p.roots();

이 클라이언트 코드는 완벽하게 합리적입니다. roots는 const 멤버 함수이며 이는 읽기 작업을 나타냅니다. 여러 스레드가 동기화 없이 읽기 작업을 수행하는 것은 안전합니다. 적어도 그래야 합니다. 이 경우에는 그렇지 않습니다. 왜냐하면 루트 내부에서 이러한 스레드 중 하나 또는 둘 모두가 데이터 멤버인 rootAreValid 및 rootVals를 수정하려고 할 수 있기 때문입니다. 즉, 이 코드는 다음과 같이 다를 수 있습니다.

동기화 없이 동일한 메모리를 읽고 쓰는 다른 스레드, 이것이 데이터 경쟁의 정의입니다. 이 코드에는 정의되지 않은 동작이 있습니다.

문제는 루트가 `const`로 선언되었지만 스레드로부터 안전하지 않다는 것입니다. `const` 선언은 C++ 98에서와 마찬가지로 C++ 11에서도 정확하므로(다항식의 근을 검색해도 다항식 값이 변경되지 않음) 수정이 필요한 것은 스레드 안전.

문제를 해결하는 가장 쉬운 방법은 뮤텍스를 사용하는 일반적인 방법입니다.

```
클래스 다항식 { 공개: 사용
RootsType =
    std::vector<double>;

RootsType 루트() const {

    std::lock_guard<std::mutex> g(m); // 뮤텍스 잠금

    if (!rootsAreValid) { // 캐시가 유효하지 않은 경우
        ...
        // 루트 계산/저장

        rootAreValid = 참;
    }

    rootVals를 반환합니다. // 뮤텍스 잠금 해제
}

private: 가
변 std::mutex m; 변경 가능한
bool rootAreValid{ false }; 변경 가능한 RootsType
rootVals{}; };
```

`std::mutex m`은 변경 가능으로 선언됩니다. 그 이유는 잠금 및 잠금 해제가 비 `const` 멤버 함수이고 루트(`const` 멤버 함수) 내에서 그렇지 않으면 `m`이 `const` 개체로 간주되기 때문입니다.

`std::mutex`은 이동 전용 유형(즉, 이동할 수 있지만 복사할 수 없는 유형)이기 때문에 다항식에 `m`을 추가하는 부작용은 다항식이 복사할 수 있는 기능을 잃는다는 점에 주목할 가치가 있습니다. 그러나 여전히 이동할 수 있습니다.

어떤 상황에서는 뮤텍스가 과도합니다. 예를 들어, 당신이 하는 모든 것이 멤버 함수가 호출된 횟수를 세는 것이라면 `std::atomic` 카운터(즉, 다른 스레드가 작업이 분할되지 않고 발생하는 것을 확인하도록 보장되는 카운터 - [항목 40 참조](#))는 종종 저렴하게 갈 수 있는 방법. (실제로 더 저렴한지는

실행 중인 하드웨어와 표준 라이브러리의 뮤텍스 구현.) 다음은 std::atomic을 사용하여 호출 수를 계산하는 방법입니다.

클래스 포인트 { 공개:

// 2D 포인트

...

double distanceFromOrigin() const noexcept {

// 항목 14 참조 //
noexcept의 경우

++콜카운트;

// 원자 증분

반환 std::sqrt((x * x) + (y * y));
}

개인: 변경

가능 std::atomic<unsigned> callCount{ 0 }; 이중 x, y; };

std::mutexes와 마찬가지로 std::atomics는 이동 전용 유형이므로 Point에 Count 호출이 있다는 것은 Point도 이동 전용임을 의미합니다.

std::atomic 변수에 대한 작업은 종종 뮤텍스 획득 및 해제보다 저렴하기 때문에 std::atomics에 더 많이 의존하고 싶은 유혹을 받을 수 있습니다. 예를 들어, 계산 비용이 많이 드는 int를 캐싱하는 클래스에서 뮤텍스 대신 한 쌍의 std::atomic 변수를 사용하려고 할 수 있습니다.

클래스 위젯 { 공개:

...

int magicValue() const {

(cacheValid)가 cachedValue를 반환하면; else { 자동 val1 = 비싼 계산1(); 자동 val2 = 비싼 계산2(); 캐시 된 값 = val1 + val2; 캐시 유효 = true; 캐시된 값을 반환 합니다.

// 어 오, 파트 1 // 어 오,
파트 2

}

사적인:

```
가변 std::atomic<bool> cacheValid{ false }; 가변 std::atomic<int>
cachedValue; };
```

이것은 효과가 있지만 때로는 생각보다 훨씬 더 열심히 일할 것입니다. 고려하다:

- 스레드는 Widget::magicValue를 호출하고, cacheValid를 false로 보고, 두 가지 값비싼 계산을 수행하고, 그 합계를 cachedValue에 할당합니다.
- 그 시점에서 두 번째 스레드는 Widget::magicValue를 호출하고 cacheValid도 false로 간주 하므로 첫 번째 스레드가 방금 완료한 것과 동일한 비용이 많이 드는 계산을 수행합니다. (이 "두 번째 스레드"는 실제로 여러 다른 스레드일 수 있습니다.)

이러한 동작은 캐싱의 목표에 위배됩니다. cachedValue 및 CacheValid에 대한 할당 순서를 반대로 하면 해당 문제가 제거되지만 결과는 다음과 같습니다.

[더 나쁜:](#)

클래스 위젯 { 공개:

...

```
int magicValue() const {
    (cacheValid)가 cachedValue를 반환하면; else { 자
    동 val1 = 비싼 계산1(); 자동 val2 = 비싼 계산2(); 캐시
    유효 = true; 반환 캐시된 값 = val1 + val2;

    // 어 오, 파트 1 // 어 오,
    // 파트 2
}
}
```

...

};

cacheValid가 false라고 가정하고 다음을 수행합니다.

- 한 스레드는 Widget::magicValue를 호출하고 cacheValid가 true로 설정된 지점까지 실행합니다.
- 그 순간 두 번째 스레드가 Widget::magicValue를 호출하고 캐시 Valid를 확인합니다. 그것이 사실이라면 쓰레드는 cachedValue를 반환합니다.

스레드가 아직 할당하지 않았습니다. 따라서 반환된 값은 잘못된.

여기 교훈이 있습니다. 동기화가 필요한 단일 변수 또는 메모리 위치의 경우 `std::atomic` 사용이 적절하지만 한 단위로 조작이 필요한 두 개 이상의 변수 또는 메모리 위치에 도달하면 뮤텍스에 도달해야 합니다. `Widget::magicValue`의 경우 다음과 같습니다.

클래스 위젯 { 공개:

...

```
int magicValue() const
{ std::lock_guard<std::mutex>
    가드(m); // 잠금 m

    (cacheValid)가 cachedValue를 반환하면; else { 자
    동 val1 = 비싼 계산1(); 자동 val2 = 비싼 계산2(); 캐시
    된 값 = val1 + val2; 캐시 유효 = true; 캐시된 값을 반환
    합니다.
```

```
} } // m 잠금 해제
...
```

private: **가**

```
변 std::mutex m; 가변 int
cachedValue; 변경 가능한 bool
cacheValid{ false };};

// 더 이상 원자가 아님 // 더
// 이상 원자가 아님
```

이제 이 항목은 여러 스레드가 객체에 대해 `const` 멤버 함수를 동시에 실행할 수 있다는 가정을 전제로 합니다. 그렇지 않은 경우(객체에서 해당 멤버 함수를 실행하는 스레드가 둘 이상 없다는 것을 보장할 수 있는) `const` 멤버 함수를 작성하는 경우 함수의 스레드 안전성은 중요하지 않습니다. 예를 들어, 단일 스레드 전용으로 설계된 클래스의 멤버 함수가 스레드로부터 안전한지는 중요하지 않습니다. 이러한 경우 `mutex` 및 `std::atomics`와 관련된 비용은 물론 이동 전용 클래스를 포함하는 렌더링의 부작용을 피할 수 있습니다. 그러나 이러한 스레딩이 없는 시나리오는 점점 더 흔하지 않으며 점점 더 드물어질 것입니다. 안전한 방법은 `const` 멤버 함수가

따라서 const 멤버 함수가 스레드로부터 안전한지 확인해야 합니다.

기억해야 할 사항

- const 멤버 함수가 절대 스레드 안전하지 않을 것이라고 확신하지 않는 한 스레드로부터 안전하게 만듭니다. 동시에 컨텍스트에서 사용됩니다.
- std::atomic 변수를 사용하면 럭스보다 더 나은 성능을 제공할 수 있지만 단일 변수 또는 메모리 위치 조작에만 적합합니다.

항목 17: 특수 멤버 함수 생성을 이해하십시오.

공식 C++ 용어에서 특수 멤버 함수는 C++에서 자체적으로 생성하려는 함수입니다. C++98에는 기본 생성자, 소멸자, 복사 생성자, 복사 할당 연산자의 네 가지 기능이 있습니다. 물론 작은 글씨도 있습니다. 이러한 함수는 필요한 경우에만 생성됩니다. 즉, 일부 코드에서 클래스에 명시적으로 선언되지 않은 상태에서 해당 함수를 사용하는 경우입니다. 기본 생성자는 클래스가 생성자를 전혀 선언하지 않은 경우에만 생성됩니다. (이렇게 하면 컴파일러가 생성자 인수가 필요하다고 지정한 클래스에 대한 기본 생성자를 만들지 못하게 됩니다.) 생성된 특수 멤버 함수는 암시적으로 공개되고 인라인되며 문제의 함수가 파생된 클래스의 소멸자가 아닌 한 가상이 아닙니다. 가상 소멸자가 있는 기본 클래스에서 상속되는 클래스.

이 경우 파생 클래스에 대해 컴파일러 생성 소멸자도 가상입니다.

그러나 당신은 이미 이러한 것들을 알고 있습니다. 예, 예, 고대 역사: 메소포타미아, Shang 왕조, FORTRAN, C++98. 그러나 시대가 변했고 C++에서 특수 멤버 함수 생성에 대한 규칙도 변했습니다. 컴파일러가 클래스에 멤버 함수를 자동으로 삽입할 때를 아는 것만큼 효과적인 C++ 프로그래밍의 핵심은 거의 없기 때문에 새 규칙을 인식하는 것이 중요합니다.

C++11부터 특별 멤버 함수 클럽에는 이동 생성자와 이동 할당 연산자라는 두 명의 피험자가 더 있습니다. 그들의 서명은 다음과 같습니다.

클래스 위젯 { 공개:

```

...
위젯(위젯&& rhs);           // 생성자 이동
위젯& 연산자=(위젯&& rhs); // 할당 연산자 이동
...
};
```

그들의 세대와 행동을 지배하는 규칙은 모방하는 형제에 대한 규칙과 유사합니다. 이동 작업은 필요할 때만 생성되며 생성된 경우 클래스의 비정적 데이터 멤버에 대해 "멤버 단위 이동"을 수행합니다. 즉, 이동 생성자는 해당 매개변수 rhs의 해당 구성원에서 클래스의 각 비정적 데이터 구성원을 이동 구성하고 이동 할당 연산자는 해당 매개변수에서 각 비정적 데이터 구성원을 이동 할당합니다. 이동 생성자는 기본 클래스 부분(있는 경우)도 이동 구성하고 이동 할당 연산자는 기본 클래스 부분을 이동 할당합니다.

이제 데이터 멤버 또는 기본 클래스를 구성하거나 할당하는 이동 작업을 참조할 때 실제로 이동이 일어난다는 보장은 없습니다.

"멤버별 이동"은 실제로 멤버별 이동 요청과 더 비슷합니다. 왜냐하면 이동이 활성화되지 않은 유형(즉, 대부분의 C++98 레거시 클래스와 같이 이동 작업에 대한 특별한 지원을 제공하지 않는 유형)은 "이동됩니다."를 통해 복사 작업을 수행합니다. 각 멤버별 "이동"의 핵심은 이동할 개체에 `std::move`를 적용하는 것이며 결과는 함수 오버로드 해결 중에 이동 또는 복사를 수행할지 여부를 결정하는 데 사용됩니다. [항목 23](#)은 이 프로세스를 자세히 다룹니다.

이 항목의 경우 멤버별 이동은 이동 작업을 지원하는 데이터 멤버 및 기본 클래스에 대한 이동 작업으로 구성되지만 그렇지 않은 경우 복사 작업으로 구성된다는 점을 기억하십시오.

복사 작업의 경우와 마찬가지로 이동 작업은 직접 선언하면 생성되지 않습니다. 그러나 생성되는 정확한 조건은 복사 작업의 조건과 약간 다릅니다.

두 개의 복사 작업은 독립적입니다. 하나를 선언해도 컴파일러가 다른 하나를 생성하는 것을 막지는 않습니다. 따라서 복사 생성자를 선언했지만 복사 할당 연산자가 없는 경우 복사 할당이 필요한 코드를 작성하면 컴파일러가 복사 할당 연산자를 생성합니다. 마찬가지로 복사 할당 연산자를 선언했지만 복사 생성자는 없지만 코드에 복사 생성이 필요한 경우 컴파일러가 복사 생성자를 생성합니다. 그것은 C++98에서 사실이었고 C++11에서도 여전히 사실입니다.

두 이동 작업은 독립적이지 않습니다. 둘 중 하나를 선언하면 컴파일러에서 다른 하나를 생성하지 못하게 됩니다. 그 근거는 클래스의 이동 생성자를 선언하는 경우 컴파일러가 생성하는 기본 멤버별 이동과 다른 이동 생성을 구현해야 하는 방법에 대해 표시하는 것입니다. 그리고 멤버별 이동 구성에 문제가 있는 경우 멤버별 이동 할당에도 문제가 있을 수 있습니다. 따라서 이동 생성자를 선언하면 이동 할당 연산자가 생성되지 않고 이동 할당 연산자를 선언하면 컴파일러에서 이동 생성자가 생성되지 않습니다.

또한 복사 작업을 명시적으로 선언하는 클래스에 대해서는 이동 작업이 생성되지 않습니다. 정당화는 복사 작업을 선언하는 것입니다(`con-`

`struct` 또는 `할당`)은 객체 복사에 대한 일반적인 접근 방식(멤버 단위 복사)이 클래스에 적합하지 않음을 나타내며 컴파일러는 멤버 단위 복사가 복사 작업에 적합하지 않은 경우 멤버 단위 이동이 적절하지 않을 수 있다고 판단합니다. 이동 작업을 위해.

이것은 다른 방향으로도 진행됩니다. 클래스에서 이동 작업(생성 또는 할당)을 선언하면 컴파일러에서 복사 작업을 비활성화합니다. (복사 작업은 삭제하면 비활성화됩니다. [항목 11](#) 참조). 결국 멤버별 이동이 개체를 이동하는 적절한 방법이 아닌 경우 멤버별 복사가 개체를 복사하는 적절한 방법이라고 기대할 이유가 없습니다. 복사 작업이 활성화되는 조건이 C++98보다 C++11에서 더 제한적이기 때문에 이것은 C++98 코드를 깨뜨릴 수 있는 것처럼 들릴 수 있지만, 그렇지 않습니다. C++98 코드에는 이동 작업이 있을 수 없습니다. C++98에는 개체를 "이동"하는 것과 같은 기능이 없었기 때문입니다. 레거시 클래스가 사용자 선언 이동 작업을 가질 수 있는 유일한 방법은 C++11용으로 추가되고 이동 의미 체계를 이용하도록 수정된 클래스가 특수 멤버 함수 생성을 위한 C++11 규칙에 따라 재생되어야 하는 경우입니다. .

아마도 당신은 3의 법칙으로 알려진 지침에 대해 들어본 적이 있을 것입니다. 3법칙은 복사 생성자, 복사 할당 연산자 또는 소멸자를 선언할 경우 이 세 가지를 모두 선언해야 한다는 것입니다. 복사 작업의 의미를 인수해야 할 필요성은 거의 항상 일종의 리소스 관리를 수행하는 클래스에서 비롯되었으며 거의 항상 (1) 한 번의 복사 작업에서 리소스 관리가 수행되고 있음을 암시한다는 관찰에서 비롯되었습니다. 아마도 다른 복사 작업에서 수행해야 할 필요가 있고 (2) 클래스 소멸자는 리소스 관리(일반적으로 해제)에도 참여할 것입니다. 관리해야 할 기준 리소스는 메모리였으며, 이것이 메모리를 관리하는 모든 표준 라이브러리 클래스(예: 동적 메모리 관리를 수행하는 STL 컨테이너)가 모두 "빅 3"인 복사 작업과 소멸자를 선언하는 이유입니다.

3법칙의 결과는 사용자가 선언한 소멸자의 존재는 단순한 멤버 단위 복사가 클래스의 복사 작업에 적합하지 않을 수 있음을 나타냅니다. 이는 차례로 클래스가 소멸자를 선언하면 복사 작업이 올바른 작업을 수행하지 않기 때문에 자동으로 생성되지 않아야 함을 의미합니다. C++98이 채택되었을 때 이 추론 라인의 중요성이 완전히 인식되지 않았기 때문에 C++98에서는 사용자 선언 소멸자의 존재가 컴파일러의 복사 작업 생성 의지에 영향을 미치지 않았습니다. 이는 C++11에서도 계속 발생하지만 복사 작업이 생성되는 조건을 제한하면 너무 많은 레거시 코드가 손상될 수 있기 때문입니다.

그러나 3법칙 뒤에 있는 추론은 여전히 유효하며 복사 작업 선언이 이동 작업의 암시적 생성을 배제한다는 관찰과 결합되어 C++11이 이동 작업을 생성하지 않는다는 사실에 동기를 부여합니다. 사용자가 선언한 소멸자가 있는 클래스.

따라서 이 세 가지가 있는 경우에만 클래스에 대해 이동 작업이 생성됩니다(필요한 경우).
사실입니다:

- 클래스에 선언된 복사 작업이 없습니다. • 클래스에 선언된 이 동 작업이 없습니다.
- 클래스에서 소멸자가 선언되지 않았습니다.

C++11은 복사 작업 또는 소멸자를 선언하는 클래스에 대한 복사 작업의 자동 생성을 더 이상 사용하지 않기 때문에 어느 시점에서 유사한 규칙이 복사 작업으로 확장될 수 있습니다. 즉, 소멸자를 선언하는 클래스의 복사 작업 생성 또는 복사 작업 중 하나에 의존하는 코드가 있는 경우 종속성을 제거하기 위해 이러한 클래스를 업그레이드하는 것을 고려해야 합니다.

컴파일러 생성 함수의 동작이 정확하다면(즉, 클래스의 비정적 데이터 멤버를 멤버 단위로 복사하는 것이 원하는 경우) 작업이 쉽습니다. 명시적으로:

클래스 위젯 { 공개:

```
...
~위젯(); // 사용자 선언 dtor

...
위젯(const 위젯&) = 기본값; // 기본 복사 ctor // 동작은 정상
입니다.

Widget& // 기본 복사 할당 operator=(const Widget&) = default; // 동작은 정상입니다

...
};
```

이 접근 방식은 다형성 기본 클래스, 즉 파생된 클래스 개체가 조작되는 인터페이스를 정의하는 클래스에서 종종 유용합니다. 다형성 기본 클래스에는 일반적으로 가상 소멸자가 있습니다. 가상 소멸자가 없는 경우 일부 작업(예: 기본 클래스 포인터 또는 참조를 통해 파생 클래스 개체에 대한 삭제 또는 typeid 사용)이 정의되지 않거나 잘못된 결과를 생성하기 때문입니다. 클래스가 이미 가상인 소멸자를 상속하지 않는 한 소멸자를 가상으로 만드는 유일한 방법은 명시적으로 그렇게 선언하는 것입니다. 종종 기본 구현이 정확하고 "= default"가 이를 표현하는 좋은 방법입니다. 그러나 사용자가 선언한 소멸자는 이동 작업의 생성을 억제하므로 이동성이 지원되어야 하는 경우 "= default"는 종종 두 번째 응용 프로그램을 찾습니다. 이동 작업을 선언하면 복사 작업이 비활성화되므로 복사 가능성도 필요한 경우 "= default" 라운드가 한 번 더 수행됩니다.

```
class Base { 공
개: 가상 ~Base()
= 기본값; // dtor를 가상으로 만듭니다.
```

```

Base(Base&&) = 기본값;           // 이동 지원
Base& 연산자=(Base&&) = 기본값;

Base(const Base&) = 기본값;       // 복사 지원
Base& 연산자=(const Base&) = 기본값;

...
};


```

실제로 컴파일러가 복사 및 이동 작업을 기꺼이 생성하고 생성된 함수가 원하는 대로 작동하는 클래스가 있더라도 직접 선언하고 "= default"를 사용하는 정책을 채택할 수 있습니다. 정의. 더 많은 작업이 필요하지만 의도가 더 명확해지며 상당히 미묘한 버그를 피할 수 있습니다. 예를 들어, 문자열 테이블을 나타내는 클래스가 있다고 가정합니다. 즉, 정수 ID를 통해 문자열 값을 빠르게 조회할 수 있는 데이터 구조가 있습니다.

클래스 StringTable { 공개:

```

문자열 테이블() {}
...
// 삽입, 삭제, 조회 등을 위한 함수, // 복사/이동/dtor 가능 없음

```

개인:

```
std::map<int, std::string> 값; };
```

클래스가 복사 작업, 이동 작업 및 소멸자를 선언하지 않는다고 가정하면 컴파일러는 사용되는 경우 이러한 함수를 자동으로 생성합니다.
매우 편리합니다.

그러나 언젠가는 기본 구성과 그러한 객체의 파괴를 기록하는 것이 유용할 것이라고 결정했다고 가정해 봅시다. 해당 기능을 추가하는 것은 쉽습니다.

클래스 StringTable { 공개:

```

StringTable()
{ makeLogEntry("StringTable 개체 생성"); }           // 추가됨

~StringTable()
{ makeLogEntry("StringTable 개체 파괴"); } // 추가됨
...
// 이전과 같은 다른 함수

```

개인:

```
std::map<int, std::string> 값; }; // 이전과
```

이것은 합리적으로 보이지만 소멸자를 선언하면 잠재적으로 심각한 부작용이 있습니다. 이동 작업이 생성되는 것을 방지합니다. 그러나 클래스의 복사 작업 생성은 영향을 받지 않습니다. 따라서 코드는 기능 테스트를 컴파일, 실행 및 통과할 가능성이 높습니다. 여기에는 이동 기능 테스트가 포함됩니다. 이 클래스는 더 이상 이동이 가능하지 않더라도 이동 요청이 컴파일되고 실행되기 때문입니다. 이러한 요청은 이 항목의 앞부분에 언급된 대로 사본을 생성하게 합니다.

이는 StringTable 객체를 "이동하는" 코드가 실제로 객체를 복사한다는 것을 의미합니다. 즉, 기본 `std::map<int, std::string>` 객체를 복사합니다. 그리고 `std::map<int, std::string>` 을 복사하는 것은 이동하는 것보다 훨씬 더 느릴 것입니다.

클래스에 소멸자를 추가하는 간단한 작업으로 인해 심각한 성능 문제가 발생할 수 있습니다! 복사 및 이동 작업이 `"= default"`를 사용하여 명시적으로 정의되어 있었다면 문제가 발생하지 않았을 것입니다.

이제 C++11에서 복사 및 이동 작업을 제어하는 규칙에 대해 끝없이 불평하는 것을 참았으니, 내가 언제 다른 두 가지 특수 멤버 함수인 기본 생성자와 소멸자에 주의를 기울일지 궁금할 것입니다. 그 시간은 지금이지만 이 문장에만 해당됩니다. 이 멤버 함수에 대해 거의 아무것도 변경되지 않았기 때문입니다. C++11의 규칙은 C++98과 거의 동일합니다.

따라서 특수 멤버 함수를 관리하는 C++11 규칙은 다음과 같습니다.

- **기본 생성자:** C++98과 동일한 규칙입니다. 클래스에 다음이 포함된 경우에만 생성됨 사용자 선언 생성자가 없습니다.
- **소멸자:** 기본적으로 C++98과 동일한 규칙입니다. 유일한 차이점은 소멸자가 기본적으로 noexcept라는 것입니다(항목 14 참조). C++98에서와 같이 기본 클래스 소멸자가 가상인 경우에만 가상입니다.
- **복사 생성자:** C++98과 동일한 런타임 동작: 비정적 데이터 멤버의 멤버별 복사 생성. 클래스에 사용자 선언 복사 생성자가 없는 경우에만 생성됩니다. 클래스가 이동 작업을 선언하면 삭제됩니다.

사용자 선언 복사 할당 연산자 또는 소멸자가 있는 클래스에서 이 함수를 생성하는 것은 더 이상 사용되지 않습니다.

- **복사 할당 연산자:** C++98과 동일한 런타임 동작: 비정적 데이터 멤버의 멤버별 복사 할당. 클래스에 사용자 선언 복사 할당 연산자가 없는 경우에만 생성됩니다. 클래스가 이동 작업을 선언하면 삭제됩니다. 사용자 선언 복사 생성자 또는 소멸자가 있는 클래스에서 이 함수를 생성하는 것은 더 이상 사용되지 않습니다.

- 이동 생성자 및 이동 할당 연산자: 각각은 비정적 데이터 멤버의 멤버 단위 이동을 수행합니다. 클래스에 사용자 선언 복사 작업, 이동 작업 또는 소멸자가 없는 경우에만 생성됩니다.

컴파일러가 특별한 멤버 함수를 생성하는 것을 방지하는 멤버 함수 템플릿의 존재에 대한 규칙에는 아무 것도 없다는 점에 유의하십시오. 즉, 위젯이 다음과 같이 표시되면

```
클래스 위젯 {
    ...
    템플릿<유형이름 T>           // 위젯 구성 // 무엇이든
    위젯(const T& rhs);
    ...
    템플릿<유형이름 T>           // 위젯 할당 // 무엇이든
    위젯& 연산자=(const T& rhs);
    ...
};
```

컴파일러는 위젯에 대한 복사 및 이동 작업을 생성할 것입니다(생성을 제어하는 일반적인 조건이 충족된다고 가정). 이러한 템플릿이 복사 생성자와 복사 할당 연산자에 대한 서명을 생성하기 위해 인스턴스화될 수 있음에도 불구하고. (T가 Widget인 경우도 마찬가지입니다.) 모든 가능성에서 이것은 거의 인정할 가치가 없는 극단적인 경우처럼 보일 것입니다. 하지만 제가 언급하는 데는 이유가 있습니다. [항목 26](#)은 이것이 중요한 결과를 가질 수 있음을 보여줍니다.

기억할 사항 • 특수 멤버

함수는 컴파일러가 자체적으로 생성할 수 있는 기본 생성자, 소멸자, 복사 작업 및 이동 작업입니다. • 이동 작업은 명시적으로 선언된 이동 작업, 복사 작업 및 소멸자가 없는 클래스에 대해서만 생성됩니다. • 복사 생성자는 명시적으로 선언된 복사 생성자가 없는 클래스에 대해서만 생성되며 이동 작업이 선언되면 삭제됩니다.

복사 할당 연산자는 명시적으로 선언된 복사 할당 연산자가 없는 클래스에 대해서만 생성되며 이동 연산이 선언되면 삭제됩니다. 명시적으로 선언된 소멸자가 있는 클래스의 복사 작업 생성은 더 이상 사용되지 않습니다.

- 멤버 함수 템플릿은 특수 멤버 생성을 억제하지 않습니다.
기능.

4장

스마트 포인터

시인과 작곡가에게는 사랑이 있습니다. 그리고 때때로 계산에 대해. 때때로 둘 다. 엘리자베스 배렛 브라우닝(Elizabeth Barrett Browning) ("내가 당신을 어떻게 사랑하나요? 제가 방법을 세어보겠습니다.")과 폴 사이먼("당신의 연인을 떠나는 50가지 방법이 있을 것입니다.")은 사랑에 대한 다소 다른 견해와 세는 것에 영감을 받아 우리는 아마도 원시 포인터가 사랑하기 어려운 이유를 열거하십시오:

1. 그 선언은 그것이 단일 객체를 가리키는지 아니면 배열을 가리키는지를 나타내지 않습니다.
2. 그것의 선언은 사용을 마쳤을 때 포인터가 가리키는 것을 파괴해야 하는지 여부에 대해 아무 것도 드러내지 않습니다.
3. 포인터가 가리키는 것을 파괴해야 한다고 결정했다면 방법을 알 방법이 없습니다. 삭제를 사용해야 합니까, 아니면 다른 파괴 메커니즘이 있습니까(예: 포인터가 전달되어야 하는 전용 파괴 함수)?
4. 삭제가 올바른 방법이라는 것을 알게 된 경우 이유 1은 단일 개체 형식("delete")을 사용할지 아니면 배열 형식("delete []")을 사용할지 여부를 알 수 없음을 의미합니다. . 잘못된 형식을 사용하면 결과가 정의되지 않습니다.
5. 포인터가 가리키는 것을 소유하고 있음을 확인하고 포인터를 제거하는 방법을 발견했다고 가정하면 코드의 모든 경로(예외로 인한 경로 포함)를 따라 정확히 한 번만 제거를 수행하는 것을 보장하기 어렵습니다. 경로가 누락되면 리소스 누수가 발생하고 두 번 이상 파괴를 수행하면 정의되지 않은 동작이 발생합니다.
6. 일반적으로 포인터가 매달려 있는지 여부, 즉 포인터가 가리킬 개체를 더 이상 보유하지 않는 메모리를 가리키는지 여부를 알 수 있는 방법이 없습니다. 땅글링 포인터는 포인터가 여전히 객체를 가리키고 있는 동안 객체가 파괴될 때 발생합니다.

원시 포인터는 확실히 강력한 도구이지만 수십 년의 경험을 통해 집중이나 훈련에서 약간의 실수만 있으면 이러한 도구가 표면상의 주인을 펼 수 있음을 보여주었습니다.

스마트 포인터는 이러한 문제를 해결하는 한 가지 방법입니다. 스마트 포인터는 원시 포인터를 감싸는 래퍼로, 원시 포인터처럼 작동하지만 많은 함정을 피할 수 있습니다. 따라서 원시 포인터보다 스마트 포인터를 선호해야 합니다.

스마트 포인터는 원시 포인터가 할 수 있는 거의 모든 작업을 수행할 수 있지만 오류 가능성은 훨씬 적습니다.

C++11에는 4개의 스마트 포인터가 있습니다: `std::auto_ptr`, `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`. 모두 동적으로 할당된 객체의 수명을 관리하는 데 도움이 되도록 설계되었습니다. 즉, 해당 객체가 적절한 시간(예외 이벤트 포함)에 적절한 방식으로 파괴되도록 하여 리소스 누출을 방지합니다.

`std::auto_ptr`은 C++98에서 더 이상 사용되지 않는 잔여물입니다. 나중에 C++11의 `std::unique_ptr`이 된 것을 표준화하려는 시도였습니다. 작업을 올바르게 수행하려면 이동 의미 체계가 필요했지만 C++98에는 그런 의미가 없었습니다. 해결 방법으로 `std::auto_ptr`은 이동을 위해 복사 작업을 선택했습니다. 이로 인해 놀라운 코드(`std::auto_ptr`를 복사하면 `null`이 설정됩니다!)와 실망스러운 사용 제한(예: 컨테이너에 `std::auto_ptr`을 저장할 수 없음)이 발생했습니다.

`std::unique_ptr`은 `std::auto_ptr`이 하는 모든 일과 그 이상을 수행합니다. 효율적으로 수행하며 개체를 복사하는 것이 의미하는 바를 왜곡하지 않고 수행합니다. 모든 면에서 `std::auto_ptr`보다 낫습니다. `std::auto_ptr`의 유일한 합법적인 사용 사례는 C++98 컴파일러로 코드를 컴파일해야 하는 경우입니다. 해당 제약 조건이 없으면 `std::auto_ptr`을 `std::unique_ptr`로 바꾸고 절대 뒤돌아보지 않아야 합니다.

스마트 포인터 API는 매우 다양합니다. 모두에게 공통적인 유일한 기능은 기본 구성입니다. 이러한 API에 대한 포괄적인 참조를 널리 사용할 수 있기 때문에 API 개요에서 자주 누락되는 정보(예: 주 목할만한 사용 사례, 런타임 비용 분석 등)에 대해 논의하는 데 집중할 것입니다. 이러한 정보를 마스터하는 것은 이러한 스마트 포인터를 사용하는 것과의 차이가 될 수 있습니다. 효과적으로 사용합니다.

항목 18: 독점 소유권 리소스 관리를 위해 `std::unique_ptr`을 사용하십시오.

스마트 포인터에 도달하면 일반적으로 `std::unique_ptr`이 가장 가까운 위치에 있어야 합니다. 기본적으로 `std::unique_ptrs`는 원시 포인터와 크기가 같고 대부분의 작업(역참조 포함)에서 정확히 동일한 명령을 실행한다고 가정하는 것이 합리적입니다. 즉, 상황에서도 사용할 수 있습니다.

메모리와 주기가 빽빽한 곳. 원시 포인터가 충분히 작고 빠르면 std::unique_ptr도 거의 확실합니다.

std::unique_ptr은 독점적 소유권 의미를 구현합니다. null이 아닌 std::unique_ptr은 항상 그 것이 가리키는 것을 소유합니다. std::unique_ptr을 이동하면 소유권이 소스 포인터에서 대상 포인터로 이전됩니다. (소스 포인터는 null로 설정됩니다.) std::unique_ptr을 복사하는 것은 허용되지 않습니다. 왜냐하면 std::unique_ptr을 복사할 수 있다면 결국 두 개의 std::unique_ptr이 동일한 리소스에 있게 될 것이기 때문입니다. 그것은 그 자원을 소유했고 따라서 파괴해야 합니다. 따라서 std::unique_ptr은 이동 전용 유형입니다. 파괴 시 null이 아닌 std::unique_ptr은 리소스를 파괴합니다. 기본적으로 리소스 파괴는 std::unique_ptr 내부의 원시 포인터에 삭제를 적용하여 수행됩니다.

std::unique_ptr의 일반적인 용도는 계층 구조의 개체에 대한 팩토리 함수 반환 유형입니다. 기본 클래스가 투자인 투자 유형(예: 주식, 채권, 부동산 등)에 대한 계층 구조가 있다고 가정합니다.

클래스 투자 { ... };

클래스 스톡:

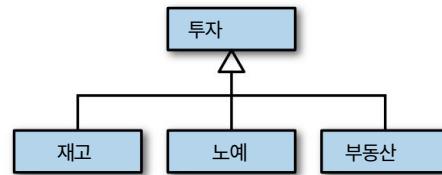
 공공 투자 { ... };

종류 채권: 공공

 투자 { ... };

클래스 부동산: 공공 투자

 { ... };



이러한 계층 구조에 대한 팩토리 함수는 일반적으로 힘에 개체를 할당하고 해당 개체에 대한 포인터를 반환합니다. 호출자는 개체가 더 이상 필요하지 않을 때 개체를 삭제할 책임이 있습니다. 이는 std::unique_ptr과 완벽하게 일치합니다. 호출자가 공장에서 반환한 리소스(즉, 리소스의 독점적 소유권)에 대한 책임을 획득하고 std::unique_ptr이 std::unique_ptr이 다음과 같은 경우 가리키는 것을 자동으로 삭제하기 때문입니다. 파괴됨. 투자 계층에 대한 팩토리 함수는 다음과 같이 선언할 수 있습니다.

```
template<typename... Ts>
std::unique_ptr<Investment>
makeInvestment(Ts&... params);
```

// std::unique_ptr을 // 주어진 인수에
// 생성된 // 객체로 반환

호출자는 다음과 같이 단일 범위에서 반환된 std::unique_ptr을 사용할 수 있습니다.

```
{
  ...
}
```

```
plInvestment = makeInvestment( arguments ); // Investment 유형을 투자
```

...

```
}
```

// 파괴 *plInvestment

그러나 팩토리에서 반환된 std::unique_ptr이 컨테이너로 이동되고 컨테이너 요소가 이후에 개체의 데이터 멤버로 이동되고 해당 개체가 나중에 소멸되는 경우와 같은 소유권 마이그레이션 시나리오에서도 사용할 수 있습니다. 그런 일이 발생하면 객체의 std::unique_ptr 데이터 멤버도 파괴되고, 그 파괴로 인해 팩토리에서 반환된 리소스가 파괴됩니다. 소유권 체인이 예외 또는 기타 비정상적인 제어 흐름(예: 초기 함수 반환 또는 루프에서 중단)으로 인해 중단된 경우 관리 리소스를 소유하는 std::unique_ptr은 결국 소멸자를 호출하고 리소스 그것으로 인해 파괴될 것입니다.

기본적으로 그 파괴는 삭제를 통해 발생하지만 생성하는 동안 std::unique_ptr 객체는 사용자 지정 삭제자를 사용하도록 구성할 수 있습니다. 그들의 자원이 파괴되는 시간. makeInvestment에 의해 생성된 객체를 직접 삭제하지 않고 대신 먼저 로그 항목을 작성해야 하는 경우 makeInvestment를 다음과 같이 구현할 수 있습니다. (설명은 코드를 따르므로 동기가 명확하지 않은 것을 보더라도 걱정하지 마십시오.)

```
auto dellInvmt = [](투자* plInvestment) // 사용자 정
                                            // 의 // 삭제자 //
{ 매크로그엔트리(p투자); plInvestment      (람다식 // 표현식)
    삭제; };

template<typename... Ts> // 수정
std::unique_ptr<Investment, decltype(dellInvmt)> // 반환 유형 makeInvestment(Ts&&...
params) { std::unique_ptr<Investment, decltype(dellInvmt)> // ptr to be pInv(nullptr,
dellInvmt); // 반환
```

¹ 이 규칙에는 몇 가지 예외가 있습니다. 대부분 비정상적인 프로그램 종료로 인해 발생합니다. 예외가 스레드의 기본 기능(예: 프로그램의 초기 스레드에 대한 main) 밖으로 전파되거나 noexcept 사양이 위반되면(항목 14 참조), 로컬 객체는 소멸되지 않을 수 있으며 std::종단 또는 종료 함수(예: std::_Exit, std::exit 또는 std::quick_exit)가 호출되면 확실히 호출되지 않습니다.

```

if ( /* Stock 객체가 생성되어야 함 */ { pInv.reset(new
Stock(std::forward<Ts>(params)...)); } else if ( /* Bond 객체가 생성되
어야 함 */ { pInv.reset(new Bond(std::forward<Ts>(params)...)); } else if ( /*
RealEstate 객체를 생성해야 함 */ { pInv.reset(new
RealEstate(std::forward<Ts>(params)...)); }

반환 pInv;
}

```

잠시 후 이것이 어떻게 작동하는지 설명하겠습니다. 하지만 먼저 전화를 건 사람인 경우 상황이 어떻게 보이는지 고려하십시오. `makeInvestment` 호출의 결과를 자동 변수에 저장한다고 가정하면 사용 중인 리소스를 삭제하는 동안 특별한 처리가 필요하다는 사실에 대해 무지한 나머지 기뻐할 것입니다. 사실, `std::unique_ptr`을 사용하면 리소스가 파괴되어야 하는 시기에 대해 걱정할 필요가 없고 프로그램의 모든 경로에서 정확히 한 번만 파괴가 발생하는지 확인할 필요가 없기 때문에 정말로 행복합니다. `std::unique_ptr`은 이러한 모든 것을 자동으로 처리합니다. 클라이언트의 관점에서 `makeInvestment`의 인터페이스는 다음과 같습니다.

달콤한.

다음을 이해하면 구현도 꽤 좋습니다.

- `dellnvmt`은 `makeInvestment`에서 반환된 개체에 대한 사용자 지정 삭제 프로그램입니다. 모든 사용자 정의 삭제 함수는 파괴할 개체에 대한 원시 포인터를 수락한 다음 해당 개체를 파괴하는 데 필요한 작업을 수행합니다. 이 경우 조치는 `makeLogEntry`를 호출한 다음 삭제를 적용하는 것입니다. `dellnvmt`을 생성하기 위해 람다 식을 사용하는 것은 편리하지만, 곧 보게 되겠지만, 기존의 함수를 작성하는 것보다 더 효율적입니다.
- 사용자 정의 삭제자를 사용하려면 해당 유형을 `std::unique_ptr`에 대한 두 번째 유형 인수로 지정해야 합니다. 이 경우 그것이 `dellnvmt`의 유형이고 이것이 `makeInvestment`의 반환 유형이 `std::unique_ptr<Investment, decltype(dellnvmt)>`인 이유입니다. (`decltype`에 대한 정보는 [항목 3](#)을 참조하십시오.)
- `makeInvestment`의 기본 전략은 null `std::unique_ptr`을 생성하고 적절한 유형의 개체를 가리키도록 만든 다음 반환하는 것입니다. 사용자 지정 삭제자 `dellnvmt`을 `pInv`와 연결하기 위해 두 번째 생성자 인수로 전달합니다.

- 원시 포인터(예: new에서)를 std::unique_ptr에 할당하려고 시도하면 원시에서 암시적으로 변환을 구성하기 때문에 컴파일 스마트 포인터. 이러한 암시적 변환은 문제가 될 수 있으므로 C++11은 포인터는 그들을 금지합니다. 이것이 pInv가 소유자를 가정하도록 재설정하는 데 사용되는 이유입니다. new를 통해 생성된 객체의 배송.
- new를 사용할 때마다 std::forward를 사용하여 인수를 완벽하게 전달합니다. makeInvestment로 전달됩니다(항목 25 참조). 이것은 모든 정보를 프로- 생성되는 객체의 생성자가 사용할 수 있는 호출자에 의해 제공됩니다.
- 사용자 정의 삭제자는 투자* 유형의 매개변수를 사용합니다. 예 대해 상관없이 makeInvestment 내에서 생성된 실제 유형의 객체(예: 주식, 채권 또는 Real Estate), Invest로 람다 식 내부에서 궁극적으로 삭제됩니다. 멘션* 객체. 이것은 기본 클래스를 통해 파생 클래스 객체를 삭제할 것임을 의미합니다. 바늘. 이것이 작동하려면 기본 클래스인 투자에 가상 소멸자가 있어야 합니다.

```
클래스 투자 {
    공공의:
        ...
        가상 ~투자();
        ...
    };
};// 필수적인
// 설계
// 요소!
```

C++14에서 함수 반환 유형 추론의 존재(항목 3 참조)는 다음을 의미합니다. makeInvestment는 이 더 간단하고 캡슐화된 방식으로 구현될 수 있습니다. 이온:

```
템플릿<유형 이름... Ts>
auto makeInvestment(Ts&&... params) { // C++14

    auto dellnvmt = [](투자* pInvestment) // 이것은 지금
    { // 내부
        makeLogEntry(p투자); // 만들다-
        pInvestment 삭제; // 투자
    };

    std::unique_ptr<Investment, decltype(dellnvmt)> //
        pInv(nullptr, dellnvmt); // 전에

    만약에 ( ... ) // 이전과
    {
        pInv.reset(new Stock(std::forward<Ts>(params)...));
    }
    그렇지 않으면 ( ... ) // 이전과
```

```

{
    pInv.reset(new Bond(std::forward<Ts>(params)...));
}
그렇지 않으면 ( … )                                // 이전과
{
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
}
반환 pInv;                                         // 이전과
}

```

나는 앞에서 기본 삭제자(즉, 삭제)를 사용할 때 추론할 수 있다고 언급했습니다.
 적절하게 std::unique_ptr 객체가 원시 포인터와 같은 크기라고 가정합니다. 언제
 사용자 지정 삭제자가 그림을 입력하면 더 이상 그렇지 않을 수 있습니다. 삭제자는
 함수 포인터는 일반적으로 std::unique_ptr의 크기를 1에서 증가시킵니다.
 둘에게 한마디. 함수 객체인 삭제자의 경우 크기 변경은 다음에 따라 달라집니다.
 함수 객체에 얼마나 많은 상태가 저장되어 있는지. 상태 비저장 함수 개체(예:
 캡처가 없는 람다 식)에는 크기 페널티가 발생하지 않으며 이는 다음을 의미합니다.
 사용자 지정 삭제 프로그램이 기능 또는 캡처리스로 구현될 수 있는 경우
 람다 식에서 람다가 선호됩니다.

```

auto delInvmt1 = [](투자* p투자)                    // 커스텀
                                            // 삭제자
{ 메이크로그엔트리(p투자); pInvestment 삭제; }      // 처럼
                                                        // 상태 비저장
                                                        // 람다
}

```

template<typename... Ts> // 반환 유형
 std::unique_ptr<Investment, decltype(delInvmt1)> // 크기는 다음과 같습니다.
 makeInvestment(Ts&&... 인수); // 투자*

```

무효 delInvmt2(투자* pInvestment)                  // 커스텀
{ makeLogEntry(pInvestment); pInvestment 삭제; }    // 삭제자
                                                        // 함수로
}

```

```

template<typename... Ts> // 반환 유형은
std::unique_ptr<Investment, // 투자 규모*
void (*)> // 더하기 최소 크기
makeInvestment(Ts&&... 매개변수); // 함수 포인터!

```

광범위한 상태의 함수 객체 삭제자는 다음의 std::unique_ptr 객체를 생성할 수 있습니다.
 상당한 크기. 사용자 지정 삭제 프로그램이 std::unique_ptrs를 만드는 것을 발견하면
 허용할 수 없을 정도로 크면 디자인을 변경해야 할 수 있습니다.

팩토리 함수는 `std::unique_ptr`의 유일한 일반적인 사용 사례가 아닙니다. Pimpl Idiom을 구현하기 위한 메커니즘으로 훨씬 더 유명합니다. 이에 대한 코드는 복잡하지 않지만 경우에 따라 간단하지 않으므로 해당 항목에 대한 전용 항목인 [22번 항목](#)을 참조하겠습니다.

`std::unique_ptr`은 개별 개체(`std::unique_ptr<T>`)와 배열(`std::unique_ptr<T[]>`)의 두 가지 형식으로 제공됩니다. 결과적으로 `std::unique_ptr`이 가리키는 엔티티의 종류에 대한 모호함이 없습니다. `std::unique_ptr` API는 사용 중인 양식과 일치하도록 설계되었습니다. 예를 들어, 단일 개체 형식에는 인덱싱 연산자(`operator[]`)가 없는 반면 배열 형식에는 역참조 연산자(`operator*` 및 `operator->`)가 없습니다.

`std::array`, `std::vector` 및 `std::string`이 원시 배열보다 거의 항상 더 나은 데이터 구조 선택이기 때문에 배열에 대한 `std::unique_ptr`의 존재는 지적 관심일 뿐입니다. 내가 생각할 수 있는 유일한 상황은 `std::unique_ptr<T[]>`이 의미가 있을 때 소유권을 가정하는 힙 배열에 대한 원시 포인터를 반환하는 C와 같은 API를 사용할 때입니다.

`std::unique_ptr`은 독점 소유권을 표현하는 C++11 방법이지만 가장 매력적인 기능 중 하나는 쉽고 효율적으로 `std::shared_ptr`로 변환한다는 것입니다.

```
std::shared_ptr<투자> sp = makeInvestment( 인 // std::unique_ptr을 // std::shared_ptr  
수 ); //로 변환
```

이것이 `std::unique_ptr` 팩토리 함수 반환 유형으로 매우 적합한 이유의 핵심 부분입니다. 팩토리 함수는 호출자가 반환하는 객체에 대해 독점적 소유권 의미 체계를 사용하기를 원하는지 또는 공유 소유권(예: `std::shared_ptr`)이 더 적절한지 여부를 알 수 없습니다. `std::unique_ptr`을 반환함으로써 팩토리는 호출자에게 가장 효율적인 스마트 포인터를 제공하지만 호출자가 더 유연한 형제로 교체하는 것을 방해하지는 않습니다. (`std::shared_ptr`에 대한 정보는 [항목 19](#)로 이동하십시오.)

기억해야 할 사항

- `std::unique_ptr`은 관리를 위한 작고 빠른 이동 전용 스마트 포인터입니다.
독점 소유권 의미 체계가 있는 리소스.
- 기본적으로 리소스 삭제는 삭제를 통해 발생하지만 사용자 지정 삭제자를 지정할 수 있습니다.
상태 저장 삭제자와 삭제자로서의 함수 포인터는 `std::unique_ptr` 개체의 크기를 늘립니다.
- `std::unique_ptr`을 `std::shared_ptr`로 변환하는 것은 쉽습니다.

항목 19: 공유 소유권 리소스 관리를 위해 std::shared_ptr 을 사용하세요.

가비지 수집 기능이 있는 언어를 사용하는 프로그래머는 C++ 프로그래머가 리소스 누출을 방지하기 위해 수행하는 작업을 비웃습니다. “얼마나 원시적인가!” 그들은 조롱한다.

“1960년대에 Lisp에서 메모를 받지 않았습니까? 인간이 아니라 기계가 자원 수명을 관리해야 합니다.” C++ 개발자는 눈을 굴립니다. “리소스가 메모리뿐이고 리소스 회수 시점이 비결정적이라는 메모를 말씀하시는 겁니까? 우리는 소멸자의 일반성과 예측 가능성을 선호합니다. 감사합니다.” 그러나 우리의 허세는 부분적으로 허세입니다. 쓰레기 수거는 정말 편리하고 수동 수명 관리는 실제로 둘칼과 곰 가죽을 사용하여 니모닉 메모리 회로를 구성하는 것과 유사할 수 있습니다. 자동으로 작동하지만(가비지 수집과 같이) 모든 리소스에 적용되고 예측 가능한 타이밍(소멸자처럼)을 갖는 시스템이라는 두 가지 장점을 모두 가질 수 없는 이유는 무엇입니까?

std::shared_ptr은 이러한 세계를 함께 묶는 C++11 방법입니다. std::shared_ptrs를 통해 액세스되는 개체는 공유 소유권을 통해 해당 포인터에 의해 관리되는 수명이 있습니다. 특정 std::shared_ptr이 개체를 소유하지 않습니다. 대신, 그것을 가리키는 모든 std::shared_ptrs는 더 이상 필요하지 않은 지점에서 파괴를 보장하기 위해 협력합니다. 객체를 가리키는 마지막 std::shared_ptr이 그곳을 가리키는 것을 멈출 때(예를 들어, std::shared_ptr이 파괴되거나 다른 객체를 가리키도록 만들어졌기 때문에), 그 std::shared_ptr은 그것이 가리키는 객체를 파괴합니다. 가비지 수집과 마찬가지로 클라이언트는 가리키는 개체의 수명 관리에 관심을 가질 필요가 없지만 소멸자와 마찬가지로 개체의 소멸 타이밍은 결정적입니다.

std::shared_ptr은 얼마나 많은 std::shared_ptr이 가리키는지 추적하는 리소스와 관련된 값인 리소스의 참조 횟수를 참조하여 리소스를 가리키는 마지막 항목인지 여부를 알 수 있습니다. std::shared_ptr 생성자는 이 카운트를 증가시키고(보통—아래 참조), std::shared_ptr 소멸자는 이 카운트를 감소시키며, 복사 할당 연산자는 둘 다 수행합니다. (만약 sp1과 sp2가 서로 다른 개체에 대한 std::shared_ptrs인 경우 "sp1 = sp2;" 할당은 sp2가 가리키는 개체를 가리키도록 sp1을 수정합니다. 할당의 순 효과는 개체에 대한 참조 횟수입니다. 원래 sp1이 가리키는 객체는 감소하고 sp2가 가리키는 객체는 증가합니다.) 감소를 수행한 후 std::shared_ptr이 참조 카운트를 0으로 보는 경우 더 이상 std::shared_ptr이 리소스를 가리키지 않으므로 std::shared_ptr은 그 것을 파괴합니다.

참조 횟수의 존재는 성능에 영향을 미칩니다.

- std::shared_ptr는 내부적으로 리소스에 대한 원시 포인터와 리소스의 참조 카운트에 대한 원시 포인터를 포함하기 때문에 원시 포인터 크기의 두 배입니다.²
- 참조 횟수에 대한 메모리는 동적으로 할당되어야 합니다. 개념적으로 참조 카운트는 개체와 연결되지만 가리키는 개체는 이에 대해 아무것도 모릅니다. 따라서 참조 카운트를 저장할 장소가 없습니다. (내장 유형의 객체를 포함하여 모든 객체가 std::shared_ptr에 의해 관리될 수 있다는 즐거운 의미가 있습니다.) 항목 21은 std::shared_ptr이 std에 의해 생성될 때 동적 할당 비용을 피할 수 있다고 설명합니다. :make_shared 하지만 std::make_shared를 사용할 수 없는 상황이 있습니다. 어느 쪽이든 참조 카운트는 동적으로 할당된 데이터로 저장됩니다. • 참조 카운트의 증가 및 감소는 원자적이어야 합니다. 다른 스레드에 동시 판독기와 기록기가 있을 수 있기 때문입니다. 예를 들어, 한 스레드의 리소스를 가리키는 std::shared_ptr은 소멸자를 실행할 수 있지만(따라서 가리키는 리소스에 대한 참조 카운트 감소), 다른 스레드에서는 std::shared_ptr이 동일한 객체에 대해 복사될 수 있습니다(따라서 동일한 참조 카운트 증가). 원자적 연산은 일반적으로 비원자적 연산보다 느리므로 참조 카운트가 일반적으로 단어 크기에 불과하더라도 읽고 쓰는 데 비교적 비용이 많이 든다고 가정해야 합니다.

내가 std::shared_ptr 생성자가 가리키는 객체에 대한 참조 카운트를 "보통" 증가시킨다고 썼을 때 내가 당신의 호기심을 자극했습니까? 객체를 가리키는 std::shared_ptr을 생성하면 항상 해당 객체를 가리키는 std::shared_ptr이 하나 더 생성됩니다. 그렇다면 항상 참조 횟수를 증가시키면 안 되는 이유는 무엇입니까?

건설을 이동, 그래서입니다. 다른 std::shared_ptr에서 std::shared_ptr을 이동 구성하면 소스 std::shared_ptr이 null로 설정되며, 이는 새 std::shared_ptr이 시작되는 순간 이전 std::shared_ptr이 리소스를 가리키는 것을 중지한다는 의미입니다. 결과적으로 참조 카운트 조작이 필요하지 않습니다.

따라서 std::shared_ptrs를 이동하는 것이 복사하는 것보다 빠릅니다. 복사하려면 참조 카운트를 늘려야 하지만 이동은 그렇지 않습니다. 이것은 구성과 마찬가지로 할당에 대해서도 마찬가지이므로 이동 구성은 복사 구성보다 빠르며 이동 지정은 복사 지정보다 빠릅니다.

std::unique_ptr(항목 18 참조)과 마찬가지로 std::shared_ptr은 기본 리소스 파괴 메커니즘으로 삭제를 사용하지만 사용자 지정 삭제 프로그램도 지원합니다.

그러나 이 지원의 디자인은 std::unique_ptr의 디자인과 다릅니다. 을 위한

2 이 구현은 표준에서 필요하지 않지만 내가 알고 있는 모든 표준 라이브러리 구현 와 함께 사용합니다.

`std::unique_ptr`, 삭제자의 유형은 스마트 포인터 유형의 일부입니다. `std::shared_ptr`의 경우 다릅니다.

```
자동 로깅Del = [](위젯 * pw) {  
    makeLogEntry(pw); 비  
    일번호 삭제; };
```

// 커스텀 삭제자 // (항목
18에서와 같이)

`표준::고유_ptr<`
위젯, decltype(loggingDel) > upw(새 위
젯, loggingDel);

// 삭제 유형은 // ptr 유형의
일부입니다.

`std::shared_ptr<Widget>`
spw(새 위젯, loggingDel);

// 삭제자 유형은 // ptr 유형의 일부
가 아닙니다.

`std::shared_ptr` 디자인은 더 유연합니다. 두 개의 `std::shared_ptr <Widget>`을 고려하세요. 각각 다른 유형의 사용자 지정 삭제자가 있습니다(예: 사용자 지정 삭제자가 람다 식을 통해 지정되기 때문).

```
auto customDeleter1 = [](위젯 * pw) { … }; auto  
customDeleter2 = [](위젯 * pw) { … };
```

// 사용자 지정 삭제자, // 각
각 다른 유형을 가짐

`std::shared_ptr<위젯> pw1(새 위젯, customDeleter1); std::shared_ptr<위젯>
pw2(새 위젯, customDeleter2);`

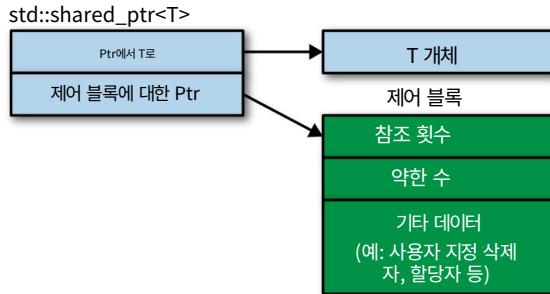
`pw1`과 `pw2`는 동일한 유형을 가지므로 해당 유형의 객체 컨테이너에 배치할 수 있습니다.

`std::vector<std::shared_ptr<위젯>> vpw{ pw1, pw2 };`

그것들은 또한 서로에게 할당될 수 있고, 각각은 `std::shared_ptr<Widget>` 유형의 매개변수를 취하는 함수에 전달될 수 있습니다. 사용자 지정 삭제기의 유형이 `std::unique_ptr`의 유형에 영향을 미치기 때문에 사용자 지정 삭제기의 유형이 다른 `std::unique_ptrs`로는 이러한 작업을 수행할 수 없습니다.

`std::unique_ptr`과의 또 다른 차이점은 사용자 지정 삭제자를 지정해도 `std::shared_ptr` 객체의 크기가 변경되지 않는다는 것입니다. 삭제자와 상관없이 `std::shared_ptr` 객체는 크기가 두 개의 포인터입니다. 그것은 좋은 소식이지만 당신을 막연하게 불안하게 만들 것입니다. 사용자 지정 삭제자는 함수 자체일 수 있으며 함수 자체에는 임의의 양의 데이터가 포함될 수 있습니다. 즉, 임의로 커질 수 있습니다. 더 이상 메모리를 사용하지 않고 `std::shared_ptr`가 임의의 크기의 삭제자를 어떻게 참조할 수 있습니까?

그것은 할 수 없습니다. 더 많은 메모리를 사용해야 할 수도 있습니다. 그러나 해당 메모리는 std::shared_ptr 객체의 일부가 아닙니다. 힙에 있거나 std::shared_ptr 작성자가 사용자 지정 할당자에 대한 std::shared_ptr 지원을 활용한 경우 할당자가 관리하는 메모리가 있는 곳입니다. 나는 앞서 std::shared_ptr 객체가 가리키는 객체에 대한 참조 카운트에 대한 포인터를 포함한다고 언급했습니다. 그것은 사실이지만 참조 카운트가 제어 블록으로 알려진 더 큰 데이터 구조의 일부이기 때문에 약간 오해의 소지가 있습니다. std::shared_ptrs에서 관리하는 각 객체에 대한 제어 블록이 있습니다. 제어 블록에는 참조 횟수 외에 사용자 지정 삭제기의 복사본이 포함되어 있습니다(지정된 경우). 사용자 지정 할당자가 지정된 경우 제어 블록에도 해당 할당자의 복사본이 포함됩니다. 제어 블록은 또한 항목 21에서 설명하는 것처럼 약한 계수로 알려진 보조 참조 계수를 포함하여 추가 데이터를 포함할 수 있지만 이 항목에서는 이러한 데이터를 무시합니다. std::shared_ptr<T> 객체와 관련된 메모리를 다음과 같이 상상할 수 있습니다.



객체의 제어 블록은 객체에 대한 첫 번째 std::shared_ptr를 생성하는 함수에 의해 설정됩니다. 최소한 그렇게 되어야 합니다. 일반적으로 객체에 대한 std::shared_ptr를 생성하는 함수가 다른 std::shared_ptr가 이미 해당 객체를 가리키고 있는지 여부를 아는 것은 불가능하므로 다음과 같은 제어 블록 생성 규칙이 사용됩니다.

- std::make_shared(항목 21 참조) 는 항상 제어 블록을 생성합니다. 가리킬 새 객체를 제조하므로 std::make_shared가 호출될 때 해당 객체에 대한 제어 블록이 확실히 없습니다.
- std::shared_ptr가 고유 소유권 포인터(예: std::unique_ptr 또는 std::auto_ptr)에서 구성될 때 제어 블록이 생성됩니다. 고유 소유권 포인터는 제어 블록을 사용하지 않으므로 가리킨 객체에 대한 제어 블록이 없어야 합니다. (구성의 일부로 std::shared_ptr은 가리키는 객체의 소유권을 가정하므로 고유 소유권 포인터가 null로 설정됩니다.)

- std::shared_ptr 생성자가 원시 포인터로 호출되면 제어 블록이 생성됩니다. 이미 제어 블록이 있는 객체에서 std::shared_ptr를 생성하려면 원시 포인터가 아닌 생성자 인수로 std::shared_ptr 또는 std::weak_ptr(항목 20 참조)을 전달해야 합니다.. . std::shared_ptr 또는 std::weak_ptrs를 생성자 인수로 사용하는 std::shared_ptr 생성자는 필요한 제어 블록을 가리키기 위해 전달된 스마트 포인터에 의존할 수 있기 때문에 새 제어 블록을 생성하지 않습니다.

이러한 규칙의 결과는 단일 원시 포인터에서 둘 이상의 std::shared_ptr를 구성하면 가리키는 객체에 여러 제어 블록이 있기 때문에 정의되지 않은 동작의 입자 가속기를 무료로 이용할 수 있다는 것입니다. 다중 제어 블록은 다중 참조 카운트를 의미하고 다중 참조 카운트는 객체가 여러 번 파괴됨을 의미합니다(각 참조 카운트에 대해 한 번). 이는 다음과 같은 코드가 나쁘고 나쁘고 나쁘다는 것을 의미합니다.

```
자동 pw = 새 위젯; // pw는 원시 ptr입니다.

...
std::shared_ptr<위젯> spw1(pw, loggingDel); // 컨트를 생성 // *pw에 대한 블록

...
std::shared_ptr<위젯> spw2(pw, loggingDel); // 2번째 생성
// 제어 블록 // for *pw!
```

동적으로 할당된 객체에 대한 원시 포인터 pw의 생성은 이 전체 장의 조언과 반대로 실행되기 때문에 좋지 않습니다. 원시 포인터보다 스마트 포인터를 선호하는 것입니다. (만일 그 충고에 대한 동기를 잊었다면 [115페이지의 내용을 새롭게 기억하십시오.](#)) 그러나 그런 것은 제쳐 두십시오. pw를 생성하는 라인은 문체를 혐오하지만 적어도 정의되지 않은 프로그램 동작을 일으키지는 않습니다.

이제 spw1의 생성자는 원시 포인터로 호출되므로 가리키는 항목에 대한 제어 블록(및 이에 따른 참조 카운트)을 생성합니다. 이 경우, 그것은 *pw(즉, pw가 가리키는 객체)입니다. 그 자체로는 괜찮지만 spw2의 생성자는 동일한 원시 포인터로 호출되므로 *pw에 대한 제어 블록(따라서 참조 카운트)도 생성합니다. 따라서 *pw는 두 개의 참조 카운트를 가지며 각각은 결국 0이 되고 결국 *pw를 두 번 파괴하려는 시도로 이어집니다.

두 번째 파괴는 정의되지 않은 동작에 대한 책임이 있습니다.

여기에 std::shared_ptr 사용에 관한 적어도 두 가지 교훈이 있습니다. 먼저 std::shared_ptr 생성자에 원시 포인터를 전달하지 않도록 하십시오. 일반적인 대안은 std::make_shared(항목 21 참조)를 사용하는 것이지만 위의 예에서는 cus-

tom deleters, std::make_shared에서는 불가능합니다. 둘째, 원시 포인터를 std::shared_ptr 생성자에 전달해야 하는 경우 원시 포인터 변수를 거치지 않고 new의 결과를 직접 전달하십시오. 위 코드의 첫 부분을 이렇게 다시 작성했다면,

```
std::shared_ptr<Widget> spw1(새 위젯, loggingDel); // new를 직접 사용
```

동일한 원시 포인터에서 두 번째 std::shared_ptr을 만드는 것이 훨씬 덜 유혹적일 것입니다. 대신 spw2를 생성하는 코드의 작성자는 자연스럽게 spw1을 초기화 인수로 사용하고(즉, std::shared_ptr 복사 생성자를 호출함) 이는 아무 문제가 되지 않습니다.

```
std::shared_ptr<Widget> spw2(spw1); // spw2는 // spw1과 동일한 제어 블록을 사용합니다.
```

원시 포인터 변수를 std::shared_ptr 생성자 인수로 사용하면 this 포인터와 관련된 여러 제어 블록으로 이어질 수 있다는 특히 놀라운 방법입니다.

프로그램이 std::shared_ptrs를 사용하여 위젯 개체를 관리하고 처리된 위젯을 추적하는 데이터 구조가 있다고 가정합니다.

```
std::vector<std::shared_ptr<Widget>> 처리된 위젯;
```

또한 Widget에 처리를 수행하는 멤버 함수가 있다고 가정합니다.

클래스 위젯 { 공개:

```
...
무효 프로세스();
...
};
```

Widget::process에 대한 합리적인 접근 방식은 다음과 같습니다.

```
무효 위젯::process() {
    ...
    // 위젯 처리
    처리된Widgets.emplace_back(이것); // 처리된 위젯 목록에 // 추가합니다. // 이것은 틀렸다!
}
```

이것이 틀렸다는 의견이 모든 것을 말해줍니다. 또는 적어도 대부분입니다. (잘못된 부분은 emplace_back의 사용이 아니라 this의 전달입니다. emplace_back에 익숙하지 않은 경우 항목 42를 참조하십시오.) 이 코드는 컴파일되지만 컨테이너에 원시 포인터(this)를 전달합니다. std::shared_ptrs. 이렇게 구성된 std::shared_ptr은 가리키는 위젯(*this)에 대한 새 제어 블록을 생성합니다. 저것

이미 해당 위젯을 가리키는 멤버 함수 외부에 std::shared_ptrs가 있는 경우 게임, 설정 및 정의되지 않은 동작에 대한 일치라는 것을 깨달을 때까지는 해롭지 않게 들립니다.

std::shared_ptr API에는 이러한 상황을 위한 기능이 포함되어 있습니다. 아마도 표준 C++ 라이브러리에 있는 모든 이름 중 가장 이상한 이름인 std::enable_shared_from_this를 가질 것입니다. 이것은 std::shared_ptrs가 관리하는 클래스가 this 포인터에서 std::shared_ptr을 안전하게 생성할 수 있도록 하려면 상속받은 기본 클래스에 대한 템플릿입니다. 이 예에서 위젯은 다음과 같이 std::enable_shared_from_this에서 상속합니다.

클래스 위젯: 공개 std::enable_shared_from_this<Widget> { 공개:

```
...
무효 프로세스();
...
};
```

내가 말했듯이 std::enable_shared_from_this는 기본 클래스 템플릿입니다. 유형 매개변수는 항상 파생되는 클래스의 이름이므로 Widget은 std::enable_shared_from_this<Widget>에서 상속됩니다. 파생 클래스에 템플릿화된 기본 클래스에서 파생 클래스를 상속한다는 아이디어가 머리를 아프게 하는 경우에는 그것에 대해 생각하지 마십시오. 코드는 완전히 합법적이고 그 뒤에 있는 디자인 패턴은 매우 잘 정립되어 있지만 표준 이름이 있지만 std::enable_shared_from_this만큼 이상합 니다. 이름은 CRTP(The Curiously Recurring Template Pattern)입니다. 그것에 대해 더 알고 싶다면 검색 엔진을 최대한 활용하세요. 여기에서 std::enable_shared_from_this로 돌아가야 하기 때문입니다.

std::enable_shared_from_this는 현재 개체에 std::shared_ptr을 생성하는 멤버 함수를 정의 하지만 제어 블록을 복제하지 않고 수행합니다. 멤버 함수는 shared_from_this이며 this 포인터와 동일한 객체를 가리키는 std::shared_ptr이 필요할 때마다 멤버 함수에서 사용합니다. Widget::process의 안전한 구현은 다음과 같습니다.

```
무효 위젯::process() {
    ...
    // 이전과 마찬가지로 위젯을 처리합니다.
    ...

    // 현재 객체에 std::shared_ptr을 추가하여 처리된 위젯에 추가합니다
    .processedWidgets.emplace_back(shared_from_this());
}
```

내부적으로 `shared_from_this`는 현재 객체에 대한 제어 블록을 조회하고 해당 제어 블록을 참조하는 새 `std::shared_ptr`를 만듭니다. 디자인은 연결된 제어 블록이 있는 현재 객체에 의존합니다. 그러면 현재 객체를 가리키는 기존 `std::shared_ptr`(예: `shared_from_this`를 호출하는 멤버 함수 외부에 하나)이 있어야 합니다. 그러한 `std::shared_ptr`가 존재하지 않는 경우(즉, 현재 객체에 연결된 제어 블록이 없는 경우) `shared_from_this`가 일반적으로 예외를 `throw`하지만 동작이 정의되지 않습니다.

클라이언트가 `std::shared_ptr` 객체를 가리키기 전에 `shared_from_this`를 호출하는 멤버 함수를 호출하는 것을 방지하기 위해 `std::enable_shared_from_this`에서 상속하는 클래스는 종종 생성자를 `private`로 선언하고 클라이언트가 `std::shared_ptr`s를 반환하는 팩토리 함수를 호출하여 객체를 생성하도록 합니다. 예를 들어 위젯은 다음과 같을 수 있습니다.

```
class Widget: public std::enable_shared_from_this<Widget> { public: // args를
개인 ctor template<typename... Ts>로 완벽하게 전달하는 팩토리 함수 // static
    std::shared_ptr<Widget> create(Ts&&... 매개변수);
```

```
...
무효 프로세스(); // 이전과
...
```

사적인:
사적인:
... // 액터
};

지금쯤이면 제어 블록에 대한 논의가 `std::shared_ptr`s와 관련된 비용을 이해하려는 열망에 의해 동기가 부여되었다는 것을 희미하게 기억할 것입니다. 너무 많은 제어 블록을 생성하지 않는 방법을 이해했으므로 이제 원래 주제로 돌아가겠습니다.

제어 블록은 일반적으로 크기가 몇 단어에 불과하지만 사용자 지정 삭제기와 할당기로 인해 더 커질 수 있습니다. 일반적인 제어 블록 구현은 예상보다 더 정교합니다. 상속을 사용하고 가상 기능도 있습니다. (가리키는 객체가 제대로 파괴되었는지 확인하는 데 사용됩니다.)

즉, `std::shared_ptr`s를 사용하면 제어 블록에서 사용하는 가상 기능에 대한 기계 비용도 발생합니다.

동적으로 할당된 제어 블록, 임의의 큰 삭제자 및 할당자, 가상 함수 기계, 원자 참조 카운트 조작에 대해 읽은 후 `std::shared_ptr`s에 대한 열정이 다소 줄어들었을 수 있습니다. 괜찮아.

모든 리소스 관리 문제에 대한 최상의 솔루션은 아닙니다. 그러나 그들이 제공하는 기능에 대해 std::shared_ptr는 매우 합리적인 비용을 요구합니다. 기본 삭제자와 기본 할당자가 사용되고 std::shared_ptr이 std::make_shared에 의해 생성되는 일반적인 조건에서 제어 블록의 크기는 약 3단어에 불과하며 할당은 기본적으로 무료입니다. (그것은 가리키는 객체에 대한 메모리 할당에 통합됩니다. 자세한 내용은 [항목 21](#)을 참조하십시오.)

std::shared_ptr을 역참조하는 것은 원시 포인터를 역참조하는 것보다 더 비싸지 않습니다. 참조 카운트 조작(예: 복사 생성 또는 복사 할당, 파괴)을 필요로 하는 작업을 수행하려면 하나 또는 두 개의 원자적 작업이 필요하지만 이러한 작업은 일반적으로 개별 기계 명령에 매핑되므로 비원자 명령에 비해 비용이 많이 들 수 있습니다. , 여전히 하나의 지침일 뿐입니다. 제어 블록의 가상 기능 기계는 일반적으로 객체가 파괴될 때 std::shared_ptr에 의해 관리되는 객체당 한 번만 사용됩니다.

이러한 적당한 비용 대신 동적으로 할당된 리소스의 수명 주기를 자동으로 관리할 수 있습니다. 대부분의 경우 std::shared_ptr을 사용하는 것이 공유 소유권을 가진 객체의 수명을 손으로 관리하는 것보다 훨씬 더 좋습니다. std::shared_ptr을 사용할 수 있는지 의심스럽다면 공유 소유권이 정말로 필요한지 다시 생각해 보세요. 배타적 소유권이 가능하거나 할 수 있는 경우 std::unique_ptr이 더 나은 선택입니다. 성능 프로필은 원시 포인터의 프로필에 가깝고 std::unique_ptr에서 std::shared_ptr로의 "업그레이드"가 쉽습니다. std::shared_ptr이 std::unique_ptr에서 생성될 수 있기 때문입니다.

그 반대는 사실이 아닙니다. 리소스의 평생 관리를 std::shared_ptr로 전환한 후에는 마음이 바뀌지 않습니다. 참조 횟수가 하나라도 std::unique_ptr이 관리하도록 하기 위해 리소스의 소유권을 회수할 수 없습니다. 리소스와 리소스를 가리키는 std::shared_ptr 간의 소유권 계약은 'til-death-do-us-part' 다양성입니다. 이혼도, 무효화도, 유예도 없다.

std::shared_ptr가 할 수 없는 다른 것은 배열 작업입니다. std::unique_ptr과의 또 다른 차이점으로 std::shared_ptr에는 단일 객체에 대한 포인터 전용으로 설계된 API가 있습니다. std::shared_ptr<T[]>이 없습니다. 때때로 "영리한" 프로그래머는 std::shared_ptr<T>를 사용하여 배열을 가리키고 배열 삭제(즉, delete [])를 수행하기 위해 사용자 지정 삭제기를 지정하는 아이디어를 우연히 발견합니다. 이것은 컴파일하도록 만들 수 있지만 끔찍한 생각입니다. 우선 std::shared_ptr은 operator[]를 제공하지 않으므로 배열에 대한 인덱싱에는 포인터 산술에 기반한 어색한 표현식이 필요합니다. 다른 예로, std::shared_ptr은 단일 객체에 대해 의미가 있지만 배열에 적용할 때 유형 시스템에 구멍을 열어주는 파생-베이스 포인터 변환을 지원합니다. (이러한 이유로,

`std::unique_ptr<T[]>` API는 이러한 변환을 금지합니다.) 가장 중요한 것은 내장 배열에 대한 다양한 C++11 대안(예: `std::array`, `std::vector`, `std::string`)이 있다는 것입니다., 명칭한 배열에 대한 스마트 포인터를 선언하는 것은 거의 항상 잘못된 디자인의 신호입니다.

기억할 사항 •

`std::shared_ptr`s는 임의 자원의 공유 수명 관리를 위해 가비지 수집에 가까운 편의성을 제공합니다.

- `std::unique_ptr`에 비해 `std::shared_ptr` 객체는 일반적으로 두 배 더 크고 제어 블록에 대한 오버헤드가 발생하며 원자적 참조 카운트 조작이 필요합니다.
- 기본 리소스 삭제는 삭제를 통해 이루어지지만 사용자 지정 삭제자가 지원됩니다. 삭제자의 유형은 `std::shared_ptr`의 유형에 영향을 미치지 않습니다.
- 원시 포인터 유형의 변수에서 `std::shared_ptr`s를 생성하지 마십시오.

항목 20: `std::shared_ptr`에 `std::weak_ptr`을 사용하라 매달릴 수 있는 포인터처럼.

역설적이게도 `std::shared_ptr`처럼 작동하는 스마트 포인터를 사용하는 것이 편리할 수 있지만(항목 19 참조), 가리키는 리소스의 공유 소유권에는 참여하지 않습니다. 즉, `std::shared_ptr`과 같은 포인터로 개체의 참조 횟수에 영향을 주지 않습니다. 이러한 종류의 스마트 포인터는 `std::shared_ptr`s에 알리지 않은 문제와 싸워야 합니다. 즉, 가리키는 것이 파괴될 가능성이 있습니다. 진정한 스마트 포인터는 매달려 있을 때, 즉 포인터가 가리키는 대상이 더 이상 존재하지 않을 때를 추적하여 이 문제를 처리합니다. 이것이 바로 스마트 포인터 `std::weak_ptr`의 종류입니다.

`std::weak_ptr`이 어떻게 유용할 수 있는지 궁금할 것입니다. `std::weak_ptr` API를 조사하면 아마 더 궁금해질 것입니다. 그것은 똑똑하지 않은 것처럼 보입니다. `std::weak_ptr`s는 역참조될 수 없으며 null 여부를 테스트할 수도 없습니다.

`std::weak_ptr`이 독립형 스마트 포인터가 아니기 때문입니다. `std::shared_ptr`의 확장입니다.

관계는 태어날 때부터 시작됩니다. `std::weak_ptr`s는 일반적으로 `std::shared_ptr`s에서 생성됩니다. 그것들은 초기화하는 `std::shared_ptr`s와 같은 장소를 가리키지만 그들이 가리키는 객체의 참조 횟수에는 영향을 미치지 않습니다:

```
자동 spw = // spw가 생성된 후 // 가리킨 위치의
           std::make_shared<위젯>();
```

```

// 참조 카운트(RC)는 1입니다. ( //
std::make_shared에 대한 정보는 // 항목
21 을 참조하십시오.)
...
std::weak_ptr<위젯> wpw(spw); // wpw는 spw와 같은 위젯을 // 가리킵니다. RC는 1로 남
습니다.
...
spw = nullptr; // RC는 0이 되고 // 위젯은 소멸됩니다.
// wpw가 이제 매달려 있습니다.

```

매달린 std::weak_ptrs는 만료되었다고 합니다. 이를 직접 테스트할 수 있으며,

만약 (wpw.expired()) ...	// wpw가 // 객체를 가리키지 않는 다면...
------------------------	---------------------------------

그러나 종종 원하는 것은 std::weak_ptr이 만료되었는지 확인하고 만료되지 않은 경우(즉, 매달려 있지 않은 경우) 가리키는 개체에 액세스하는 것입니다. 이것은 원하는 것보다 쉽습니다.

std::weak_ptrs에는 역참조 작업이 없기 때문에 코드를 작성할 방법이 없습니다. 존재하더라도 검사와 역참조를 분리하면 경쟁 조건이 발생합니다. 만료된 호출과 역참조 작업 사이에 다른 스레드가 개체를 가리키는 마지막 std::shared_ptr을 재할당하거나 파괴할 수 있으므로 해당 개체가 파괴 됩니다. 이 경우 역참조는 정의되지 않은 동작을 생성합니다.

필요한 것은 std::weak_ptr이 만료되었는지 확인하고 만료되지 않은 경우 가리키는 개체에 대한 액세스를 제공하는 원자적 작업입니다. 이것은 std::weak_ptr에서 std::shared_ptr을 생성하여 수행됩니다. 작업은 std::shared_ptr을 만들려고 할 때 std::weak_ptr이 만료된 경우 발생하려는 상황에 따라 두 가지 형식으로 제공됩니다. 한 가지 형식은 std::weak_ptr::lock이며 std::shared_ptr을 반환합니다. std::weak_ptr이 만료된 경우 std::shared_ptr은 null입니다.

std::shared_ptr<위젯> spw1 = wpw.lock(); // wpw가 만료된 경우 // spw1은 null입니다.

자동 spw2 = wpw.lock();	// 위와 같지만 // auto를 사용한다.
-----------------------	-----------------------------

다른 형식은 std::weak_ptr을 인수로 사용하는 std::shared_ptr 생성자입니다. 이 경우 std::weak_ptr이 만료되면 예외가 발생합니다.

std::shared_ptr<위젯> spw3(wpw);	// wpw가 만료되면 //
--------------------------------	-----------------

std::bad_weak_ptr을 던집니다.

그러나 std::weak_ptrs가 어떻게 유용할 수 있는지 여전히 궁금할 것입니다. 고유 ID를 기반으로 읽기 전용 개체에 대한 스마트 포인터를 생성하는 팩토리 함수를 고려하십시오. 팩토리 함수 반환 유형에 관한 항목 18의 조언에 따라 std::unique_ptr을 반환합니다.

```
std::unique_ptr<const 위젯> loadWidget(WidgetID ID);
```

loadWidget이 비용이 많이 드는 호출이고(예: 파일 또는 데이터베이스 I/O를 수행하기 때문에) ID가 반복적으로 사용되는 것이 일반적이라면 합리적인 최적화는 loadWidget이 수행하는 작업을 수행하지만 결과를 캐시하는 함수를 작성하는 것입니다. 요청된 모든 위젯으로 캐시가 막히면 그 자체로 성능 문제가 발생할 수 있으므로 다른 합리적인 최적화는 캐시된 위젯이 더 이상 사용되지 않을 때 폐기하는 것입니다.

이 캐싱 팩토리 함수의 경우 std::unique_ptr 반환 유형이 적합하지 않습니다.

호출자는 캐시된 개체에 대한 스마트 포인터를 확실히 받아야 하고 호출자는 해당 개체의 수명을 확실히 결정해야 하지만 캐시에도 개체에 대한 포인터가 필요합니다. 캐시의 포인터는 매달려 있을 때 감지할 수 있어야 합니다. 팩토리 클라이언트가 팩토리에서 반환된 개체를 사용하여 완료되면 해당 개체가 파괴되고 해당 캐시 항목이 매달려 있기 때문입니다. 따라서 캐시된 포인터는 std::weak_ptrs여야 합니다. 매달릴 때 이를 감지할 수 있는 포인터입니다.

이는 팩토리의 반환 유형이 std::shared_ptr이어야 함을 의미합니다. 왜냐하면 std::weak_ptrs는 객체의 수명이 std::shared_ptrs에 의해 관리될 때만 매달려 있을 때를 감지할 수 있기 때문입니다.

다음은 loadWidget의 캐싱 버전을 간단하게 구현한 것입니다.

```
std::shared_ptr<const 위젯> fastLoadWidget(WidgetID id) { 정적  
std::unordered_map<WidgetID, std::weak_ptr<const 위젯>> 캐시;
```

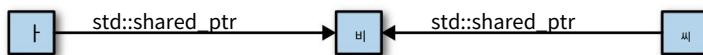
자동 objPtr = 캐시[id].lock(); // objPtr은 std::shared_ptr입니다 // 캐시된 객체로 (또는 객체가 캐시에 없는 경우 // null)

```
if (!objPtr) { objPtr  
= loadWidget(id); 캐시[id] =  
objPtr; } 반환 objPtr;  
  
}  
}
```

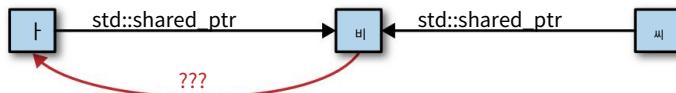
이 구현은 C++11의 해시 테이블 컨테이너(`std::unordered_map`) 중 하나를 사용하지만 존재해야 하는 WidgetID 해싱 및 동등 비교 기능은 표시하지 않습니다.

`fastLoadWidget`의 구현은 캐시가 더 이상 사용되지 않는(따라서 파괴된) 위젯에 해당하는 만료된 `std::weak_ptr`를 누적할 수 있다는 사실을 무시합니다. 구현을 개선할 수 있지만 `std::weak_ptr`에 대한 추가 통찰력을 제공하지 않는 문제에 시간을 할애하기보다는 두 번째 사용 사례인 관찰자 디자인 패턴을 고려해 보겠습니다. 이 패턴의 주요 구성 요소는 주체(상태가 변경될 수 있는 객체)와 관찰자(상태 변경이 발생할 때 알림을 받는 객체)입니다. 대부분의 구현에서 각 주제는 관찰자에 대한 포인터를 보유하는 데이터 멤버를 포함합니다. 이는 주체가 상태 변경 알림을 쉽게 발행할 수 있도록 합니다. 피험자는 관찰자의 수명을 제어하는 데 관심이 없지만(즉, 언제 파괴되는지), 관찰자가 파괴되면 주체가 이후에 관찰자에 액세스하려고 시도하지 않도록 하는 데에는 큰 관심이 있습니다.. 합리적인 디자인은 각 주체가 `std::weak_ptr`의 컨테이너를 관찰자에게 보유하도록 하여 주체가 포인터를 사용하기 전에 포인터가 매달려 있는지 여부를 결정할 수 있도록 하는 것입니다.

`std::weak_ptr` 유ти리티의 마지막 예로서 객체가 있는 데이터 구조를 고려하십시오.
A, B 및 C에서 A와 C는 B의 소유권을 공유하므로 `std::shared_ptr`를 보유합니다.



B에서 A로 다시 포인터를 갖는 것도 유용하다고 가정합니다. 이것은 어떤 종류의 포인터여야 합니까?



세 가지 선택 사항이 있습니다.

- 원시 포인터. 이 접근 방식을 사용하면 A가 파괴되지만 C가 계속 B를 가리킬 경우 B에는 매달릴 A에 대한 포인터가 포함됩니다. B는 이를 감지할 수 없으므로 B는 부주의하게 랭글링 포인터를 참조할 수 있습니다. 정의되지 않은 동작이 발생합니다.
- `std::shared_ptr`. 이 디자인에서 A와 B는 서로 `std::shared_ptr`를 포함합니다. 결과 `std::shared_ptr` 주기(A는 B를 가리키고 B는 A를 가리킴)

A와 B가 모두 파괴되는 것을 방지합니다. A와 B가 다른 프로그램 데이터 구조에서 도달할 수 없는 경우에도(예: C가 더 이상 B를 가리키지 않기 때문에) 각각의 참조 카운트는 1입니다. 그런 일이 발생하면 모든 실용적인 목적을 위해 A와 B가 누출될 것입니다. 프로그램에서 액세스할 수 없지만 해당 리소스는 회수되지 않습니다.

- `std::weak_ptr`. 이렇게 하면 위의 두 가지 문제가 모두 방지됩니다. A가 파괴되면 B의 포인터가 다시 매달려 있지만 B는 이를 감지할 수 있습니다. 또한 A와 B가 서로를 가리킬지라도 B의 포인터는 A의 참조 횟수에 영향을 미치지 않으므로 `std::shared_ptrs`가 더 이상 가리키지 않을 때 A가 파괴되는 것을 막을 수 없습니다.

`std::weak_ptr`을 사용하는 것이 분명히 이러한 선택 중 최고입니다. 그러나 `std::shared_ptrs`의 예상 주기를 깨기 위해 `std::weak_ptrs`를 사용해야 하는 경우가 그리 흔하지 않다는 점은 주목할 가치가 있습니다. 트리와 같은 엄격한 계층적 데이터 구조에서 자식 노드는 일반적으로 부모만 소유합니다. 부모 노드가 파괴되면 자식 노드도 파괴되어야 합니다. 따라서 부모에서 자식으로의 링크는 일반적으로 `std::unique_ptrs`로 가장 잘 표현됩니다. 자식 노드에서 부모로의 백링크는 원시 포인터로 안전하게 구현할 수 있습니다. 자식 노드는 부모보다 수명이 길지 않아야 하기 때문입니다. 따라서 매달린 부모 포인터를 역참조하는 자식 노드의 위험이 없습니다.

물론 모든 포인터 기반 데이터 구조가 엄격하게 계층적인 것은 아니며 캐싱 및 관찰자 목록 구현과 같은 상황에서뿐만 아니라 `std::weak_ptr`이 준비되어 있다는 것을 아는 것이 좋습니다.

효율성 관점에서 `std::weak_ptr` 스토리는 본질적으로 `std::shared_ptr`의 스토리와 동일합니다. `std::weak_ptr` 객체는 `std::shared_ptr` 객체와 크기가 동일하고 `std::shared_ptr`과 동일한 제어 블록을 사용하며([항목 19 참조](#)) 생성, 파괴 및 할당과 같은 작업에는 원자 참조 카운트 조작이 포함됩니다. . 내가 이 항목의 시작 부분에서 `std::weak_ptrs`가 참조 카운팅에 참여하지 않는다고 썼기 때문에 아마 당신을 놀라게 할 것입니다. 내가 쓴 것이 아니라는 점만 빼면. 내가 쓴 것은 `std::weak_ptrs`가 개체의 공유 소유권에 참여하지 않으므로 가리키는 개체의 참조 횟수에 영향을 미치지 않는다는 것입니다. 실제로 제어 블록에는 두 번째 참조 카운트가 있으며 `std::weak_ptrs`가 조작하는 것은 이 두 번째 참조 카운트입니다. 자세한 내용은 [항목 21](#)로 계속 진행하십시오 .

기억해야 할 사항

- 매달릴 수 있는 std::shared_ptr과 유사한 포인터에 std::weak_ptr을 사용합니다. •
- std::weak_ptr의 잠재적 사용 사례에는 캐싱, 관찰자 목록 및 std::shared_ptr 주기 방지가 포함됩니다.

항목 21: new 를 직접 사용하는 것 보다

std::make_unique 및 std::make_shared 를 선호하십시오 .

std::make_unique 및 std::make_shared의 경기장을 평준화하는 것으로 시작하겠습니다.
std::make_shared는 C++11의 일부이지만 슬프게도 std::make_unique는 그렇지 않습니다. C++14부터 표준 라이브러리에 합류했습니다. C++11을 사용하는 경우 std::make_unique의 기본 버전은 직접 작성하기 쉽기 때문에 두려워하지 마십시오.
여기 보세요:

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&... params) {

    return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

보시다시피 make_unique는 해당 매개변수를 생성 중인 객체의 생성자에 완벽하게 전달하고 new가 생성한 원시 포인터에서 std::unique_ptr를 생성하고 생성된 std::unique_ptr를 반환합니다. 이 형식의 함수는 배열이나 사용자 지정 삭제자를 지원하지 않지만(항목 18 참조) 필요한 경우 약간의 노력으로 make_unique를 만들 수 있음을 보여줍니다 . C++14 표준 라이브러리 구현으로 업그레이드할 때 공급업체에서 제공한 버전과 충돌하는 것을 원하지 않기 때문입니다.

std::make_unique 및 std::make_shared는 세 가지 make 함수 중 두 가지입니다. 즉, 임의의 인수 집합을 가져와 동적으로 할당된 객체의 생성자에게 완벽하게 전달하고 해당 객체에 대한 스마트 포인터를 반환하는 함수입니다.

세 번째 make 함수는 std::allocate_shared입니다. 첫 번째 인수가 동적 메모리 할당에 사용할 할당자 객체라는 점을 제외하고는 std::make_shared와 같은 역할을 합니다.

3 가능한 한 최소한의 노력으로 완전한 기능을 갖춘 make_unique를 만들려면 그것을 일으킨 표준화 문서를 검색한 다음 거기에 서 찾을 수 있는 구현을 복사하십시오. 원하는 문서는 2013-04-18 일자 Stephan T. Lavavej의 N3656입니다.

make 함수를 사용한 스마트 포인터 생성과 사용하지 않은 스마트 포인터 생성의 가장 사소한 비교조차도 그러한 함수를 사용하는 것이 선호되는 첫 번째 이유를 보여줍니다. 고려하다:

```
자동 upw1(std::make_unique<위젯>()); // make 함수로
```

```
std::unique_ptr<위젯> upw2(새 위젯); // make 함수 없이
```

```
자동 spw1(std::make_shared<위젯>()); // make 함수로
```

```
std::shared_ptr<위젯> spw2(새 위젯); // make 함수 없이
```

나는 본질적인 차이점을 강조했습니다. new를 사용하는 버전은 생성되는 유형을 반복하지만 make 함수는 그렇지 않습니다. 반복되는 유형은 소프트웨어 엔지니어링의 핵심 원칙에 위배됩니다. 코드 중복을 피해야 합니다. 소스 코드의 복제는 컴파일 시간을 증가시키고 객체 코드를 부풀리게 하며 일반적으로 코드 기반을 작업하기 더 어렵게 만듭니다. 종종 일관성 없는 코드로 발전하고 코드 기반의 불일치로 인해 종종 버그가 발생합니다. 게다가, 무언가를 두 번 타이핑하는 것은 한 번 타이핑하는 것보다 더 많은 노력을 필요로 하고, 누가 타이핑 부담을 줄이는 팬이 아니겠습니까?

make 함수를 선호하는 두 번째 이유는 예외 안전과 관련이 있습니다. 어떤 우선 순위에 따라 위젯을 처리하는 함수가 있다고 가정합니다.

```
무효 processWidget(std::shared_ptr<Widget> spw, int 우선순위);
```

값으로 std::shared_ptr를 전달하는 것이 의심스러울 수 있지만 항목 41 에서는 processWidget이 항상 std::shared_ptr의 복사본을 만드는 경우(예: 처리된 위젯을 추적하는 데이터 구조에 저장하여) 이렇게 할 수 있다고 설명합니다. 합리적인 디자인 선택.

이제 관련 우선 순위를 계산하는 함수가 있다고 가정합니다.

정수 계산 우선 순위();

std::make_shared 대신 new를 사용하는 processWidget에 대한 호출에서 이를 사용합니다.

```
processWidget(std::shared_ptr<Widget>(new Widget), // 잠재적인 computePriority()); //
리소스 // 누출!
```

주석에서 알 수 있듯이 이 코드는 new에 의해 생성된 위젯을 누출할 수 있습니다. 하지만 어떻게? 호출 코드와 호출된 함수는 모두 std::shared_ptrs를 사용하고 있으며 std::shared_ptrs는 리소스 누수를 방지하도록 설계되었습니다. 그들은 자동으로

그곳을 가리키는 마지막 std::shared_ptr이 사라지면 그들이 가리키는 것을 파괴하십시오.

모든 사람이 모든 곳에서 std::shared_ptrs를 사용하고 있다면 이 코드가 어떻게 누출될 수 있습니까?

대답은 컴파일러가 소스 코드를 객체 코드로 번역하는 것과 관련이 있습니다. 런타임에 함수에 대한 인수에 대한 인수는 함수가 호출되기 전에 평가되어야 하므로 processWidget에 대한 호출에서 processWidget이 실행을 시작하기 전에 다음 사항이 발생해야 합니다.

- "new Widget"이라는 표현이 평가되어야 합니다. 즉, Widget이 생성되어야 합니다.
힙에.
- 관리를 담당하는 std::shared_ptr<Widget>의 생성자
new에 의해 생성된 포인터는 실행되어야 합니다.
- computePriority를 실행해야 합니다.

컴파일러는 이 순서대로 실행하는 코드를 생성할 필요가 없습니다. "new Widget"은 std::shared_ptr 생성자
가 호출되기 전에 실행되어야 합니다. 왜냐하면 new의 결과는 해당 생성자에 대한 인수로 사용되기 때문입니다.
그들 사이에.

즉, 컴파일러는 다음 순서로 작업을 실행하는 코드를 내보낼 수 있습니다.

1. "새 위젯"을 수행합니다.
2. computePriority를 실행합니다.
3. std::shared_ptr 생성자를 실행합니다.

이러한 코드가 생성되고 런타임에 computePriority가 예외를 생성하면 1단계에서 동적으로 할당된 위젯이 누출됩니다. 3단계에서 관리를 시작해야 하는 std::shared_ptr에 저장되지 않기 때문입니다.

std::make_shared를 사용하면 이 문제를 피할 수 있습니다. 호출 코드는 다음과 같습니다.

```
processWidget(std::make_shared<Widget>(), // 잠재적인 computePriority()가 없음); // 리소스 누수
```

런타임에 std::make_shared 또는 computePriority가 먼저 호출됩니다. std::make_shared인 경우, 동적으로 할당된 위젯에 대한 원시 포인터는 computePriority가 호출되기 전에 반환된 std::shared_ptr에 안전하게 저장됩니다. 그런 다음 computePriority가 예외를 생성하면 std::shared_ptr 소멸자가 자신이 소유한 위젯이 소멸되었음을 확인합니다. 그리고 computePriority가 먼저 호출되고 예외가 발생하면 std::make_shared가 호출되지 않으므로 걱정할 동적으로 할당된 Widget이 없습니다.

`std::shared_ptr` 및 `std::make_shared`을 `std::unique_ptr` 및 `std::make_unique`로 교체하면 정확히 동일한 추론이 적용됩니다. 따라서 `new` 대신 `std::make_unique`을 사용하는 것은 `std::make_shared`을 사용하는 것만큼 예외로부터 안전한 코드를 작성하는 데 중요합니다.

`std::make_shared`의 특별한 기능(`new`의 직접 사용과 비교하여)은 효율성이 향상되었다는 것입니다. `std::make_shared`을 사용하면 컴파일러가 더 간결한 데이터 구조를 사용하는 더 작고 빠른 코드를 생성할 수 있습니다. `new`의 다음 직접 사용을 고려하십시오.

```
std::shared_ptr<Widget> spw(새 위젯);
```

이 코드는 메모리 할당을 수반하지만 실제로는 두 가지를 수행합니다.

[항목 19](#)는 모든 `std::shared_ptr`이 무엇보다도 가리킨 개체에 대한 참조 횟수를 포함하는 제어 블록을 가리킨다고 설명합니다. 이 제어 블록의 메모리는 `std::shared_ptr` 생성자에 할당됩니다. `new`를 직접 사용하려면 위젯에 대한 메모리 할당과 제어 블록에 대한 두 번째 할당이 필요합니다.

`std::make_shared`가 대신 사용되는 경우,

```
자동 spw = std::make_shared<Widget>();
```

하나의 할당으로 충분합니다. `std::make_shared`가 `Widget` 객체와 제어 블록을 모두 보유하기 위해 단일 메모리 청크를 할당하기 때문입니다. 이 최적화는 코드에 하나의 메모리 할당 호출만 포함하기 때문에 프로그램의 정적 크기를 줄이고 메모리가 한 번만 할당되기 때문에 실행 코드의 속도를 높입니다. 또한 `std::make_shared`를 사용하면 제어 블록에 일부 부기 정보가 필요하지 않으므로 프로그램의 총 메모리 공간이 잠재적으로 줄어듭니다.

`std::make_shared`에 대한 효율성 분석은 `std::allocate_shared`에도 동일하게 적용되므로 `std::make_shared`의 성능 이점은 해당 기능에도 확장됩니다.

`new`를 직접 사용하는 것보다 `make` 함수를 선호하는 주장은 강력합니다.

그러나 소프트웨어 엔지니어링, 예외 안전 및 효율성 이점에도 불구하고 이 항목의 지침은 이러한 기능에 배타적으로 의존하지 않고 `make` 기능을 선호하는 것입니다. 사용할 수 없거나 사용해서는 안 되는 상황이 있기 때문입니다.

예를 들어, 어떤 `make` 함수도 커스텀 삭제자의 지정을 허용하지 않지만([항목 18](#)과 19 참조), `std::unique_ptr`과 `std::shared_ptr` 모두 그렇게 하는 생성자를 가지고 있습니다. 위젯에 대한 사용자 정의 삭제자가 주어지면,

```
자동 위젯 삭제자 = [](위젯* pw) { … };
```

`new`를 사용하여 스마트 포인터를 만드는 것은 간단합니다.

```
std::unique_ptr<위젯, decltype(widgetDeleter)> upw(새 위젯,
    widgetDeleter);
```

```
std::shared_ptr<Widget> spw(새 위젯, widgetDeleter);
```

make 함수로 같은 일을 할 수 있는 방법은 없습니다.

make 함수의 두 번째 제한은 구현의 구문적 세부 사항에서 비롯됩니다. [항목 7](#) 은 std::initializer_list 매개변수가 있거나 없는 생성자를 유형 오버로드하는 객체를 생성할 때 중괄호를 사용하여 객체를 생성하는 것이 std::initializer_list 생성자를 선호하는 반면, 괄호를 사용하여 객체를 생성하면 non-std:를 호출한다고 설명합니다. initializer_list 생성자.

make 함수는 매개변수를 객체의 생성자에 완벽하게 전달하지만 괄호를 사용하거나 중괄호를 사용하여 그렇게 합니까? 일부 유형의 경우 이 질문에 대한 대답이 큰 차이를 만듭니다. 예를 들어 이러한 호출에서

```
자동 upv = 표준::make_unique<표준::벡터<int>>(10, 20);
```

```
자동 spv = std::make_shared<std::vector<int>>(10, 20);
```

결과 스마트 포인터는 각각 값이 20인 10개의 요소가 있는 std::vector를 가리킵니까? 아니면 결과가 불확실한가요?

좋은 소식은 이것이 불확실하지 않다는 것입니다. 두 호출 모두 모든 값이 20으로 설정된 크기 10의 std::vectors를 생성합니다. 즉, make 함수 내에서 완벽한 전달 코드는 중괄호가 아닌 괄호를 사용합니다. 나쁜 소식은 중괄호 이나셜라이저를 사용하여 가리키는 객체를 구성하려면 new를 직접 사용해야 한다는 것입니다.

make 함수를 사용하려면 중괄호 이나셜라이저를 완벽하게 전달할 수 있는 기능이 필요하지만 [항목 30](#) 에서 설명하는 것처럼 중괄호 이나셜라이저는 완벽하게 전달할 수 없습니다. 그러나 [항목 30](#)은 해결 방법도 설명합니다. 자동 유형 추론을 사용하여 중괄호 이나셜라이저에서 std::initializer_list 객체를 만든 다음([항목 2](#) 참조), 자동 생성된 객체를 make 함수를 통해 전달합니다.

```
// std::initializer_list 생성 auto initList =
{ 10, 20 };
```

```
// std::initializer_list를 사용하여 std::vector 생성 ctor auto spv =
std::make_shared<std::vector<int>>(initList);
```

std::unique_ptr의 경우 이러한 두 가지 시나리오(사용자 지정 삭제자 및 중괄호 이나셜라이저)는 make 함수에 문제가 있는 유일한 시나리오입니다. std::shared_ptr 및 해당 make 함수의 경우 두 가지가 더 있습니다. 둘 다 엣지 케이스이지만 일부 개발자는 엣지에 살고 있으며 귀하도 그 중 하나일 수 있습니다.

일부 클래스는 자체 버전의 operator new 및 operator delete를 정의합니다. 이러한 함수의 존재는 이러한 유형의 객체에 대한 전역 메모리 할당 및 할당 해제 루틴이 부적절함을 의미합니다. 종종, 클래스 특정 루틴은 클래스 객체의 정확한 크기의 메모리 덩어리를 할당 및 할당 해제하도록 설계되었습니다. 예를 들어 클래스 Widget에 대한 operator new 및 operator delete는 종종 정확히 sizeof(Widget)의 메모리. 이러한 루틴은 std::allocate_shared 요청의 메모리 양이 동적으로 할당된 메모리의 크기가 아니기 때문에 사용자 지정 할당(std::allocate_shared를 통해) 및 할당 해제(사용자 지정 삭제기를 통해)에 대한 std::shared_ptr 지원에 적합하지 않습니다. 객체의 경우 해당 객체의 크기에 제어 블록의 크기를 더한 것입니다. 따라서 make 함수를 사용하여 operator new 및 operator delete의 클래스별 버전으로 유형의 객체를 만드는 것은 일반적으로 좋지 않은 생각입니다.

std::make_shared의 크기와 속도 이점은 std::shared_ptr의 제어 블록이 관리되는 객체와 동일한 메모리 청크에 배치되는 새 줄기를 직접 사용하는 것과 비교합니다. 해당 객체의 참조 횟수가 0이 되면 객체가 소멸됩니다(즉, 소멸자가 호출됨). 그러나 동적으로 할당된 메모리의 동일한 청크에 둘 다 포함되어 있기 때문에 제어 블록도 파괴될 때까지 점유하는 메모리를 해제할 수 없습니다.

내가 언급했듯이 제어 블록에는 참조 횟수 자체를 넘어선 부기 정보가 포함되어 있습니다. 참조 카운트는 제어 블록을 참조하는 std::shared_ptr의 수를 추적하지만 제어 블록에는 제어 블록을 참조하는 std::weak_ptr의 수를 집계하는 두 번째 참조 카운트가 포함됩니다. 이 두 번째 참조 카운트를 약한 카운트라고 합니다.

⁴ std::weak_ptr이 만료되었는지 확인할 때([항목 19](#) 참조) 참조하는 제어 블록의 참조 카운트(약한 카운트가 아님)를 검사하여 만료됩니다. 참조 카운트가 0이면(즉, 가리킨 객체에 이를 참조하는 std::shared_ptr이 없어 소멸된 경우) std::weak_ptr이 만료된 것입니다. 그렇지 않으면 그렇지 않습니다.

std::weak_ptrs가 제어 블록을 참조하는 한(즉, 약한 개수가 0보다 큰 경우) 해당 제어 블록은 계속 존재해야 합니다. 그리고 제어 블록이 존재하는 한 이를 포함하는 메모리는 할당된 상태로 유지되어야 합니다. std::shared_ptr make 함수에 의해 할당된 메모리는 마지막 std::shared_ptr과 그것을 참조하는 마지막 std::weak_ptr이 파괴될 때까지 할당 해제될 수 없습니다.

⁴ 실제로, 약한 카운트의 값은 제어 블록을 참조하는 std::weak_ptrs의 수와 항상 같지는 않습니다. 라이브러리 구현자가 더 나은 코드 생성을 촉진하는 약한 카운트에 추가 정보를 넣는 방법을 찾았기 때문입니다. 이 항목의 목적을 위해 우리는 이것을 무시하고 약한 카운트의 값이 제어 블록을 참조하는 std::weak_ptrs의 수라고 가정합니다.

객체 유형이 상당히 크고 마지막 파괴 사이의 시간이 있는 경우
`std::shared_ptr`과 마지막 `std::weak_ptr`이 중요하며, 사이에 지연이 발생할 수 있습니다.
 객체가 파괴되고 점유한 메모리가 해제될 때:

```
클래스 RealBigType { ... };

자동 pBigObj =
    std::make_shared<ReallyBigType>(); // 매우 크게 생성
                                         // 객체를 통해
                                         // std::make_shared

... // std::shared_ptrs 및 std::weak_ptrs 생성
   // 큰 개체, 작업에 사용

... // 여기서 파괴된 객체에 대한 최종 std::shared_ptr,
   // 하지만 std::weak_ptrs는 남아 있습니다.

... // 이 기간 동안 이전에 점유되었던 메모리
   // 큰 개체에 의해 할당된 상태로 유지됨

... // 여기서 파괴된 객체에 대한 최종 std::weak_ptr;
   // 제어 블록 및 개체에 대한 메모리가 해제됩니다.
```

`new`를 직접 사용하면 `reallyBigType` 객체의 메모리를 해제할 수 있습니다.

마지막 `std::shared_ptr`이 파괴되자마자:

```
클래스 RealBigType { ... }; // 이전과

std::shared_ptr<ReallyBigType> pBigObj( newReallyBigType);
                                         // 매우 크게 생성
                                         // new를 통한 객체

... // 이전과 같이 std::shared_ptrs를 생성하고
   // std::weak_ptrs를 객체로 사용, 함께 사용

... // 여기서 파괴된 객체에 대한 최종 std::shared_ptr,
   // 하지만 std::weak_ptrs는 남아 있습니다.
   // 객체에 대한 메모리 할당이 해제됩니다.

... // 이 기간 동안 메모리만
   // 제어 블록은 할당된 상태로 유지됩니다.

... // 여기서 파괴된 객체에 대한 최종 std::weak_ptr;
   // 제어 블록의 메모리가 해제됨
```

`std::make_shared`을 사용할 수 없는 상황에 처한 경우
 또는 부적절하면 예외 안전 유형으로부터 자신을 보호하고 싶을 것입니다.

앞에서 본 문제. 이를 수행하는 가장 좋은 방법은 new를 직접 사용할 때 다른 작업을 수행하지 않는 명령문에서 결과를 스마트 포인터 생성자에 즉시 전달하도록 하는 것입니다. 이렇게 하면 컴파일러가 new 사용과 newed 개체를 관리할 스마트 포인터에 대한 생성자 호출 사이에 예외를 생성할 수 있는 코드를 생성하지 못하게 됩니다.

예를 들어, 앞서 조사한 프로세스 위젯 함수에 대한 안전하지 않은 예외 호출에 대한 약간의 수정을 고려하십시오. 이번에는 사용자 지정 삭제자를 지정합니다.

```
void processWidget(std::shared_ptr<Widget> spw, // int 우선순위 이전과 같이);
```

```
무효 cusDel(위젯 *ptr); // 커스텀 // 삭제자
```

다음은 안전하지 않은 예외 호출입니다.

```
프로세스위젯() // 이전과 마찬가지
    로 std::shared_ptr<Widget>(new Widget, cusDel), // 잠재적인 computePriority() //
    resource ); // 누출!
```

회상: "new Widget" 이후에 std::shared_ptr 생성자 이전에 computePriority가 호출되고 computePriority에서 예외가 발생하면 동적으로 할당된 위젯이 누출됩니다.

여기서 사용자 지정 삭제기를 사용하면 std::make_shared를 사용할 수 없으므로 문제를 피하는 방법은 위젯 할당과 std::shared_ptr 생성을 자체 명령문에 넣은 다음 결과로 processWidget을 호출하는 것입니다. std::shared_ptr. 이 기술의 본질은 다음과 같습니다. 잠시 후 살펴보겠지만 성능을 개선하기 위해 조정할 수 있습니다.

```
std::shared_ptr<위젯> spw(새 위젯, cusDel);
processWidget(spw, computePriority()); // 정확하지만 // 최적은 아님;
                                            아래 참조
```

이것은 std::shared_ptr이 생성자가 예외를 생성하더라도 생성자에 전달된 원시 포인터의 소유권을 가정하기 때문에 작동합니다. 이 예에서 spw의 생성자가 예외를 발생시키면(예: 제어 블록에 대한 메모리를 동적으로 할당할 수 없기 때문에) "new Widget"으로 인해 포인터에서 cusDel이 호출된다 는 것이 여전히 보장됩니다.

사소한 성능 장애는 예외 안전하지 않은 호출에서 rvalue를 processWidget에 전달한다는 것입니다.

프로세스위젯(

```
std::shared_ptr<Widget>(new Widget, cusDel), // arg는 rvalue 입 computePriority() );
```

그러나 예외 안전 호출에서 lvalue를 전달합니다.

```
processWidget(spw, computePriority()); // 인수는 lvalue입니다.
```

processWidget의 std::shared_ptr 매개변수는 값으로 전달되기 때문에 rvalue에서 구성하려면 이동만 수반되지만 lvalue에서 구성하려면 복사가 필요합니다. std::shared_ptr의 경우 차이가 클 수 있습니다. std::shared_ptr을 복사하려면 참조 카운트의 원자 단위 증가가 필요하지만 std::shared_ptr을 이동하는 데 참조 카운트 조작이 전혀 필요하지 않기 때문입니다. 예외 안전 코드가 예외 안전 코드의 성능 수준을 달성하려면 spw에 std::move를 적용하여 rvalue로 전환해야 합니다([항목 23](#) 참조).

```
processWidget(std::move(spw), // 효율적이고 // 예외 안전
computePriority());
```

그것은 흥미롭고 알 가치가 있지만 일반적으로 make 함수를 사용하지 않을 이유가 거의 없기 때문에 일반적으로 관련이 없습니다. 그리고 다른 방법으로 해야 할 강력한 이유가 없는 한 make 함수를 사용하는 것이 좋습니다.

기억해야 할 사항

- new를 직접 사용하는 것과 비교하여 make 함수는 소스 코드 중복을 제거하고 예외 안전성을 개선하며 std::make_shared 및 std::allocate_shared의 경우 더 작고 빠른 코드를 생성합니다.
- make 기능의 사용이 부적절한 상황에는 다음이 포함됩니다.
 사용자 지정 삭제자와 종말호 이니셜라이저를 전달하려는 욕구를 지정합니다.
- std::shared_ptrs의 경우 make 함수가 부적절할 수 있는 추가 상황에는 (1) 사용자 지정 메모리 관리가 있는 클래스 및 (2) 메모리 문제가 있는 시스템, 매우 큰 개체 및 해당보다 오래 지속되는 std::weak_ptrs가 포함됩니다. std::shared_ptrs.

항목 22: Pimpl Idiom을 사용할 때 특별한 정의 구현 파일의 멤버 함수.

과도한 빌드 시간과 싸워야 했던 적이 있다면 Pimpl("구현에 대한 포인터") 관용구에 익숙할 것입니다. 이는 클래스의 데이터 멤버를 구현 클래스(또는 구조체)에 대한 포인터로 대체하고

기본 클래스에 있던 데이터 멤버를 구현 클래스로 옮기고 포인터를 통해 간접적으로 해당 데이터 멤버에 액세스합니다. 예를 들어 위젯이 다음과 같다고 가정합니다.

```
클래스 위젯 { 공개: // 헤더 "widget.h"

    위젯();
    ...

    개인: 표준::
        문자열 이름; std::vector<
        더블> 데이터;
        가제트 g1, g2, g3; }; // 가젯은 사용자가 정의한 유형입니다.

    개인: 표준::
```

위젯의 데이터 멤버는 `std::string`, `std::vector` 및 `Gadget` 유형이므로 위젯이 컴파일하려면 해당 유형의 헤더가 있어야 하며 이는 위젯 클라이언트가 `<string>`, `<vector>` 및 `gadget.h`.

이러한 헤더는 위젯 클라이언트의 컴파일 시간을 늘리고 해당 클라이언트를 헤더 내용에 종속시킵니다. 헤더의 내용이 변경되면 위젯 클라이언트는 다시 컴파일해야 합니다. 표준 헤더 `<string>` 및 `<vector>`는 자주 변경되지 않지만, `gadget.h`가 자주 수정될 수 있습니다.

C++98에서 `Pimpl Idiom`을 적용하면 `Widget`이 데이터 멤버를 선언되었지만 정의되지 않은 구조체에 대한 원시 포인터로 대체할 수 있습니다.

```
클래스 위젯 { 공개: // 여전히 "widget.h" 헤더에 있음

    위젯();
    ~위젯(); // dtor가 필요합니다. 아래 참조
    ...

    개인: 구조
        체 Impl; // 구현 구조체 선언 // 및 포인터
        Impl *pImpl; };

    개인: 표준::
```

`Widget`이 더 이상 `std::string`, `std::vector` 및 `Gadget` 유형을 언급하지 않기 때문에 위젯 클라이언트는 더 이상 이러한 유형의 헤더를 `#include`할 필요가 없습니다. 이는 컴파일 속도를 높이고 이러한 헤더의 내용이 변경되더라도 위젯 클라이언트는 영향을 받지 않는다는 의미이기도 합니다.

선언되었지만 정의되지 않은 유형을 불완전 유형이라고 합니다.

`Widget::Impl`이 그런 유형입니다. 불완전한 유형으로 할 수 있는 일은 거의 없지만 이에 대한 포인터를 선언하는 것은 그 중 하나입니다. `Pimpl Idiom`은 이를 활용합니다.

Pimpl Idiom의 1부는 불완전한 유형에 대한 포인터인 데이터 멤버의 선언입니다. 파트 2는 원래 클래스에 있던 데이터 멤버를 보유하는 개체의 동적 할당 및 할당 해제입니다. 할당 및 위치 해제 코드는 구현 파일(예: 위젯의 경우 `widget.cpp`)에 들어갑니다.

```
#include "widget.h"                                // impl에서, 파일 "widget.cpp"
#include "gadget.h"
#include <문자열> #include
<벡터>

struct Widget::Impl { 표준::문
    자열 이름; std::vector<더블
    > 데이터;
    가제트 g1, g2, g3; };

Widget::Widget() : pImpl(new Impl) {}               // 이 위젯 객체에 // 데이터 멤버를 할당합니다.

Widget::~Widget() { pImpl.삭제; }                  // 이 객체의 // 데이터 멤버를 파괴합니다.
```

여기에서는 `std::string`, `std::vector` 및 `Gadget`에 대한 헤더에 대한 전체 종속성이 계속 존재한다는 것을 명확히 하기 위해 `#include` 지시문을 보여주고 있습니다. 그러나 이러한 종속성은 `widget.h`(위젯 클라이언트에 표시되고 사용됨)에서 `widget.cpp`(위젯 구현자에게만 표시되고 사용됨)로 이동되었습니다. 또한 `Impl` 개체를 동적으로 할당 및 할당 해제하는 코드를 강조 표시했습니다. 위젯이 파괴될 때 이 개체를 할당 해제해야 하는 것은 위젯 소멸자가 필요한 이유입니다.

그러나 나는 당신에게 C++98 코드를 보여주었고, 그것은 지난 천년의 냄새를 풍깁니다. 그것은 원시 포인터와 원시 `new` 및 `raw` 삭제를 사용하며 모두 그냥... 원시입니다. 이 장은 스마트 포인터가 원시 포인터보다 더 낫다는 아이디어에 기반을 두고 있으며, 우리가 원하는 것이 위젯 생성자 내부에 `Widget::Impl` 객체를 동적으로 할당하고 위젯이 `std::unique_ptr`인 것과 동시에 소멸되도록 하는 것이라면: `unique_ptr`([항목 18 참조](#))은 정확히 우리에게 필요한 도구입니다. 원시 `pImpl` 포인터를 `std::unique_ptr`로 교체하면 헤더 파일에 대한 이 코드가 생성됩니다.

```
클래스 위젯 { 공개:
    위젯();
    ...
}
```

사적인:

```
구조체 Impl;
std::unique_ptr<Impl> pImpl; }; // 원시 포인터 대신 // 스마트 포
인터 사용
```

구현 파일의 경우:

```
#include "widget.h" // "widget.cpp"에서
#include "gadget.h"
#include <문자열> #include
<벡터>

struct Widget::Impl { 표준::문
    자열 이름; std::vector<더블
    > 데이터;
    가제트 g1, g2, g3; };

// 이전과
```

```
Widget::Widget() // 항목 21에 따라 생성 : pImpl(std::make_unique<Impl>()) //
std::unique_ptr {} // std::make_unique를 통해
```

위젯 소멸자가 더 이상 존재하지 않을 수 있습니다. 그것은 우리가 그것에 넣을 코드가 없기 때문입니다. std::unique_ptr은 std::unique_ptr이 파괴될 때 가리키는 것을 자동으로 삭제하므로 우리는 아무것도 삭제할 필요가 없습니다.

이것이 스마트 포인터의 매력 중 하나입니다. 수동 리소스 릴리스로 손을 더럽힐 필요가 없습니다.

이 코드는 컴파일되지만 가장 사소한 클라이언트 사용은 다음을 수행하지 않습니다.

```
#include "widget.h"

위젯 w; // 오류!
```

받는 오류 메시지는 사용 중인 컴파일러에 따라 다르지만 일반적으로 이 텍스트는 불완전한 유형에 sizeof 또는 delete를 적용하는 것에 대해 언급합니다. 이러한 작업은 이러한 유형으로 수행할 수 있는 작업이 아닙니다.

std::unique_ptrs를 사용하는 Pimpl Idiom의 명백한 실패는 (1) std::unique_ptr이 불완전한 유형을 지원하는 것으로 광고되고 (2) Pimpl Idiom이 std::unique_ptrs 가장 일반적인 사용 사례 중 하나이기 때문에 놀라운 것입니다. 다행히 코드를 작동시키는 것은 쉽습니다. 문제의 원인에 대한 기본적인 이해만 있으면 됩니다.

문제는 w가 소멸될 때 생성되는 코드로 인해 발생합니다(예: 범위를 벗어남). 그 시점에서 소멸자가 호출됩니다. std::unique_ptr을 사용하는 클래스 정의에서 소멸자를 선언하지 않았습니다. 왜냐하면 거기에 넣을 코드가 없었기 때문입니다. 컴파일러 생성 특수 멤버에 대한 일반적인 규칙에 따라

함수([항목 17](#) 참조)에서 컴파일러는 소멸자를 생성합니다. 해당 소멸자 내에서 컴파일러는 위젯의 데이터 멤버 pImpl에 대한 소멸자를 호출하는 코드를 삽입합니다. pImpl은 std::unique_ptr<Widget::Impl>, 즉 기본 삭제자를 사용하는 std::unique_ptr입니다. 기본 삭제자는 std::unique_ptr 내부의 원시 포인터에서 삭제를 사용하는 함수입니다. 그러나 삭제를 사용하기 전에 구현에서는 일반적으로 원시 포인터가 불완전한 유형을 가리키지 않도록 기본 삭제 프로그램에서 C++11의 static_assert를 사용합니다. 컴파일러가 위젯 w의 소멸을 위한 코드를 생성할 때 일반적으로 실패하는 static_assert를 만나며 이것이 일반적으로 오류 메시지로 이어집니다. 이 메시지는 w가 소멸되는 지점과 관련이 있습니다. 왜냐하면 모든 컴파일러 생성 특수 멤버 함수와 마찬가지로 위젯의 소멸자가 암시적으로 인라인되기 때문입니다. 메시지 자체는 종종 w가 생성되는 라인을 참조하는데, 이는 나중에 암시적 파괴로 이어지는 개체를 명시적으로 생성하는 소스 코드이기 때문입니다.

문제를 해결하려면 std::unique_ptr<Widget::Impl>을 파괴하는 코드가 생성되는 지점에서 Widget::Impl이 완전한 유형인지 확인하기만 하면 됩니다. 유형은 정의가 표시되고 Widget::Impl이 widget.cpp 내부에 정의될 때 완료됩니다. 성공적인 컴파일의 핵심은 컴파일러가 Widget::Impl 뒤의 widget.cpp 내부에서만 위젯의 소멸자의 본문(즉, 컴파일러가 std::unique_ptr 데이터 멤버를 파괴하는 코드를 생성할 위치)을 보도록 하는 것입니다. 정의되었습니다.

이를 위한 정리는 간단합니다. 위젯의 소멸자를 widget.h에 선언하되 거기에서 정의하지 마십시오.

```
클래스 위젯 { 공개:
    위젯();
    ~위젯();
    ...
}

개인: 구조                                // 이전과 같이 "widget.h"에서
    체 Impl;
    std::unique_ptr<Impl> pImpl;;
```

Widget::Impl이 정의된 후 widget.cpp에서 정의하십시오.

```
#include "widget.h"                          // 이전과 같이 "widget.cpp"에서
#include "gadget.h"
#include <문자열> #include
<벡터>

struct 위젯::Impl {                       // 이전과 같이 정의
```

```
std::문자열 이름; // 위젯::Impl
std::vector<더블> 데이터;
가제트 g1, g2, g3; };
```

```
위젯::위젯(): // 이전과
pImpl(std::make_unique<Impl>()) {}
```

```
위젯::~위젯() {} // ~위젯 정의
```

이것은 잘 작동하고 최소한의 타이핑이 필요하지만 컴파일러가 생성한 소멸자가 올바른 일을 할 것이라는 점을 강조하고 싶다면(당신이 선언한 유일한 이유는 위젯의 구현 파일에서 정의가 생성되도록 하기 위함), "= default"로 소멸자 본문을 정의할 수 있습니다.

```
위젯::~Widget() = 기본값; // 위와 같은 효과
```

컴파일러 생성 이동 작업은 기본 std::unique_ptr에서 이동을 수행하는 것과 같이 정확히 원하는 작업을 수행하기 때문에 Pimpl Idiom을 사용하는 클래스는 이동 지원을 위한 자연스러운 후보입니다. 항목 17에서 설명하는 것처럼 Widget에서 소멸자의 선언은 컴파일러가 이동 작업을 생성하는 것을 방지하므로 이동 지원을 원하는 경우 함수를 직접 선언해야 합니다. 컴파일러 생성 버전이 올바르게 작동한다는 점을 감안할 때 다음과 같이 구현하고 싶을 것입니다.

```
클래스 위젯 { 공개: // 여전히 //
    위젯(); // "widget.h"에 있음
    ~위젯();
```

위젯(위젯&& rhs) = 기본값; // 옳은 생각, Widget& operator=(Widget&& rhs) = 기본값; //
잘못된 코드!

...

```
개인: 구조 // 이전과
체 Impl;
std::unique_ptr<Impl> pImpl;};
```

이 접근 방식은 소멸자 없이 클래스를 선언하는 것과 같은 종류의 문제를 야기하며 동일한 근본적인 이유에서 발생합니다. 컴파일러가 생성한 이동 할당 연산자는 재할당하기 전에 pImpl이 가리키는 객체를 파괴해야 하지만 위젯 헤더 파일에서 pImpl은 불완전한 유형을 가리킵니다. 상황-

이동 생성자는 다릅니다. 문제는 컴파일러가 일반적으로 내부에서 예외가 발생하는 경우 pImpl을 파괴하는 코드를 생성합니다. 이동 생성자와 pImpl을 파괴하려면 Impl이 완료되어야 합니다.

문제는 이전과 동일하기 때문에 수정도 마찬가지입니다.

작업을 구현 파일로 이동:

클래스 위젯 { 공개: // 여전히 "widget.h"에 있음

```
위젯();
~위젯();
```

```
위젯(위젯&& rhs); // 선언
위젯& 연산자=(위젯&& rhs); // 만
```

...

```
개인: 구조 // 이전과
체 Impl;
std::unique_ptr<Impl> pImpl;
};

#include <문자열> // 이전과,
...
// "widget.cpp"에서

struct 위젯::Impl { … }; // 이전과

Widget::Widget() : // 이전과
pImpl(std::make_unique<Impl>())
{}

위젯::~Widget() = 기본값; // 이전과
```

```
Widget::Widget(Widget&& rhs) = 기본값; // 정의
Widget& Widget::operator=(Widget&& rhs) = 기본값; // 작업
```

Pimpl Idiom은 클래스 간의 컴파일 종속성을 줄이는 방법입니다. 구현 및 클래스의 클라이언트, 그러나 개념적으로 관용구의 사용은 클래스가 나타내는 것을 변경합니다. 원래 Widget 클래스에는 std::string, std::vector 및 Gadget 데이터 멤버, 그리고 Gadgets가 다음과 같다고 가정합니다. std::strings 및 std::vectors는 복사할 수 있습니다. Widget이 복사 작업을 지원합니다. 우리는 이러한 함수를 직접 작성해야 합니다. (1) 컴파일러는 다음과 같은 이동 전용 유형이 있는 클래스에 대해 복사 작업을 생성하지 않습니다. std::unique_ptr 및 (2) 그렇게 하더라도 생성 된 함수는 복사 만합니다.

`std::unique_ptr` (즉, 얇은 복사 수행), 그리고 우리는 무엇을 복사하기를 원합니까?
포인터가 가리키는(즉, 깊은 복사를 수행).

이제 익숙한 의식에서 헤더 파일에 함수를 선언하고
구현 파일에서 구현하십시오.

```
클래스 위젯 { 공개:                                     // 여전히 "widget.h"에 있음
...
// 이전과 같은 다른 함수

위젯(const 위젯 및 rhs); // 선언
위젯& 연산자=(const 위젯& rhs); // 만

개인: 구조                                         // 이전과
체 Impl;
std::unique_ptr<Impl> pImpl;
};

#include "widget.h"                                // 이전과,
...
// "widget.cpp"에서

struct 위젯::Impl { … };                         // 이전과

위젯::~Widget() = 기본값;                       // 이전과 같은 다른 함수

Widget::Widget(const Widget& rhs) :             // ctor 복사
pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs) // 복사 연산자=
{
    *pImpl = *rhs.pImpl;
    반환 *이;
}
```

두 함수 구현은 모두 관습적입니다. 각각의 경우에 우리는 단순히
소스 개체(rhs)에서 대상 개체까지 `Impl` 구조체의 필드
(*이것). 필드를 하나씩 복사하는 대신
컴파일러는 `Impl`에 대한 복사 작업을 생성하고 이러한 작업은
각 필드를 자동으로 따라서 다음을 호출하여 위젯의 복사 작업을 구현합니다.
`Widget::Impl`의 컴파일러 생성 복사 작업. 복사 생성자에서 주의
우리는 여전히 `std::make_unique`보다 `std::make_unique` 사용을 선호하는 항목 21 의 조언을 따릅니다.
새것의 직접적인 사용.

Pimpl Idiom을 구현하기 위해 std::unique_ptr은 사용할 스마트 포인터입니다. 왜냐하면 객체 내부(예: Widget 내부)의 pImpl 포인터는 해당 구현 객체(예: Widget::Impl 객체)에 대한 독점적 소유권을 갖기 때문입니다.). 그래도 pImpl에 대해 std::unique_ptr 대신 std::shared_ptr를 사용하면 이 항목의 조언이 더 이상 적용되지 않는다는 것을 알게 된다는 점은 흥미롭습니다. Widget에서 소멸자를 선언할 필요가 없으며 사용자가 선언한 소멸자가 없으면 컴파일러는 우리가 원하는 것을 정확히 수행하는 이동 작업을 행복하게 생성할 것입니다. 즉, widget.h에 이 코드가 주어지면,

```
클래스 위젯 { 공개:
    위젯();
    ...
    // "widget.h"에서
}

개인: 구조
체 Impl;
std::shared_ptr<Impl> pImpl; };           // std::shared_ptr //
                                                std::unique_ptr 대신
                                                // dtor에 대한 선언 없음 // 또는 이동 작업
```

위젯.h를 #포함하는 이 클라이언트 코드,

```
위젯 w1;
자동 w2(std::이동(w1));                  // 이동 구성 w2
w1 = 표준::이동(w2);                   // 이동 할당 w1
```

모든 것이 우리가 바라는 대로 컴파일되고 실행될 것입니다. w1은 기본적으로 생성되고, 그 값은 w2로 이동되고, 그 값은 다시 w1으로 이동되고, w1과 w2가 모두 파괴됩니다(따라서 가리키는 대상이 Widget::Impl 객체가 소멸됨).

pImpl 포인터에 대한 std::unique_ptr과 std::shared_ptr 간의 동작 차이는 이러한 스마트 포인터가 사용자 지정 삭제자를 지원하는 방식이 다르기 때문입니다. std::unique_ptr의 경우 삭제기 유형은 스마트 포인터 유형의 일부이며 이를 통해 컴파일러는 더 작은 런타임 데이터 구조와 더 빠른 런타임 코드를 생성할 수 있습니다. 이러한 효율성 향상의 결과는 컴파일러 생성 특수 함수(예: 소멸자 또는 이동 작업)를 사용할 때 지정 유형이 완전해야 한다는 것입니다. std::shared_ptr의 경우 삭제자의 유형은 스마트 포인터 유형의 일부가 아닙니다. 이것은 더 큰 런타임 데이터 구조와 다소 느린 코드를 필요로 하지만 컴파일러 생성 특수 기능이 사용될 때 지정 유형이 완전할 필요는 없습니다.

Pimpl 관용구의 경우 std::unique_ptr과 std::shared_ptr의 특성 사이에는 실제로 절충점이 없습니다. Widget과 같은 클래스와 Widget::Impl과 같은 클래스 간의 관계는 독점적 소유권이고 std::를 만들기 때문입니다. unique_ptr 작업에 적합한 도구입니다. 그럼에도 불구하고 공유 소유권이 존재하는 상황(따라서 std::shared_ptr이 적합한 설계 선택임)과 같은 다른 상황에서는 std::unique_ptr 사용이 수반하는 기능 정의 후프를 건너뛸 필요가 없다는 점을 아는 것이 좋습니다.

기억할 사항 • Pimpl

Idiom은 컴파일 종속성을 줄여 빌드 시간을 단축합니다.

클래스 클라이언트와 클래스 구현 사이.

- std::unique_ptr pImpl 포인터의 경우 클래스 헤더에 특수 멤버 함수를 선언하지만 구현 파일에서는 구현합니다. 기본 함수 구현이 허용되는 경우에도 이 작업을 수행합니다.
- 위의 조언은 std::unique_ptr에 적용되지만 std::shared_ptr에는 적용되지 않습니다.

5장

Rvalue 참조, 이동 의미 및 완벽한 전달

그것들에 대해 처음 배울 때 의미 체계를 이동하고 완벽한 전달은 매우 간단해 보입니다.

- 이동 의미 체계를 사용하면 컴파일러에서 비용이 많이 드는 복사 작업을 비용이 덜 드는 이동으로 교체할 수 있습니다. 복사 생성자와 복사 할당 연산자가 객체 복사의 의미를 제어하는 것과 같은 방식으로, 이동 생성자와 이동 할당 연산자는 이동 의미에 대한 제어를 제공합니다. 이동 의미 체계를 사용하면 `std::unique_ptr`, `std::future` 및 `std::thread`와 같은 이동 전용 유형을 생성할 수도 있습니다.
- 완벽한 전달은 임의의 인수를 취하는 함수 템플릿을 작성하고 이를 다른 함수에 전달하여 대상 함수가 전달 함수에 전달된 것과 정확히 동일한 인수를 받도록 하는 것을 가능하게 합니다.

Rvalue 참조는 이 두 가지 서로 다른 기능을 함께 묶는 접착제입니다.

이동 의미 체계와 완벽한 전달을 모두 가능하게 하는 기본 언어 메커니즘이입니다.

이러한 기능에 대한 경험이 많을수록 첫 인상이 속담에 나오는 빙산의 은유적 일각에 불과하다는 것을 더 많이 깨닫게 됩니다. 이동 의미 체계, 완벽한 전달 및 rvalue 참조의 세계는 보이는 것보다 더 미묘한 차이가 있습니다. 예를 들어 `std::move`는 아무 것도 움직이지 않으며 완벽한 전달은 불완전합니다. 이동 작업이 복사보다 항상 저렴하지는 않습니다. 그들이 기대하는 것처럼 항상 저렴하지는 않습니다. 이동이 유효한 컨텍스트에서 항상 호출되는 것은 아닙니다. "type&&" 구문이 항상 rvalue 참조를 나타내는 것은 아닙니다.

이러한 기능을 아무리 자세히 살펴봐도 항상 밝혀야 할 것이 더 많은 것처럼 보일 수 있습니다. 다행히도 깊이에는 한계가 있습니다. 이 장에서는 기반암으로 안내합니다. 일단 도착하면 C++11의 이 부분이 훨씬 더 이해가 될 것입니다. 예를 들어 std::move 및 std::forward의 사용 규칙을 알 수 있습니다. "type&&"의 모호한 특성에 익숙해질 것입니다. 이동 작업의 동작 프로필이 놀라울 정도로 다양한 이유를 이해할 수 있을 것입니다. 모든 조각이 제자리에 떨어질 것입니다. 그 시점에서 이동 의미론, 완벽한 전달 및 rvalue 참조가 다시 한 번 매우 간단해 보이기 때문에 시작했던 곳으로 돌아갈 것입니다. 그러나 이번에는 그대로 있을 것입니다.

이 장의 항목에서 매개변수는 유형이 rvalue 참조인 경우에도 항상 lvalue임을 명심하는 것이 특히 중요합니다. 즉, 주어진

무효 f(위젯&& w);

매개변수 w는 유형이 rvalue-reference-to-Widget임에도 불구하고 lvalue입니다. (이것이 당신을 놀라게 한다면 2 페이지에서 시작하는 lvalue와 rvalue의 개념을 검토하십시오.)

항목 23: std::move 와 std::forward를 이해하라.

하지 않는 일의 관점에서 std::move 및 std::forward에 접근하는 것이 유용합니다. std::move는 아무 것도 움직이지 않습니다. std::forward는 아무 것도 전달하지 않습니다. 런타임 시 둘 다 아무 것도 하지 않습니다. 실행 코드를 생성하지 않습니다. 단일 바이트가 아닙니다.

std::move 및 std::forward는 캐스트를 수행하는 단순한 함수(실제로는 함수 템플릿)입니다. std::move는 해당 인수를 rvalue로 무조건 캐스트하지만 std::forward는 특정 조건이 충족되는 경우에만 이 캐스트를 수행합니다. 그게 다야 설명은 새로운 일련의 질문으로 이어지지만, 근본적으로 그것이 완전한 이야기입니다.

이야기를 더 구체적으로 만들기 위해 다음은 C++11에서 std::move를 구현한 샘플입니다. 표준의 세부 사항을 완전히 따르지는 않지만 매우 가깝습니다.

```
template<typename T> // 네임스페이스 std에서
typename remove_reference<T>::type&&
move(T&& param) {
    반환 유형 사용 = // 별칭 선언; 유형 이름
    remove_reference<T>::유형&&; // 항목 9 참조

    return static_cast<ReturnType>(매개변수);
}
```

코드의 두 부분을 강조 표시했습니다. 하나는 함수의 이름입니다. 반환 유형 사양이 다소 시끄럽고 소음에서 방향을 잃지 않기를 바랍니다. 다른 하나는 기능의 본질을 구성하는 캐스트입니다.

보시다시피 std::move는 객체에 대한 참조(정확히 말하면 범용 참조, 항목 24 참조)를 사용하고 동일한 객체에 대한 참조를 반환합니다.

함수 반환 유형의 "&&" 부분은 std::move가 rvalue 참조를 반환함을 의미하지만 항목 28에서 설명하는 것처럼 T 유형이 lvalue 참조인 경우 T&&는 lvalue 참조가 됩니다. 이를 방지하기 위해 유형 특성(항목 9 참조) std::remove_reference가 T에 적용되어 참조가 아닌 유형에 "&&"가 적용되도록 합니다. 이는 std::move가 진정으로 rvalue 참조를 반환한다는 것을 보장하며, 함수에서 반환된 rvalue 참조가 rvalue이기 때문에 중요합니다. 따라서 std::move는 인수를 rvalue로 캐스팅하고 그게 전부입니다.

제쳐두고, std::move는 C++14에서 덜 소란스럽게 구현할 수 있습니다. 함수 반환 유형 추론(항목 3 참조)과 표준 라이브러리의 별칭 템플릿 std::remove_reference_t(항목 9 참조) 덕분에 std::move를 다음과 같이 작성할 수 있습니다.

```
템플릿<유형이름 T> decltype(자  
동) 이동(T&& 매개변수) {  
  
    ReturnType 사용 = remove_reference_t<T>&&; return  
    static_cast<ReturnType>(매개변수);  
}
```

// C++14; 여전히 // 네임스
페이지 std에 있음

눈이 편하죠?

std::move는 인수를 rvalue로 캐스팅하는 것 외에는 아무것도 하지 않기 때문에 더 나은 이름이 rvalue_cast와 같을 수 있다는 제안이 있었습니다.

우리가 가진 이름은 std::move이므로 std::move가 하는 일과 하지 않는 일을 기억하는 것이 중요합니다. 캐스팅합니다. 그것은 움직이지 않는다.

물론 rvalue는 이동할 수 있는 후보이므로 객체에 std::move를 적용하면 컴파일러에 해당 객체가 이동할 수 있음을 알립니다. 이것이 std::move의 이름이 이동될 수 있는 객체를 쉽게 지정할 수 있도록 하는 이유입니다.

실제로 rvalue는 일반적으로 이동할 수 있는 후보일 뿐입니다. 주석을 나타내는 클래스를 작성한다고 가정합니다. 클래스의 생성자는 주석을 구성하는 std::string 매개변수를 취하고 매개변수를 데이터 멤버에 복사합니다. 항목 41의 정보로 플러시하여 값별 매개변수를 선언합니다.

```
클래스 주석 { 공개: 명시적  
주석(std::문자열 텍스트); //  
복사할 매개변수,
```

```
... // 항목 41에 따라 // 값으로
}; 전달
```

그러나 Annotation의 생성자는 텍스트 값만 읽으면 됩니다. 수정할 필요가 없습니다. 가능하면 항상 const를 사용하는 오랜 전통에 따라 선언을 수정하여 텍스트가 const가 되도록 합니다.

```
class Annotation { public:
    명시적 주석(const
        std::string text)
    ...
};
```

텍스트를 데이터 멤버에 복사할 때 복사 작업에 대한 비용을 지불하지 않으려면 [항목 41](#)의 조언에 충실하고 std::move를 텍스트에 적용하여 rvalue를 생성합니다.

```
class Annotation { public:
    명시적 Annotation(const
        std::string text) : value(std::move(text)) // 텍스트를 값으로
    "이동"합니다. 이 코드 { ... }
    // 보이는 대로 하지 않습니다!
    ...
};
```

개인: 표준::
문자열 값; };

이 코드는 컴파일됩니다. 이 코드는 연결됩니다. 이 코드가 실행됩니다. 이 코드는 데이터 멤버 값을 텍스트 내용으로 설정합니다. 이 코드와 비전의 완벽한 실현을 구분하는 유일한 것은 텍스트가 가치로 옮겨지는 것이 아니라 복사된다는 것입니다. 물론 텍스트는 std::move에 의해 rvalue로 캐스트되지만 텍스트는 const std::string으로 선언되므로 캐스트 전에 text는 lvalue const std::string이고 캐스트 결과는 rvalue const std::string, 그러나 그 전체에 걸쳐 constness는 남아 있습니다.

컴파일러가 호출할 std::string 생성자를 결정해야 할 때 미치는 영향을 고려하십시오. 두 가지 가능성 이 있습니다.

```
클래스 문자열 { 공개:
    문자열(const 문자열 및 rhs); 문자열(문자열&& rhs); // ctor 복사 //
    ...
}; // ctor 이동
```

주석 생성자의 멤버 초기화 목록에서 std::move(text)의 결과는 const std::string 유형의 rvalue입니다. 이동 생성자는 비 const std::string에 대한 rvalue 참조를 사용하기 때문에 해당 rvalue는 std::string의 이동 생성자에 전달할 수 없습니다. 그러나 rvalue는 const rvalue에 바인딩할 수 있는 lvalue-reference-to-const가 허용되기 때문에 복사 생성자에 전달할 수 있습니다. 따라서 멤버 초기화는 텍스트가 rvalue로 캐스팅된 경우에도 std::string의 복사 생성자를 호출합니다! 이러한 동작은 const-correctness를 유지하는 데 필수적입니다. 개체에서 값을 이동하면 일반적으로 개체가 수정되므로 언어는 const 개체를 수정할 수 있는 함수(예: 이동 생성자)에 전달되는 것을 허용하지 않아야 합니다.

이 예에서 두 가지 교훈을 얻을 수 있습니다. 첫째, 객체로부터 이동할 수 있도록 하려면 객체를 const로 선언하지 마십시오. const 객체에 대한 이동 요청은 자동으로 복사 작업으로 변환됩니다. 둘째, std::move는 실제로 아무 것도 이동하지 않을 뿐만 아니라 캐스팅 중인 개체가 이동할 수 있다는 보장도 없습니다. 객체에 std::move를 적용한 결과에 대해 확실히 알고 있는 것은 그것이 rvalue라는 것뿐입니다.

std::forward의 스토리는 std::move의 스토리와 비슷하지만 std::move는 인수를 rvalue로 무조건 캐스팅하지만 std::forward는 특정 조건에서만 수행합니다. std::forward는 조건부 캐스트입니다. 캐스팅할 때와 캐스팅하지 않을 때를 이해하려면 일반적으로 std::forward가 어떻게 사용되는지 기억하십시오. 가장 일반적인 시나리오는 다른 함수에 전달될 범용 참조 매개변수를 사용하는 함수 템플릿입니다.

```
무효 프로세스(const Widget& lvalArg); 무효 프로세스 // lvalue 처리 // rvalue 처리
(Widget&& rvalArg);

템플릿<유형 이름 T> 무효 // param을 프로세스에 전달하는 템플릿
logAndProcess(T& 매개변수) {

    지금 자동 = // 현재 시간 가져오기
        std::chrono::system_clock::now();

    makeLogEntry("프로세스' 호출", 지금); 프로세스
        (std::forward<T>(param));
}

logAndProcess에 대한 두 번의 호출을 고려하십시오. 하나는 lvalue이고 다른 하나는 rvalue입니다.
```

```
위젯 w;
로그앤프로세스(w); // lvalue로 호출 // rvalue로 호출
logAndProcess(표준::이동(w));
```

`logAndProcess` 내부에서 매개변수 `param`이 함수 프로세스에 전달됩니다. 프로세스는 `lvalue` 및 `rvalue`에 대해 오버로드됩니다. `lvalue`로 `logAndProcess`를 호출하면 자연스럽게 `lvalue`가 프로세스에 `lvalue`로 전달될 것으로 예상하고 `rvalue`로 `logAndProcess`를 호출할 때 프로세스의 `rvalue` 오버로드가 호출될 것으로 예상합니다.

그러나 `param`은 모든 함수 매개변수와 마찬가지로 `lvalue`입니다. 따라서 `logAndProcess` 내부의 프로세스에 대한 모든 호출은 프로세스에 대한 `lvalue` 오버로드를 호출하기를 원할 것입니다. 이를 방지하려면 `param`이 초기화된 인수(`logAndProcess`에 전달된 인수)가 `rvalue`인 경우에만 `param`을 `rvalue`로 캐스팅하는 메커니즘이 필요합니다. 이것이 바로 `std::forward`가 하는 일입니다. 이것이 `std::forward`가 조건부 캐스트인 이유입니다. 인수가 `rvalue`로 초기화된 경우에만 `rvalue`로 캐스트됩니다.

`std::forward`가 인수가 `rvalue`로 초기화되었는지 여부를 어떻게 알 수 있는지 궁금할 것입니다. 예를 들어 위의 코드에서 `std::forward`는 `param`이 `lvalue` 또는 `rvalue`로 초기화되었는지 여부를 어떻게 알 수 있습니까? 간단한 대답은 정보가 `logAndProcess`의 템플릿 매개변수 `T`로 인코딩된다는 것입니다. 이 매개변수는 인코딩된 정보를 복구하는 `std::forward`로 전달됩니다. 정확한 작동 방식에 대한 자세한 내용은 [항목 28을 참조하세요](#).

`std::move`와 `std::forward`는 모두 캐스트로 축소되고 유일한 차이점은 `std::move`가 항상 캐스트하는 반면 `std::forward`는 때때로 캐스트한다는 점을 감안할 때 `std`를 생략할 수 있는지 여부를 묻습니다. :이동하고 어디에서나 `std::forward`를 사용하십시오. 순전히 기술적인 관점에서 대답은 예입니다. `std::forward`는 모든 것을 할 수 있습니다. `std::move`는 필요하지 않습니다. 물론 캐스트를 어디에서나 작성할 수 있기 때문에 두 기능 모두 실제로 필요한 것은 아니지만, 우리가 동의하기를 바랍니다.

`std::move`의 매력은 편의성, 오류 가능성 감소 및 더 큰 명확성입니다. 이동 생성자가 호출된 횟수를 추적하려는 클래스를 고려하십시오. 이동 구성 중에 증가하는 정적 카운터만 있으면 됩니다. 클래스의 유일한 비정적 데이터가 `std::string`이라고 가정하면 다음은 이동 생성자를 구현하는 일반적인 방법(예: `std::move` 사용)입니다.

```
클래스 위젯 { 공개:
    위젯(위젯& rhs) :
        s(std::move(rhs.s)) { +
            +moveCtorCalls; }
```

...

개인: 정적

```
std::size_t moveCtorCalls; 표준::문자열 s; };
```

std::forward로 동일한 동작을 구현하려면 코드는 다음과 같습니다.

```
클래스 위젯 { 공개: 위
젯(위젯&& rhs) :
    ...
};
```

// 비관습적, // 바람직하지
않은 // 구현

```
s(std::forward<std::string>(rhs.s)) { +
    +moveCtorCalls; }
```

먼저 std::move에는 함수 인수(rhs.s)만 필요한 반면 std::forward에는 함수 인수(rhs.s)와 템플릿 유형 인수(std::string)가 모두 필요합니다. 그런 다음 std::forward에 전달하는 유형은 참조가 아니어야 합니다. 전달되는 인수가 rvalue라는 인코딩 규칙이기 때문입니다(항목 28 참조). 함께 이것은 std::move가 std::forward보다 타이핑이 덜 필요하고 우리가 전달하는 인수가 rvalue임을 인코딩하는 유형 인수를 전달하는 수고를 덜어준다는 것을 의미합니다. 또한 잘못된 유형을 전달할 가능성도 제거합니다(예: std::string&, 데이터 멤버가 이동 생성 대신 복사 생성됨).

더 중요한 것은 std::move를 사용하면 rvalue에 대한 무조건 캐스트가 전달되는 반면 std::forward를 사용하면 rvalue가 바인딩된 참조에 대해서만 rvalue로 캐스트된다는 것입니다. 아주 다른 두 가지 행동입니다. 첫 번째 함수는 일반적으로 이동을 설정하는 반면 두 번째 함수는 원래의 lvaluefulness 또는 rvaluefulness를 유지하는 방식으로 객체를 다른 함수에 전달합니다. 이러한 작업은 매우 다르기 때문에 이를 구별하기 위해 두 개의 다른 함수(및 함수 이름)가 있는 것이 좋습니다.

기억해야 할 사항

- std::move는 rvalue에 대한 무조건 캐스트를 수행합니다. 그 자체로는 아무것도 움직이지 않습니다.
- std::forward는 해당 인수가 바인딩된 경우에만 해당 인수를 rvalue로 캐스팅합니다. rvalue로.
- std::move나 std::forward는 런타임에 아무 것도 하지 않습니다.

항목 24: 범용 참조를 rvalue 참조와 구별하십시오.

진실이 너희를 자유케 하리라고 말하지만, 올바른 상황에서 잘 선택된 거짓말도 똑같이 자유를 줄 수 있다. 이 항목은 그런 거짓말입니다. 그러나 우리는 소프트웨어를 다루고 있기 때문에 "거짓 말"이라는 단어를 피하고 대신 이 항목이 "추상화"로 구성되어 있다고 말합시다.

일부 유형 T에 대한 rvalue 참조를 선언하려면 T&&를 작성합니다. 따라서 소스 코드에서 "T&&"가 표시되면 rvalue 참조를 보고 있다고 가정하는 것이 합리적으로 보입니다. 아아, 그렇게 간단하지 않습니다.

무효 f(위젯&& 매개변수);	// rvalue 참조
위젯&& var1 = 위젯();	// rvalue 참조
자동&& var2 = var1;	// rvalue 참조 가 아님
template<typename T> 무 효 f(std::vector<T>&& param);	// rvalue 참조
template<typename T> 무 효 f(T&& 매개변수);	// rvalue 참조 가 아님

사실, "T&&"는 두 가지 다른 의미를 가지고 있습니다. 하나는 물론 rvalue 참조입니다. 이러한 참조는 예상한 대로 정확히 작동합니다. 참조는 rvalue에만 바인딩되며 주된 이유는 이동할 수 있는 개체를 식별하는 것입니다.

"T&&"의 다른 의미는 rvalue 참조 또는 lvalue 참조입니다. 이러한 참조는 소스 코드에서 rvalue 참조(즉, "T&&")처럼 보이지만 lvalue 참조(즉, "T&")인 것처럼 동작할 수 있습니다. 이들의 이중 특성으로 인해 rvalue(rvalue 참조와 같은) 및 lvalue(lvalue 참조와 같은)에 바인딩할 수 있습니다. 게다가 그들은 const 또는 non-const 객체, volatile 또는 non-volatile 객체, 심지어 const와 volatile 둘 다인 객체에도 바인딩할 수 있습니다. 그들은 거의 모든 것에 바인딩할 수 있습니다. 이러한 전례 없이 유연한 참조는 고유한 이름을 가질 가치가 있습니다. 나는 그것들을 보편적인 참조라고 부른다. 1 보편적 참조는 두 가지 상황에서 발생합니다. 가장 일반적인 것은 위 샘플 코드의 이 예와 같은 함수 템플릿 매개변수입니다.

1 항목 25 는 범용 참조에 거의 항상 std::forward가 적용되어야 한다고 설명하며 이 책이 출판되면서 C++ 커뮤니티의 일부 구성원은 범용 참조를 전달 참조로 참조하기 시작했습니다.

```
template<typename T> 무
효 f(T&& 매개변수); // param은 범용 참조입니다.
```

두 번째 컨텍스트는 위의 샘플 코드에서 가져온 것을 포함하는 자동 선언입니다.

```
자동&& var2 = var1; // var2는 범용 참조입니다.
```

이러한 컨텍스트의 공통점은 유형 추론이 있다는 것입니다. 템플릿 f에서 param의 유형이 추론되고 var2에 대한 선언에서 var2의 유형이 추론됩니다. 이를 유형 추론이 누락된 다음 예제(위의 샘플 코드에서도 참조)와 비교하십시오. 유형 추론 없이 "T&&"가 표시되면 rvalue 참조를 보고 있는 것입니다.

```
무효 f(위젯&& 매개변수); // 유형 추론 없음; // param은
rvalue 참조입니다.
```

```
위젯&& var1 = 위젯(); // 유형 추론 없음; // var1은
rvalue 참조입니다.
```

범용 참조는 참조이므로 초기화해야 합니다. 범용 참조의 이니셜라이저는 rvalue 참조를 나타내는지 아니면 lvalue 참조를 나타내는지 여부를 결정합니다. 이니셜라이저가 rvalue이면 범용 참조는 rvalue 참조에 해당합니다. 이니셜라이저가 lvalue이면 범용 참조는 lvalue 참조에 해당합니다. 함수 매개변수인 범용 참조의 경우 이니셜라이저는 호출 사이트에서 제공됩니다.

```
template<typename T> 무
효 f(T&& 매개변수); // param은 범용 참조입니다.
```

```
위젯 w; f(w); // f에 전달된 lvalue; param의 유형은 // Widget&입니다(즉,
lvalue 참조).
```

```
f(표준::이동(w)); // f에 전달된 rvalue; param의 유형은 // Widget&&입니다
(즉, rvalue 참조).
```

참조가 보편화되기 위해서는 형식 연역이 필요하지만 충분하지 않습니다. 참조 선언의 형식도 정확해야 하며 그 형식은 상당히 제한적입니다. 정확히 "T&&"여야 합니다. 이전에 본 샘플 코드에서 이 예제를 다시 살펴보세요.

```
template<typename T> 무효
f(std::vector<T>&& param); // param은 rvalue 참조입니다.
```

f가 호출될 때 유형 T가 추론될 것입니다(호출자가 명시적으로 지정하지 않는 한, 우리가 신경 쓰지 않을 극단적인 경우). 그러나 param의 유형 선언 형식은-

tion은 "T&&"가 아니라 "std::vector<T>&&"입니다. 이는 param이 보편적인 참조일 가능성을 배제합니다. 따라서 param은 rvalue 참조이며, lvalue를 f에 전달하려고 하면 컴파일러에서 기꺼이 확인할 수 있습니다.

```
표준::벡터<int> v; f(v);
// 오류! lvalue를 // rvalue 참조에 바인딩할
// 수 없습니다.
```

const 한정자의 단순한 존재만으로도 참조가 보편성에서 제외되기에 충분합니다.

```
template<typename T> 무
효 f(const T&& 매개변수); // param은 rvalue 참조입니다.
```

템플릿에 있고 "T&&" 유형의 함수 매개변수가 표시되면 이것이 범용 참조라고 가정할 수 있습니다. 당신은 할 수 없습니다. 템플릿에 있다고 해서 형식 추론이 보장되지 않기 때문입니다. std::vector에서 다음 push_back 멤버 함수를 고려하십시오.

```
template<class T, class Allocator = allocator<T>> // C++에서 클래스 벡터 {
// 표준 공개: void
push_back(T&& x);

...
};
```

push_back의 매개변수는 확실히 범용 참조에 대한 올바른 형식을 가지고 있지만 이 경우에는 형식 추론이 없습니다. 이는 push_back이 포함될 특정 벡터 인스턴스화 없이는 존재할 수 없으며 해당 인스턴스화 유형이 push_back에 대한 선언을 완전히 결정하기 때문입니다. 즉, 말하는

```
std::vector<위젯> v;
std::vector 템플릿이 다음과 같이 인스턴스화되도록 합니다.
```

```
클래스 벡터<위젯, 할당자<위젯>> { 공개: 무효 push_back(위젯
&& x);
// rvalue 참조
...
};
```

이제 push_back이 유형 추론을 사용하지 않는다는 것을 분명히 알 수 있습니다. vector<T>에 대한 이 push_back(2개가 있습니다. 함수가 오버로드됨)은 항상 rvalue-reference-to-T 유형의 매개변수를 선언합니다.

대조적으로 std::vector의 개념적으로 유사한 emplace_back 멤버 함수는 형식 추론을 사용합니다.

```

template<class T, class Allocator = allocator<T>> // 여전히 클래스 벡터 { public: template
<class... Args> void emplace_back(Args&&... args); // C++
// 표준
...
};

...

```

여기에서 유형 매개변수 Args는 벡터의 유형 매개변수 T와 무관하므로 emplace_back이 호출될 때마다 Args를 추론해야 합니다. (알겠습니다. Args는 실제로 유형 매개변수가 아니라 매개변수 팩이지만 이 논의의 목적을 위해 마치 유형 매개변수인 것처럼 취급할 수 있습니다.)

emplace_back의 유형 매개변수의 이름이 Args이지만 여전히 보편적인 참조라는 사실은 이것이 "T&&"여야 하는 보편적인 참조의 형태라는 이전 의견을 강화합니다. T라는 이름을 사용할 필요는 없습니다. 예를 들어, 다음 템플릿은 형식 ("type&&") 이 올바르고 param의 유형이 추론 되기 때문에 범용 참조를 사용합니다 (호출자가 명시적으로 지정하는 코너 케이스 제외 유형):

```

템플릿<유형명 MyTemplateType> 무효 // param은 // 범용
someFunc(MyTemplateType&& 매개변수); 참조입니다.

```

자동 변수도 범용 참조가 될 수 있다고 앞서 언급했습니다. 더 많은

정확하고 auto&& 유형으로 선언된 변수는 범용 참조입니다. 유형 추론이 발생하고 올바른 형식("T&&")을 갖기 때문입니다. 자동 범용 참조는 함수 템플릿 매개변수에 사용되는 범용 참조만큼 일반적이지는 않지만 C++11에서 때때로 발생합니다. C++14 람다 표현식이 auto&& 매개변수를 선언할 수 있기 때문에 C++14에서 더 많이 발생합니다. 예를 들어, 임의의 함수 호출에 소요된 시간을 기록하기 위해 C++14 람다를 작성하려면 다음과 같이 할 수 있습니다.

```

자동 timeFuncInvocation = [](자동
&& 가능, 자동&&... 매개변수) {
    시작 타이머;
    std::forward<decltype(func)>(func) // 함수 호출
    std::forward<decltype(params)...>(params)... // on params );
    타이머를 종지하고 경과 시간을 기록하십시오. };

```

람다 내부의 "std::forward<decltype(blah blah blah)>" 코드에 대한 반응이 "뭐...?!"라면 이는 아마도 항목 33을 아직 읽지 않았다는 의미일 것입니다.

걱정하지 마세요. 이 항목에서 중요한 것은 auto&& 매개변수입니다.

람다가 선언합니다. func는 호출 가능한 모든 개체, lvalue 또는 rvalue에 바인딩할 수 있는 범용 참조입니다. args는 임의 유형의 객체에 바인딩할 수 있는 0개 이상의 범용 참조(즉, 범용 참조 매개변수 팩)입니다. 자동 범용 참조 덕분에 결과는 timeFuncInvocation이 거의 모든 함수 실행 시간을 측정 할 수 있다는 것입니다. (“임의”와 “대부분의”의 차이점에 대한 정보는 항목 30을 참조하십시오.)

이 전체 항목(보편 참조의 기초)은 거짓말...어, “추상화”라는 것을 명심하십시오. 기본 진실은 항목 28 이 현정 된 주제인 참조 축소로 알려져 있습니다 . 그러나 진실은 추상화를 덜 유용하게 만들지 않습니다.

rvalue 참조와 범용 참조를 구분하면 소스 코드를 더 정확하게 읽는 데 도움이 되며("그 T&&는 rvalue에만 바인딩합니까 아니면 모든 것에 바인딩합니까?") 동료와 통신할 때 모호성을 피할 수 있습니다("여기서는 rvalue 참조가 아닌 범용 참조를 사용하고 있습니다..."). 또한 구분에 의존하는 항목 25 와 26 을 이해할 수 있습니다 . 따라서 추상화를 수용하십시오. 그것을 즐기십시오. 뉴턴의 운동 법칙(기술적으로 올바르지 않음)이 일반적으로 아인슈타인의 일반 상대성 이론("진리")만큼 유용하고 적용하기 쉬운 것처럼 보편적 참조 개념도 일반적으로 작업보다 선호됩니다. 참조 축소의 세부 사항을 통해.

기억해야 할 사항

- 함수 템플릿 매개변수가 추론된 유형 T에 대해 유형 T&&를 가지거나 객체가 auto&&를 사용하여 선언된 경우 매개변수 또는 객체는 범용 참조입니다.
엔스.
- 타입 선언의 형태가 정확히 type&& 가 아니거나 타입 추론이 일어나지 않는 경우 type&& 는 rvalue 참조를 나타낸다. • 범용 참조는 rvalue로 초기화된 경우 rvalue 참조에 해당합니다. lval로 초기화된 경우 lvalue 참조에 해당합니다.
화.

항목 25: rvalue 참조에 std::move 를 사용하세요. std:: 보편 참조에서 전달합니다.

Rvalue 참조는 이동할 수 있는 대상에만 바인딩됩니다. rvalue 참조 매개변수가 있는 경우 바인딩된 개체가 이동할 수 있음을 알 수 있습니다.

```
클래스 위젯 {
    위젯(위젯&& rhs); // rhs 는 확실히 참조
```

```
... // 이동할 수 있는 객체
};
```

이 경우 해당 함수가 객체의 rvalue를 활용할 수 있도록 이러한 객체를 다른 함수에 전달하고 싶을 것입니다. 그렇게 하는 방법은 그러한 객체에 바인딩된 매개변수를 rvalue로 캐스팅하는 것입니다. [항목 23](#)에서 설명하는 것처럼 std::move가 하는 일 뿐만 아니라 다음을 위해 생성되었습니다.

```
클래스 위젯 { 공개:
Widget(Widget&&
rhs) : 이름
{
    std::move(rhs.name)),
    p(std::move(rhs.p)) { ... }
}
```

```
개인: 표준::
문자열 이름;
std::shared_ptr<SomeDataStructure> p;};
```

반면에 범용 참조([항목 24](#) 참조)는 이동할 수 있는 객체에 바인딩될 수 있습니다. 범용 참조는 rvalue로 초기화된 경우에만 rvalue로 캐스트되어야 합니다. [항목 23](#)은 이것이 정확히 std::forward가 하는 일이라고 설명합니다.

```
클래스 위젯 { 공개: 템
플릿<유형 이름 T> 무
    효 setName(T&& newName)
    { 이름 = std::forward<T>(newName); } // newName은 //
                                            // 범용 참조입니다.
}
...;
```

간단히 말해서, rvalue 참조는 항상 rvalue에 바인딩되어 있기 때문에 다른 함수로 전달할 때 rvalue로 무조건 캐스트되어야 합니다(std::move를 통해). 범용 참조는 rvalue로 조건부로 캐스트되어야 합니다(std::forward를 통해) 전달할 때 가끔 rvalue에만 바인딩되기 때문입니다.

[항목 23](#)에서는 rvalue 참조에 std::forward를 사용하면 적절한 동작을 나타낼 수 있지만 소스 코드는 장황하고 오류가 발생하기 쉽고 단조로우므로 rvalue 참조와 함께 std::forward를 사용하는 것을 피해야 한다고 설명합니다. 더 나쁜 것은 유니버설 참조와 함께 std::move를 사용하는 아이디어입니다. 왜냐하면 예기치 않게 lvalue(예: 지역 변수)를 수정하는 효과가 있을 수 있기 때문입니다.

```
클래스 위젯 {
    공공의:
        템플릿<유형이름 T>
        무효 setName(T&& newName) { 이
            름 = std::move(newName); }           // 범용 참조
            ...                                // 컴파일하지만
                                            // 나쁘다, 나쁘다, 나쁘다!
```

사적인:

```
    std::문자열 이름;
    std::shared_ptr<SomeDataStructure> p;
};

표준::문자열 getWidgetName();           // 팩토리 함수

위젯 w;

자동 n = getWidgetName();                // n은 지역 변수입니다.

w.setName(n);                          // n을 w로 이동！

...
                                            // n의 값은 이제 알 수 없음
```

여기서 지역 변수 n은 호출자가 용서받을 수 있는 w.setName에 전달됩니다.

n에 대한 읽기 전용 작업이라고 가정합니다. 그러나 setName이 내부적으로 사용하기 때문에 std::move는 참조 매개변수를 rvalue로 무조건 캐스트합니다. n의 값은 w.name으로 이동하고 n은 지정되지 않은 setName에 대한 호출에서 돌아옵니다. 정의된 가치. 그것은 전화를 건 사람을 절망에 빠뜨릴 수 있는 종류의 행동입니다. 폭행.

setName이 매개변수를 유니버설로 선언해서는 안 된다고 주장할 수 있습니다.

살 참조. 이러한 참조는 const가 될 수 없지만(항목 24 참조) setName은 확실합니다.

매개변수를 수정하면 안 됩니다. setName이 단순히

const lvalue 및 rvalue에 대해 오버로드된 경우 전체 문제가 피했다. 이와 같이:

```
클래스 위젯 {
    공공의:
        무효 setName(const std::string& newName) { 이름 =
            newName; }                         //에서 설정
                                                // const lvalue

        무효 setName(std::string&& newName) { 이름 =
            std::move(newName); }             //에서 설정
                                                // r값

        ...
};

};
```

이 경우에는 확실히 작동하지만 단점이 있습니다. 첫째, 작성하고 유지해야 할 소스 코드가 더 많습니다(단일 템플릿 대신 두 가지 기능). 둘째, 효율성이 떨어질 수 있습니다. 예를 들어 다음과 같은 `setName` 사용을 고려하십시오.

```
w.setName("아델라 노박");
```

`setName` 버전이 범용 참조를 사용하면 문자열 리터럴 "Adela Novak"이 `setName`으로 전달되고 여기서 `w` 내부의 `std::string`에 대한 할당 연산자로 전달됩니다. 따라서 `w`의 이름 데이터 멤버는 문자열 리터럴에서 직접 할당됩니다. 임시 `std::string` 개체가 발생하지 않습니다. 그러나 `setName`의 오버로드된 버전을 사용하면 `setName`의 매개변수가 바인딩할 임시 `std::string` 개체가 생성되고 이 임시 `std::string`이 `w`의 데이터 멤버로 이동됩니다. 따라서 `setName`에 대한 호출은 하나의 `std::string` 생성자(임시 생성), 하나의 `std::string` 이동 할당 연산자(`newName`을 `w.name`으로 이동), 하나의 `std::string` 소멸자(파괴 임시). 이는 `const char*` 포인터를 사용하는 `std::string` 할당 연산자만 호출하는 것보다 확실히 더 비싼 실행 시퀀스입니다. 추가 비용은 구현마다 다를 수 있으며 해당 비용을 걱정할 가치가 있는지 여부는 응용 프로그램마다, 라이브러리마다 다르지만 사실은 범용 참조를 사용하는 템플릿을 쌍으로 교체한다는 것입니다. `lvalue` 참조 및 `rvalue` 참조에 오버로드된 함수로 인해 경우에 따라 런타임 비용이 발생할 수 있습니다.

위젯의 데이터 멤버가 `std::string`이라는 것을 아는 것이 아니라 임의의 유형이 될 수 있도록 예제를 일 반화하면 모든 유형이 `std::string`([항목 29](#) 참조).

그러나 `lvalue` 및 `rvalue`에 대한 오버로딩의 가장 심각한 문제는 소스 코드의 볼륨이나 관용성이 아 니며 코드의 런타임 성능도 아닙니다.

그것은 디자인의 열악한 확장성입니다. `Widget::setName`은 매개변수를 하나만 사용하므로 두 개의 오버로드만 필요하지만 더 많은 매개변수를 사용하는 함수(각각 `lvalue` 또는 `rvalue`가 될 수 있음)의 경우 오버로드 수가 기하학적으로 증가합니다. n 매개변수는 $2n$ 오버로드가 필요합니다. 그리고 그 것이 최악은 아닙니다. 일부 함수(실제로는 함수 템플릿)는 매개변수를 무제한으로 사용하며, 각 매개 변수는 `lvalue` 또는 `rvalue`가 될 수 있습니다. 이러한 함수의 포스터 자식은 `std::make_shared`이고 C++14부터 `std::make_unique`입니다([항목 21](#) 참조). 가장 일반적으로 사용되는 오버로드 선언을 확인하세요.

```
template<class T, class... Args> // C++11에서
shared_ptr<T> make_shared(Args&&... args); // 기준

template<class T, class... Args> // C++14에서
unique_ptr<T> make_unique(Args&&... args); // 기준
```

이와 같은 함수의 경우 lvalue 및 rvalue에 대한 오버로드는 옵션이 아닙니다. 범용 참조가 유일한 방법입니다. 그리고 그러한 함수 내에서 std::forward는 다른 함수로 전달될 때 범용 참조 매개변수에 적용됩니다. 정확히 해야 할 일입니다.

글쎄, 일반적으로. 결국. 그러나 처음에는 반드시 그런 것은 아닙니다. 어떤 경우에는 단일 함수에서 rvalue 참조 또는 범용 참조에 바인딩된 개체를 두 번 이상 사용하고 싶고 다른 작업을 완료할 때까지 개체가 이동되지 않았는지 확인하고 싶을 것입니다. . 이 경우 참조의 최종 사용에만 std::move(rvalue 참조의 경우) 또는 std::forward(범용 참조의 경우)를 적용할 수 있습니다. 예를 들어:

```
템플릿<유형 이름 T> 무효 // 텍스트는 //
setSignText(T&& 텍스트) { 대학입니다. 참조

    기호.setText(텍스트); // 텍스트를 사용하지만 //
                           수정하지 않음

    자동 지금 = // 현재 시간 가져오기
        std::chrono::system_clock::now();

    signHistory.add(자금,
                    std::forward<T>(텍스트)); // 조건부로 캐스트 // 텍스트를 rvalue로
    }
}
```

여기서 우리는 signHistory.add를 호출할 때 그 값을 사용하기를 원하기 때문에 텍스트의 값이 sign.setText에 의해 변경되지 않도록 하고 싶습니다. 따라서 범용 참조의 최종 사용에만 std::forward를 사용합니다.

std::move의 경우에도 동일한 생각이 적용되지만(즉, 마지막으로 사용된 rvalue 참조에 std::move를 적용), 드문 경우지만 std::를 호출하고 싶을 것입니다. std::move 대신 move_if_noexcept를 사용합니다. 시기와 이유를 알아보려면 항목 14를 참조하십시오.

값으로 반환하는 함수에 있고 rvalue 참조 또는 범용 참조에 바인딩된 개체를 반환하는 경우 참조를 반환할 때 std::move 또는 std::forward를 적용하고 싶을 것입니다. 그 이유를 알아보려면 왼쪽 행렬이 rvalue인 것으로 알려진 두 행렬을 함께 추가하는 operator+ 함수를 고려하십시오(따라서 행렬의 합을 유지하기 위해 저장소를 재사용할 수 있음).

행렬 연산	// 값별 반환
자+(행렬&& lhs, const 행렬& rhs) { lhs += rhs;	

```

    반환 표준::이동(lhs);           // lhs를 다음으로 이동
}                                // 반환 값

```

`lhs`를 `return` 문에서 `rvalue`로 캐스팅하면(`std::move`를 통해) `lhs`는
함수의 반환 값 위치로 이동했습니다. `std::move`에 대한 호출이 생략된 경우
테드,

```

행렬                           // 위와 같이
연산자+(행렬&& lhs, const 행렬& rhs)
{
    lhs += rhs;
    반환 lhs;                  // lhs를 복사
}                                // 반환 값

```

`lhs`가 `lvalue`라는 사실은 컴파일러가 대신 그것을 반환값에 복사하도록 강제할 것입니다.
가치 위치. `Matrix` 유형이 이동 구성을 지원한다고 가정합니다.
`return` 문에서 `std::move`를 사용하여 복사 생성보다 더 효율적입니다.
더 효율적인 코드를 생성합니다.

`Matrix`가 이동을 지원하지 않는 경우 `rvalue`로 캐스팅해도 문제가 되지 않습니다.
`rvalue`는 단순히 `Matrix`의 복사 생성자에 의해 복사됩니다([항목 23 참조](#)). 매트릭스가
나중에 이동을 지원하도록 수정되었으며, `operator+`는 다음에 자동으로 혜택을 받을 것입니다.
컴파일됩니다. 그렇기 때문에 잃을 것은 없습니다(그리고 아마도
획득) 함수에서 반환되는 `rvalue` 참조에 `std::move`를 적용하여
값으로 반환합니다.

상황은 범용 참조 및 `std::forward`와 유사합니다. 기능을 고려하십시오-
축소되지 않은 `Fraction` 개체를 사용하는 템플릿 `reduceAndCopy`,
축소한 다음 축소된 값의 복사본을 반환합니다. 원래 개체가
`rvalue`, 그 값은 반환 값으로 이동되어야 합니다(따라서
복사본 만들기), 그러나 원본이 `lvalue`인 경우 실제 복사본을 만들어야 합니다.
따라서:

```

템플릿<유형이름 T>
분수                           // 값별 반환
reduceAndCopy(T&& frac) {      // 범용 참조 매개변수
    frac.reduce();
    반환 std::forward<T>(frac); // rvalue를 리턴으로 이동
}                                // 값, lvalue 복사

```

`std::forward`에 대한 호출이 생략되면 `frac`은 무조건 복사됩니다.
`reduceAndCopy`의 반환 값.

일부 프로그래머는 위의 정보를 사용하여 상황으로 확장하려고 합니다.
적용되지 않는 곳입니다. "rvalue 참조 매개변수에서 `std::move`를 사용하는 경우

반환 값으로 복사하면 복사 구성이 이동 구성으로 바뀝니다."라고 그들은 "내가 반환하는 로컬 변수에 대해 동일한 최적화를 수행할 수 있습니다."라고 말합니다.

다시 말해서, 그들은 다음과 같이 값으로 지역 변수를 반환하는 함수가 주어졌을 때,

```
위젯 makeWidget() { // makeWidget의 "복사" 버전
    위젯 w; // 지역 변수
    ...
    반환 w; // w를 반환 값으로 "복사"
}
```

"복사"를 이동으로 전환하여 "최적화"할 수 있습니다.

```
위젯 makeWidget() { // makeWidget의 이동 버전
    위젯 w;
    ...
    반환 표준::이동(w); // w를 반환 값으로 이동 // (이 작업을 수행하지 마십시오!)
}
```

인용 부호를 자유롭게 사용한 것은 이러한 추론에 결함이 있음을 알려줄 것입니다. 그런데 왜 결함이 있습니까?

이런 종류의 최적화에 관해서는 표준화 위원회가 그러한 프로그래머보다 훨씬 앞서 있기 때문에 결함이 있습니다. makeWidget의 "복사" 버전은 함수의 반환 값에 할당된 메모리에 구성함으로써 지역 변수 w를 복사할 필요를 피할 수 있다는 것이 오래 전에 인식되었습니다. 이것은 반환 값 최적화(RVO)로 알려져 있으며, C++ 표준이 존재하는 한 명시적으로 축복받았습니다.

그러한 축복을 표현하는 것은 까다로운 일입니다. 왜냐하면 당신은 그러한 복사 생략이 소프트웨어의 관찰 가능한 동작에 영향을 미치지 않는 곳에서만 하용하기를 원하기 때문입니다.

표준의 윤법주의적(아마도 유독한) 산문을 바꾸어 말하면, 이 특별한 축복은 컴파일러가 (1) 로컬 객체의 유형이 동일한 경우 값으로 반환되는 함수에서 로컬 객체²의 복사(또는 이동)를 생략할 수 있다고 말합니다. 함수에 의해 반환되는 것과 (2) 로컬 객체가 반환되는 것입니다. 이를 염두에 두고 makeWidget의 "복사" 버전을 다시 살펴보십시오.

² 적격 지역 객체에는 대부분의 지역 변수(예: makeWidget 내부의 w)와 return 문의 일부로 생성된 임시 객체가 포함됩니다. 함수 매개변수가 적합하지 않습니다. 어떤 사람들은 RVO를 명령된 로컬 객체와 명명되지 않은(즉, 임시) 로컬 객체에 적용하는 것을 구별하여 RVO라는 용어를 명명되지 않은 객체로 제한하고 해당 애플리케이션을 명명된 객체에 대해 NRVO(반환 값 최적화)라고 부릅니다.

```

위젯 makeWidget() {
    위젯 w;
    ...
    반환 w; // w를 반환 값으로 "복사"
}

```

두 조건 모두 여기에서 충족되며 이 코드에 대해 모든 랜찮은 C++ 컴파일러가 w를 복사하는 것을 피하기 위해 RVO를 사용한다고 말할 때 저를 믿으셔도 됩니다. 즉, makeWidget의 "복사" 버전은 실제로 아무 것도 복사하지 않습니다.

makeWidget의 이동 버전은 이름이 하는 대로 수행합니다(Widget이 이동 생성자를 제공한다고 가정). w의 내용을 makeWidget의 반환 값 위치로 이동합니다. 그러나 컴파일러는 RVO를 사용하여 이동을 제거하고 함수의 반환 값에 할당된 메모리에 다시 w를 생성하지 않는 이유는 무엇입니까? 대답은 간단합니다. 그들은 할 수 없습니다. 조건 (2)는 반환되는 것이 로컬 객체인 경우에만 RVO가 수행될 수 있다고 규정하지만, 이는 makeWidget의 이동 버전이 수행하는 작업이 아닙니다. return 문을 다시 살펴보십시오.

반환 표준::이동(w);

여기서 반환되는 것은 로컬 객체 w가 아니라 w에 대한 참조이며 std::move(w)의 결과입니다. 로컬 객체에 대한 참조를 반환하는 것은 RVO에 필요한 조건을 충족하지 않으므로 컴파일러는 w를 함수의 반환 값 위치로 이동해야 합니다. 반환되는 로컬 변수에 std::move를 적용하여 컴파일러 최적화를 도우려는 개발자는 실제로 컴파일러에서 사용할 수 있는 최적화 옵션을 제한하고 있습니다!

그러나 RVO는 최적화입니다. 컴파일러는 허용된 경우에도 복사 및 이동 작업을 생략할 필요가 없습니다. 편집증이 심해서 컴파일러가 복사 작업으로 벌을 줄까 봐 걱정할 수도 있습니다. 또는 예를 들어 함수의 서로 다른 제어 경로가 서로 다른 지역 변수를 반환하는 경우와 같이 컴파일러가 RVO를 구현하기 어려운 경우가 있음을 인식할 만큼 통찰력이 있습니다. (컴파일러는 함수의 반환 값에 할당된 메모리에 적절한 지역 변수를 생성하기 위해 코드를 생성해야 하지만 컴파일러는 어떤 지역 변수가 적절한지 어떻게 결정할 수 있습니까?) 그렇다면 기꺼이 이동 비용을 지불할 수 있습니다. 사본 비용에 대한 보험으로. 즉, std::move를 반환하는 로컬 객체에 적용하는 것이 여전히 합리적이라고 생각할 수 있습니다. 단순히 복사본에 대해 비용을 지불하지 않을 것이라는 것을 알기 때문에 안심할 수 있기 때문입니다.

이 경우 std::move를 로컬 객체에 적용하는 것은 여전히 나쁜 생각입니다. RVO를 축복하는 표준의 일부는 계속해서 RVO에 대한 조건이 충족되지만 컴파일러가 복사 제거를 수행하지 않기로 선택한 경우 반환되는 객체를 rvalue로 처리해야 한다고 말합니다. 실제로 표준은 RVO가 다음과 같을 때 요구합니다.

허용된 경우 복사 생략이 발생하거나 std::move가 반환되는 로컬 객체에 암시적으로 적용됩니다. 따라서 makeWidget의 "복사" 버전에서는

```
위젯 makeWidget() {           // 이전과

    위젯 w;
    ...

    반환 w;
}
```

컴파일러는 w 복사를 생략하거나 함수를 다음과 같이 작성된 것처럼 처리해야 합니다.

```
위젯 makeWidget() {

    위젯 w;
    ...

    반환 표준::이동(w);           // 복사 생략이 수행되지 않았기 때문에 // w를
}                               rvalue로 처리합니다.
```

상황은 값별 함수 매개변수의 경우와 유사합니다. 함수의 반환 값과 관련하여 복사 생략에 적합하지 않지만 컴파일러는 반환되는 경우 이를 rvalue로 처리해야 합니다. 결과적으로 소스 코드가 다음과 같으면

```
위젯 makeWidget(위젯 w) {           // 함수의 반환값과 같은 유형의 값별 매개변수

    ...
    반환 w;
}
```

컴파일러는 다음과 같이 작성된 것처럼 처리해야 합니다.

```
위젯 makeWidget(위젯 w) {

    ...
    반환 표준::이동(w);           // w를 rvalue로 취급
}
```

이것은 값으로 반환되는 함수에서 반환되는 로컬 객체에 std::move를 사용하는 경우 컴파일러를 도울 수 없음을 의미합니다(수행하지 않으면 로컬 객체를 rvalue로 취급해야 합니다) 복사 생략), 그러나 확실히 방해할 수 있습니다(RVO 제외). 지역 변수에 std::move를 적용하는 것이 합리적인 일이 될 수 있는 상황이 있습니다. 그렇지 않으면 RVO에 적합하거나 by value 매개변수를 반환하는 return 문의 내용이 포함되지 않습니다.

기억해야 할 사항

- rvalue 참조에 std::move를 적용하고 범용 참조에 std::forward를 적용합니다.
각각이 마지막으로 사용된 시간입니다.
- rvalue 참조 및 범용 참조에 대해 동일한 작업을 수행합니다.
값으로 반환하는 함수에서 반환됩니다.
- std::move 또는 std::forward를 로컬 객체에 적용하지 마십시오. 그렇지 않으면 반환 값 최적화에 적합합니다.

항목 26: 범용 참조에 대한 오버로딩을 피하십시오.

이름을 매개변수로 사용하고 현재 날짜와 시간을 기록한 다음 전역 데이터 구조에 이름을 추가하는 함수를 작성해야 한다고 가정합니다. 다음과 같은 함수를 생각해 낼 수 있습니다.

```
std::multiset<std::string> 이름; // 전역 데이터 구조

무효 logAndAdd(const std::string& 이름) {
    지금 자동 = // 현재 시간 가져오기
        std::chrono::system_clock::now();

    log(지금, "logAndAdd"); // 로그 항목 만들기

    이름.emplace(이름); // 전역 데이터에 이름 추가 // 구조체; 항목 42 참
    조 // emplace에 대한 정보
}
```

이것은 불합리한 코드는 아니지만 가능한 한 효율적이지 않습니다. 세 가지 잠재적 호출을 고려하십시오.

```
std::string petName("달라");

logAndAdd(애완동물 이름); // lvalue std::string 전달

logAndAdd(std::string("페르세포네")); // rvalue std::string 전달

logAndAdd("파티 독"); // 문자열 리터럴 전달
```

첫 번째 호출에서 logAndAdd의 매개변수 이름은 변수 petName에 바인딩됩니다.
logAndAdd 내에서 name은 궁극적으로 names.emplace로 전달됩니다. name은 lvalue이므로
이름에 복사됩니다. lvalue(petName)가 logAndAdd에 전달되었기 때문에 해당 복사를 피할 방법이
없습니다.

두 번째 호출에서 매개변수 이름은 rvalue("Persephone"에서 명시적으로 생성된 임시 std::string)에 바인딩됩니다. name 자체는 lvalue이므로 이름으로 복사되지만 원칙적으로 그 값이 이름으로 이동할 수 있음을 인식합니다. 이 통화에서 우리는 사본 비용을 지불하지만

이동하다.

세 번째 호출에서 매개변수 이름은 다시 rvalue에 바인딩되지만 이번에는 "Patty Dog"에서 암시적으로 생성된 임시 std::string입니다. 두 번째 호출에서와 마찬가지로 name은 이름으로 복사되지만 이 경우 원래 logAndAdd에 전달된 인수는 문자열 리터럴입니다. 해당 문자열 리터럴이 emplace에 직접 전달되었다면 임시 std::string을 생성할 필요가 전혀 없었을 것입니다.

대신 emplace는 문자열 리터럴을 사용하여 std::multiset 내부에 직접 std::string 객체를 생성했을 것입니다. 이 세 번째 호출에서 우리는 std::string을 복사하기 위해 비용을 지불하고 있지만 실제로는 이동에 대해 비용을 지불할 이유가 없습니다.

logAndAdd를 다시 작성하여 범용 참조(항목 24 참조)를 수행하고 항목 25에 따라 std::이 참조를 emplace로 전달하여 두 번째 및 세 번째 호출의 비효율성을 제거할 수 있습니다. 결과는 다음과 같이 말해줍니다.

```
template<typename T> 무
효 logAndAdd(T&& 이름) { auto
now =
    std::chrono::system_clock::now(); log(지금, "logAndAdd"); 이름
.emplace(std::forward<T>(이름)); }
```

```
std::string petName("달라"); // 이전과
logAndAdd(애완동물 이름); // 이전과 같이 // lvalue를
// 다중 집합에 복사합니다.
```

```
logAndAdd(std::string("페르세포네")); // 대신 rvalue를 이동 합니다. // 복사하는 대신
```

```
logAndAdd("파티 독"); // 임시 std::string을 복사하는 대
// 신 // 다중 집합에 // std::string 생
성
```

만세, 최적의 효율!

이것이 이야기의 끝이라면 여기서 멈추고 자랑스럽게 은퇴할 수 있지만 클라이언트가 항상 logAndAdd의 이름에 직접 액세스할 수 있는 것은 아닙니다.

필요합니다. 일부 클라이언트에는 logAndAdd가 테이블에서 해당 이름을 찾는 데 사용하는 인덱스만 있습니다. 이러한 클라이언트를 지원하기 위해 logAndAdd가 오버로드됩니다.

```
std::문자열 이름FromIdx(int idx);           // 반환 이름 // idx에
                                                // 해당

무효 logAndAdd(int idx) {                   // 새로운 오버로드

    자동 지금 = std::chrono::system_clock::now(); log(지금,
    "logAndAdd"); 이름.emplace(이름FromIdx(idx));
}

}
```

두 오버로드에 대한 호출 해결은 예상대로 작동합니다.

```
std::string petName("달라");                // 이전과

logAndAdd(애완동물 이름); // 이전과 마찬가지로 이러한
logAndAdd(std::string("Persephone")); // 모든 호출 호출 logAndAdd("Patty
Dog"); // T&& 오버로드

로그 및 추가(22);                          // int 오버로드 호출
```

사실, 해상도는 너무 많이 기대하지 않는 경우 예상대로 작동합니다. 클라이언트가 인덱스를 보유하고 있고 이를 logAndAdd에 전달한다고 가정합니다.

```
짧은 이름idx;
...
// namelidx에 값을 줍니다.

logAndAdd(이름 IDx);                     // 오류!
```

마지막 줄에 대한 설명은 그다지 명료하지 않으므로 여기서 무슨 일이 일어나는지 설명하겠습니다.

두 개의 logAndAdd 오버로드가 있습니다. 범용 참조를 사용하는 사람은 T가 짧은 것으로 추론할 수 있으므로 정확한 일치를 얻을 수 있습니다. int 매개변수가 있는 오버로드는 프로모션과만 짧은 인수와 일치할 수 있습니다. 일반적인 오버로드 해결 규칙에 따라 정확한 일치는 승격과의 일치를 능가하므로 범용 참조 오버로드가 호출됩니다.

그 오버로드 내에서 매개변수 이름은 전달된 short에 바인딩됩니다. 그러면 name은 std::forwarded하여 이름의 emplace 멤버 함수(std::multiset<std::string>)로 전달됩니다. std::string 생성자로 전달합니다. short가 필요한 std::string에 대한 생성자가 없으므로 multiset::emplace 호출 내부에서 std::string 생성자 호출이 호출됩니다.

내부에서 logAndAdd에 대한 호출이 실패합니다. 범용 참조 오버로드가 int보다 짧은 인수에 더 적합 했기 때문입니다.

범용 참조를 사용하는 함수는 C++에서 가장 탐욕스러운 함수입니다. 거의 모든 유형의 인수에 대해 정확히 일치하도록 인스턴스화합니다. (이것이 사실이 아닌 몇몇 종류의 인수는 항목 30에 설명되어 있습니다.) 이것이 오버로딩과 범용 참조를 결합하는 것이 거의 항상 나쁜 생각인 이유입니다. 범용 참조 오버로드는 개발자보다 훨씬 더 많은 인수 유형을 정리합니다. 오버로딩을 수행하는 것은 일반적으로 예상됩니다.

이 구덩이에 빠지는 쉬운 방법은 완벽한 전달 생성자를 작성하는 것입니다. logAndAdd 예제에 대한 약간의 수정은 문제를 보여줍니다. std::string 또는 std::string을 조회하는 데 사용할 수 있는 인덱스를 사용할 수 있는 자유 함수를 작성하는 대신 동일한 작업을 수행하는 생성자가 있는 Person 클래스를 상상해 보세요.

```
class Person
{ public:
    template<typename T> 명
    시적 Person(T&& n) : // 퍼펙트 포워딩 ctor; // 데이터 멤버 초기화
        name(std::forward<T>(n)) {}

    명시적 Person(int idx) : 이름 // 정수 ctor
        (nameFromIdx(idx)) {}

    ...
}

개인: 표준::
문자열 이름;};
```

logAndAdd의 경우와 마찬가지로 int 이외의 정수 유형(예: std::size_t, short, long 등)을 전달하면 int 오버로드 대신 범용 참조 생성자 오버로드를 호출하고 이는 컴파일로 이어집니다. 실패. 그러나 여기서 문제는 훨씬 더 나쁩니다. Person에 눈에 보이는 것보다 더 많은 과부하가 있기 때문입니다. 항목 17은 적절한 조건에서 C++가 복사 생성자와 이동 생성자를 모두 생성한다고 설명하며, 이는 복사 또는 이동 생성자의 서명을 생성하기 위해 인스턴스화할 수 있는 템플릿화된 생성자가 클래스에 포함된 경우에도 마찬가지입니다. Person에 대한 복사 및 이동 생성자가 생성되면 Person은 효과적으로 다음과 같이 보일 것입니다.

```
class Person
{ public:
    template<typename T> 명 // 퍼펙트 포워딩 ctor
    시적 Person(T&& n) :
        name(std::forward<T>(n)) {}
```

명시적 사람(int idx);	// 정수 ctor
사람(const Person& rhs);	// ctor 복사 // (컴파일러 생성)
사람(사람&& rhs);	// ctor 이동 // (컴파일러 생성)
...	
};	

이것은 당신이 컴퓨터 주변에서 너무 많은 시간을 보낸 경우에만 직관적인 행동으로 이어집니다. 파일러와 컴파일러 작성자는 인간이 어떤 것인지 잊어버렸습니다.

사람 p("낸시");

```
auto cloneOfP(p);  
// p에서 새로운 Person 생성;  
// 컴파일되지 않습니다!
```

여기에서 우리는 다음과 같이 보이는 다른 Person에서 Person을 만들려고 합니다.

복사 구성에 대한 명백한 사례는 얻을 수 있습니다. (*p*는 lvalue이므로 추방할 수 있습니다.)

이동을 통해 수행되는 "복사"에 대해 생각할 수 있는 모든 생각

operation.) 그러나 이 코드는 복사 생성자를 호출하지 않습니다. 완벽한 전달 생성자를 호출합니다. 그 함수는 Person의 초기화를 시도할 것입니다.

Person 객체(p)가 있는 std::string 데이터 멤버. std::문자열이 없는 문자열 생성자가 Person을 사용하면 컴파일러는 분노에 찬 손을 던질 것입니다.
길고 이해할 수 없는 오류 메시지로 당신을 처벌할 수 있습니다.
그들의 불만.

"왜" 완벽 전달 생성자가 호출됩니까?

복사 생성자 대신? Person을 다른 Person으로 초기화하고 있습니다!"

실제로 우리는 그렇습니다. 그러나 컴파일러는 C++의 규칙과

여기서 관련성은 오버로드된 함수에 대한 호출의 해결을 관리하는 것들입니다.

컴파일러는 다음과 같이 추론합니다. `cloneOfP`가 비 `const lvalue`로 초기화되고 있습니다.

(p), 이는 템플릿화된 생성자를 인스턴스화하여

Person 유형의 비 const lvalue입니다. 이러한 인스턴스화 후 Person 클래스는 다음과 같습니다.
이것:

클래스 사람 {

공공의:

명시적 Person(Person& n) : 이름
(std::forward<Person&>(n)) {} //에서 인스턴스화
// 퍼펙트 포워딩
// 템플릿

명시적 사람(int idx): // 이전과

```
    사람(const Person& rhs);
    ...
    // 복사 ctor // (컴파일러 생성)

};

성명서에서,
auto cloneOfP(p);
```

p는 복사 생성자나 인스턴스화된 템플릿에 전달될 수 있습니다. 복사 생성자를 호출하려면 복사 생성자의 매개 변수 유형과 일치하도록 p에 const를 추가해야 하지만 인스턴스화된 템플릿을 호출하면 이러한 추가가 필요하지 않습니다.

따라서 템플릿에서 생성된 오버로드가 더 잘 일치하므로 컴파일러는 더 잘 일치하는 함수에 대한 호출을 생성하도록 설계된 작업을 수행합니다. 따라서 Person 유형의 비 const lvalue "복사"는 복사 생성자가 아니라 완전 전달 생성자에 의해 처리됩니다.

복사할 객체가 `const`가 되도록 예제를 약간 변경하면 완전히 다른 조정이 들립니다.

```
const 사람 cp("낸시"); // 객체는 이제 const입니다.
```

```
auto cloneOfP(cp); // 복사 생성자를 호출합니다!
```

복사할 개체가 이제 `const`으로 복사 생성자가 가져온 매개변수와 정확히 일치합니다. 템플릿화된 생성자는 동일한 서명을 갖도록 인스턴스화될 수 있습니다.

```
클래스 Person { 공  
개: 명시적  
Person(const Person& n);
```

// 템플릿에서 // 인스턴스화

```
    사람(const Person& rhs);  
    ...  
};  
// 복사 ctor // (컴파일러 생성)
```

그러나 이것은 중요하지 않습니다. 왜냐하면 C++의 오버로드 해결 규칙 중 하나는 템플릿 인스턴스화와 템플릿이 아닌 함수(즉, "정상" 함수)가 함수 호출에 동등하게 잘 일치하는 상황에서라는 것입니다. , 일반 기능이 선호됩니다. 따라서 복사 생성자(일반 함수)는 동일한 서명을 가진 인스턴스화된 템플릿을 능가합니다.

(컴파일러가 템플릿화된 생성자를 인스턴스화하여 복사 생성자가 가질 서명을 얻을 수 있을 때 복사 생성자를 생성하는 이유가 궁금하다면 [항목 17](#)을 검토하세요.)

완전 전달 생성자와 컴파일러 생성 복사 및 이동 작업 간의 상호 작용은 상속이 그림에 들어갈 때 훨씬 더 많은 주름을 발생시킵니다. 특히 파생 클래스 복사 및 이동 작업의 기준 구현은 매우 놀라울 정도로 작동합니다. 여기, 살펴보세요:

```
class SpecialPerson: public Person { public:  
    SpecialPerson(const SpecialPerson& rhs) // 복  
    사 ctor; Calls : Person(rhs) // 기본 클래스 { ... } // ctor를 전달합니다!
```

```
SpecialPerson(SpecialPerson&& rhs) // ctor 이동; 호출 // 기본 클  
: 사람(표준::이동(rhs)) { … }; 래스 // 전달 ctor!
```

주석에서 알 수 있듯이 파생 클래스 복사 및 이동 생성자는 기본 클래스의 복사 및 이동 생성자를 호출하지 않고 기본 클래스의 완벽한 전달 생성자를 호출합니다! 이유를 이해하려면 파생 클래스 함수가 SpecialPerson 유형의 인수를 사용하여 기본 클래스에 전달한 다음 Person 클래스의 생성자에 대한 템플릿 인스턴스화 및 과부하 해결 결과를 통해 작업한다는 점에 유의하십시오. SpecialPerson을 사용하는 std::string 생성자가 없기 때문에 궁극적으로 코드가 컴파일되지 않습니다.

저는 지금까지 범용 참조 매개변수에 대한 오버로딩이 가능하면 피해야 하는 것임을 확신했기를 바랍니다. 그러나 범용 참조에 대한 오버로딩이 나쁜 생각이라면 대부분의 인수 유형을 전달하지만 일부 인수 유형을 특별한 방식으로 처리해야 하는 함수가 필요하다면 어떻게 해야 할까요?

그 계란은 여러 가지 방법으로 스크램블을 풀 수 있습니다. 사실 너무 많아서 전체 항목을 그들에게 바쳤습니다. [27](#) 번 항목입니다. 다음 항목입니다. 계속 읽으십시오, 당신은 그것에 부딪힐 것입니다.

기억할 사항 • 범용 참조

에 대한 오버로딩은 거의 항상 범용 참조 과부하가 예상보다 더 자주 호출되도록 합니다. • 완전 전달 생성자는 일반적으로 비 const lvalue에 대한 복사 생성자보다 더 잘 일치하고 기본 클래스 복사 및 이동 생성자에 대한 파생 클래스 호출을 가로채기 때문에 특히 문제가 됩니다.

항목 27: 범용 참조 오버로딩에 대한 대안 을 숙지하십시오.

[항목 26](#)은 범용 참조에 대한 오버로딩이 독립형 및 멤버 함수(특히 생성자) 모두에 대해 다양한 문제를 일으킬 수 있다고 설명합니다.

그러나 그러한 오버로딩이 유용할 수 있는 예도 제공합니다. 우리가 원하는 대로 행동하기만 하면! 이 항목은 범용 참조에 대한 오버로드를 피하거나 일치할 수 있는 인수 유형을 제한하는 방식으로 사용하여 원하는 동작을 달성하는 방법을 탐구합니다.

다음 논의는 [항목 26](#)에 소개된 예를 기반으로 합니다. 최근에 해당 항목을 읽지 않았다면 계속하기 전에 검토하고 싶을 것입니다.

오버로딩 포기

[항목 26](#)의 첫 번째 예인 logAndAdd는 단순히 오버로드가 예상되는 다른 이름을 사용하여 범용 참조에 대한 오버로드의 단점을 피할 수 있는 많은 함수를 나타냅니다. 예를 들어 두 개의 logAndAdd 오버로드는 logAndAddName 및 logAndAddNameIdx로 나눌 수 있습니다. 아아, 이 접근 방식은 우리가 고려한 두 번째 예인 Person 생성자에서는 작동하지 않습니다. 왜냐하면 생성자 이름은 언어에 의해 고정되기 때문입니다. 게다가 누가 과부하를 포기하고 싶습니까?

Pass by const T& 대

안은 C++98로 되돌리고 pass-by-universal-reference를 pass-by-lvalue-reference-to-const로 바꾸는 것입니다. 사실, 이것이 [Item 26](#)이 고려하는 첫 번째 접근 방식입니다([175페이지 참조](#)). 단점은 디자인이 우리가 원하는 만큼 효율적이지 않다는 것입니다.

범용 참조와 오버로딩의 상호 작용에 대해 이제 우리가 알고 있는 것을 알면, 일을 단순하게 유지하기 위해 일부 효율성을 포기하는 것이 처음에 나타난 것보다 더 매력적인 절충안이 될 수 있습니다.

값에 의한 전달 복

잡성을 증가시키지 않고 성능을 향상시킬 수 있는 접근 방식은 참조에 의한 전달 매개변수를 직관적으로 값에 의한 전달로 대체하는 것입니다. 디자인은 [항목 41](#)의 조언을 따르고 객체를 복사할 것임을 알 때 값으로 전달하는 것을 고려합니다. 따라서 어떻게 작동하고 얼마나 효율적인지에 대한 자세한 논의는 해당 항목을 참조하겠습니다. 여기서는 Person 예제에서 이 기술을 어떻게 사용할 수 있는지 보여드리겠습니다.

```
class Person
{ public: 명시적
    Person(std::string n) // T&& ctor를 대체합니다. 보다
```

```

    : 이름(std::move(n)) {}           // std::move 사용을 위한 항목 41

    명시적 Person(int idx) : 이름      // 이전과
    (nameFromIdx(idx)) {}
    ...
}

개인: 표준::
문자열 이름; };

```

정수만 취하는 std::string 생성자가 없기 때문에 Person 생성자에 대한 모든 int 및 int 유사 인수 (예: std::size_t, short, long)는 int 오버로드로 유입됩니다. 유사하게, std::string 유형의 모든 인수(및 std::strings가 생성될 수 있는 것, 예를 들어 "Ruth"와 같은 리터럴)는 std::string을 사용하는 생성자로 전달됩니다. 따라서 발신자에게는 놀라움이 없습니다. 어떤 사람들은 null 포인터를 나타내기 위해 0 또는 NULL을 사용하는 것이 int 오버로드를 호출한다는 사실에 놀랄 수 있지만, 그런 사람들은 [항목 8](#)을 참조하고 0을 사용할 생각이 있을 때까지 반복적으로 읽어야 한다고 주장할 수 있습니다. 또는 널 포인터로서의 NULL은 반동을 일으킵니다.

태그 디스패치 사용

lvalue-reference-to-const에 의한 전달이나 값에 의한 전달 모두 완벽한 전달을 지원하지 않습니다. 범용 참조를 사용하는 동기가 완전 전달인 경우 범용 참조를 사용해야 합니다. 다른 선택은 없습니다. 그러나 우리는 오버로딩을 포기하고 싶지 않습니다. 따라서 오버로딩을 포기하지 않고 범용 참조를 포기하지 않는다면 어떻게 범용 참조에 대한 오버로드를 피할 수 있습니까?

사실 그렇게 어렵지 않습니다. 오버로드된 함수에 대한 호출은 모든 오버로드의 모든 매개변수와 호출 사이트의 모든 인수를 살펴본 다음 모든 매개변수/인수 조합을 고려하여 전체적으로 가장 잘 일치하는 함수를 선택하여 해결됩니다. 범용 참조 매개변수는 일반적으로 전달된 모든 항목에 대해 정확한 일치를 제공하지만 범용 참조가 범용 참조가 아닌 다른 매개변수를 포함하는 매개변수 목록의 일부인 경우 비범용 참조 매개변수에 대한 충분히 불량한 일치는 과부하를 노크할 수 있습니다. 실행에서 보편적인 참조와 함께. 이것이 태그 디스패치 접근 방식의 기초이며 예제를 통해 앞서 설명한 설명을 더 쉽게 이해할 수 있습니다.

[177페이지](#)의 logAndAdd 예제에 태그 디스패치를 적용할 것 입니다. 다음은 해당 예제의 코드입니다.

```

std::multiset<std::string> 이름;           // 전역 데이터 구조

템플릿<유형이름 T>                     // 로그 항목을 만들고 추가

```

```
무효 logAndAdd(T&& 이름) { 자동 // 데이터 구조의 이름
    지금 =
        std::chrono::system_clock::now(); log(지금, "logAndAdd");
        이름.emplace(std::forward<T>(이름)); }
```

그 자체로 이 함수는 잘 작동하지만 인덱스로 객체를 찾는 데 사용되는 int를 사용하는 오버로드를 도입 했다면 문제의 항목 26 으로 돌아갈 것 입니다. 이 항목의 목표는 이를 방지하는 것입니다. 오버로드를 추가하는 대신 logAndAdd를 다시 구현하여 두 개의 다른 함수(하나는 정수 값용이고 다른 하나는 다른 모든 함수)에 위임합니다. logAndAdd 자체는 정수와 정수가 아닌 모든 인수 유형을 허용합니다.

실제 작업을 수행하는 두 함수의 이름은 logAndAddImpl로 지정됩니다. 즉, 오버로딩을 사용합니다. 함수 중 하나는 범용 참조를 사용합니다. 그래서 우리는 오버로딩과 유니버설 참조를 둘 다 갖게 될 것입니다. 그러나 각 함수는 전달되는 인수가 정수인지 여부를 나타내는 두 번째 매개변수도 사용합니다. 이 두 번째 매개변수는 항목 26에 설명된 황무지에 빠지는 것을 방지하는 것입니다. 두 번째 매개변수가 어떤 과부하가 선택되는지를 결정하는 요소가 되도록 정렬할 것이기 때문입니다.

네, 알겠습니다. “블라, 블라, 블라. 그만 말하고 코드를 보여줘!” 문제 없어요.
다음은 업데이트된 logAndAdd의 거의 정확한 버전입니다.

```
template<typename T> 무
효 logAndAdd(T&& 이름) {

    logAndAddImpl(std::forward<T>(이름),
                  std::is_integral<T>()); // 정확하지 않음
}
```

이 함수는 매개변수를 logAndAddImpl로 전달하지만 해당 매개변수의 유형(T)이 정수인지 여부를 나타내는 인수도 전달합니다. 최소한 그렇게 해야 합니다. rvalue인 정수 인수의 경우에도 마찬가지입니다. 그러나 항목 28 에서 설명하는 것처럼 lvalue 인수가 범용 참조 이름에 전달되면 T에 대해 추론된 유형은 lvalue 참조가 됩니다. 따라서 int 유형의 lvalue가 logAndAdd에 전달되면 T는 int&로 추론됩니다. 참조는 정수형이 아니기 때문에 정수형이 아닙니다. 즉, 인수가 실제로 정수 값을 나타내더라도 std::is_integral<T>는 모든 lvalue 인수에 대해 false가 됩니다.

문제를 인식하는 것은 문제를 해결하는 것과 같습니다. 왜냐하면 항상 편리한 표준 C++ 라이브러리에는 유형 특성(항목 9 참조), std::remove_reference가 있어 이름이 암시하는 것과 우리가 필요로 하는 것 모두를 수행하기 때문입니다. 모든 참조 한정자 제거 유형에서. 따라서 logAndAdd를 작성하는 적절한 방법은 다음과 같습니다.

```

template<유형 이름 T> 무효
logAndAdd(T&& 이름)
{ logAndAddImpl( std::forward<T>(이
    름), std::is_integral<유형 이름
        std::remove_reference<T>::type>() );
}

```

이것은 트릭을 수행합니다. (C++14에서는 강조 표시된 텍스트 대신 `std::remove_reference_t<T>`를 사용하여 몇 가지 키 입력을 저장할 수 있습니다. 자세한 내용은 [항목 9](#)를 참조하세요.)

이를 처리하면 호출되는 함수 `logAndAddImpl`에 주의를 기울일 수 있습니다. 두 가지 오버로드가 있으며 첫 번째는 정수가 아닌 형식에만 적용할 수 있습니다(즉, `std::is_integral<typename std::remove_ref erence<T>::type> == false`인 형식).

```

template<typename T> 무효 logAndAddImpl(T&& 이름, std::false_type) { // 정수가 아닌 // 인수: // 추가
    자동 지금 = std::chrono::system_clock::now(); // 전역 데이터 log(now, "logAndAdd");
    이름.emplace(std::forward<T>(이름)); // 구조
}

```

강조 표시된 매개변수의 역학을 이해하면 간단한 코드입니다. 개념적으로 `logAndAdd`는 적분 유형이 `logAndAdd`에 전달되었는지 여부를 나타내는 부울을 `logAndAddImpl`에 전달하지만 `true`와 `false`는 런타임 값이며 올바른 `logAndAddImpl` 오버로드를 선택하기 위해 오버로드 해결(컴파일 시간 현상)을 사용해야 합니다. 즉, `true`에 해당하는 유형과 `false`에 해당하는 다른 유형이 필요합니다. 이러한 요구는 표준 라이브러리가 `std::true_type` 및 `std::false_type`이라는 이름으로 필요한 것을 제공할 만큼 충분히 일반적입니다. `logAndAdd`에 의해 `logAndAddImpl`에 전달된 인수는 `T`가 정수이면 `std::true_type`에서, `T`가 정수가 아니면 `std::false_type`에서 상속되는 유형의 개체입니다. 최종 결과는 이 `logAndAddImpl` 오버로드가 `T`가 정수 유형이 아닌 경우에만 `logAndAdd` 호출에 대해 실행 가능한 후보라는 것입니다.

두 번째 과부하는 반대 경우를 다룹니다: `T`가 정수형일 때. 이 경우 `logAndAddImpl`은 전달된 인덱스에 해당하는 이름을 찾고 해당 이름을 다시 `logAndAdd`에 전달합니다.

```
std::문자열 이름FromIdx(int idx); // 항목 26 에서와 같이
```

```
무효 logAndAddImpl(int idx, std::true_type) // 정수 { logAndAdd(nameFromIdx(idx)); }
// 인수: // 이름을 찾고 //
logAndAdd를 // 호출합니다.
```

인덱스에 대한 logAndAddImpl이 해당 이름을 찾아 logAndAdd에 전달하도록 하면(여기에서 다른 logAndAddImpl 오버로드로 전달됨) 로깅 코드를 두 logAndAddImpl 오버로드에 둘 필요가 없습니다.

이 디자인에서 `std::true_type` 및 `std::false_type` 유형은 우리가 원하는 방식으로 과부하 해결을 강제 실행하는 것이 유일한 목적인 "태그"입니다. 이러한 매개변수의 이름도 지정하지 않았습니다. 그 것들은 런타임에 아무런 역할을 하지 않으며 사실 우리는 컴파일러가 태그 매개변수가 사용되지 않는다는 것을 인식하고 프로그램의 실행 이미지에서 태그 매개변수를 최적화하기를 바랍니다. (일부 컴파일러는 적어도 일부는 그렇게 합니다.) `logAndAdd` 내부의 오버로드된 구현 함수에 대한 호출은 적절한 태그 개체가 생성되도록 하여 작업을 올바른 오버로드로 "전달"합니다. 따라서 이 디자인의 이름은 태그 디스패치입니다. 이것은 템플릿 메타프로그래밍의 표준 빌딩 블록이며 현대 C++ 라이브러리 내부의 코드를 더 많이 볼수록 더 자주 접하게 될 것입니다.

우리의 목적을 위해 태그 디스패치에 대해 중요한 것은 작동 방식보다는 [항목 26](#)에 설명된 문제 없이 범용 참조와 오버로딩을 결합하는 방법입니다. 디스패칭 함수인 `logAndAdd`는 제약 없는 범용 참조 매개변수를 취합니다, 하지만 이 함수는 오버로드되지 않습니다. 구현 함수인 `logAndAddImpl`은 오버로드되고 하나는 범용 참조 매개변수를 취하지만 이러한 함수에 대한 호출의 해결은 범용 참조 매개변수뿐만 아니라 태그 매개변수에 따라 달라지며 태그 값이 설계됩니다. 하나 이상의 과부하가 실행 가능한 일치가 되지 않도록 합니다. 결과적으로 호출되는 오버로드를 결정하는 것은 태그입니다. 범용 참조 매개변수가 항상 해당 인수와 정확히 일치한다는 사실은 중요하지 않습니다.

범용 참조를 사용하는 템플릿 제한 태그 디스패치의 핵심은 클라이언트 API로

단일(오버로드되지 않은) 기능의 존재입니다. 이 단일 함수는 수행할 작업을 구현 함수에 전달합니다. 오버로드되지 않은 디스패치 함수를 만드는 것은 일반적으로 쉽지만 [항목 26](#)에서 고려하는 두 번째 문제 사례인 Person 클래스에 대한 완전 전달 생성자의 경우([178페이지 참조](#))는 예외입니다. 컴파일러는 복사 및 이동 생성자를 스스로 생성할 수 있으므로 생성자 하나만 작성하고 그 안에서 태그 디스패치를 사용하더라도 일부 생성자 호출은 태그 디스패치 시스템을 우회하는 컴파일러 생성 함수에 의해 처리될 수 있습니다.

사실 진짜 문제는 컴파일러가 생성한 함수가 때때로 태그 디스패치 디자인을 우회하는 것이 아니라 항상 통과하지 않는다는 것입니다. 거의 항상 클래스의 복사 생성자가 해당 유형의 lvalue 복사 요청을 처리하기를 원하지만 [항목 26](#)에서 알 수 있듯이 범용 참조를 사용하는 생성자를 제공하면 범용 참조 생성자(복사 생성자가 아닌)가 비 const lvalue를 복사할 때 호출됩니다. 그 항목은 또한 기본 클래스가 완전 전달 생성자를 선언할 때 올바른 동작이 기본 클래스의 복사 및 호출할 이동 생성자.

범용 참조를 사용하는 오버로드된 함수가 원하는 것보다 탐욕스럽지만 단일 디스패치 기능으로 작동할 만큼 탐욕스럽지 않은 이와 같은 상황에서 태그 디스패치는 원하는 droid가 아닙니다. 범용 참조가 포함된 함수 템플릿이 사용되도록 허용되는 조건을 세분화할 수 있는 다른 기술이 필요합니다. 당신에게 필요한 것은 std::enable_if입니다.

std::enable_if는 컴파일러가 특정 템플릿이 존재하지 않는 것처럼 작동하도록 하는 방법을 제공합니다. 이러한 템플릿은 비활성화되어 있습니다. 기본적으로 모든 템플릿이 활성화되지만 std::enable_if를 사용하는 템플릿은 std::enable_if에 지정된 조건이 충족되는 경우에만 활성화됩니다. 우리의 경우 전달되는 유형이 Person이 아닌 경우에만 Person 완전 전달 생성자를 활성화하려고 합니다. 전달되는 유형이 Person이면 완전 전달 생성자를 비활성화하려고 합니다(즉, 컴파일러가 이를 무시하도록 함). 그렇게 하면 클래스의 복사 또는 이동 생성자가 호출을 처리하게 되므로 Person 객체는 다른 Person으로 초기화됩니다.

그 아이디어를 표현하는 방법은 특별히 어렵지는 않지만, 특히 이전에 한 번도 본 적이 없는 경우 구문이 엉뚱하기 때문에 쉽게 설명하겠습니다. std::enable_if의 조건 부분을 둘러싸는 일부 사용구가 있으므로 이것으로 시작하겠습니다.

다음은 Person의 완전 전달 생성자에 대한 선언으로, 단순히 사용하는 데 필요한 만큼만 std::enable_if를 표시합니다. std::enable_if를 사용하면 함수 구현에 영향을 미치지 않기 때문에 이 생성자에 대한 선언만 표시합니다. 구현은 [항목 26](#)과 동일하게 유지됩니다.

```
클래스 Person { 공
개: 템플릿<유형 이름
T, 유형 이름 = 유형 이름
std::enable_if<조건>::유형> 명시적 Person(T&& n);
```

...

```
};
```

강조 표시된 텍스트를 정확히 이해하려면 세부 사항을 설명하는 데 시간이 걸리고 이 책에서 설명할 공간이 충분하지 않기 때문에 유감스럽게도 다른 출처를 참조하시기 바랍니다. (연구하는 동안 "SFINAE"와 std::enable_if를 살펴보십시오. SFINAE는 std::enable_if를 작동시키는 기술이기 때문입니다.) 여기서 저는 이 생성자가 활성화.

우리가 지정하고자 하는 조건은 T가 Person이 아니라는 것, 즉 T가 Person이 아닌 다른 유형인 경우에만 템플릿화된 생성자가 활성화되어야 한다는 것입니다. 두 유형이 동일한지 여부를 결정하는 유형 특성(std::is_same) 덕분에 우리가 원하는 조건은 !std::is_same<Person, T>::value인 것 같습니다. (표현식의 시작 부분에 있는 "!"에 주목하십시오. Person과 T가 같지 않기를 바랍니다.)

이것은 우리가 필요로 하는 것에 가깝지만, [항목 28](#)에서 설명하는 것처럼 lvalue로 초기화된 범용 참조에 대해 추론된 유형은 항상 lvalue 참조 이기 때문에 정확하지 않습니다. 즉, 이와 같은 코드의 경우

```
사람 p("낸시");  
auto cloneOfP(p); // lvalue에서 초기화
```

범용 생성자의 유형 T는 Person&로 추론됩니다. Person 및 Person& 유형은 동일하지 않으며 std::is_same의 결과는 std::is_same<Person, Person&>::value가 false임을 반영합니다.

Person의 템플릿화된 생성자는 T가 Person이 아닌 경우에만 활성화되어야 한다고 말할 때 우리가 의미하는 바를 보다 정확하게 생각하면 T를 볼 때 무시하고 싶어한다는 것을 깨닫게 될 것입니다.

- 참조인지 여부. 범용 참조 생성자를 활성화해야 하는지 여부를 결정하기 위해 Person, Person& 및 Person&& 유형은 모두 Person과 동일합니다.
- const 또는 volatile인지 여부. 우리가 아는 한 const Person과 휘발성 Person과 const 휘발성 Person은 모두 Person과 동일합니다.

이는 해당 유형이 Person과 동일한지 확인하기 전에 T에서 참조, const 및 휘발성을 제거하는 방법이 필요하다는 것을 의미합니다. 다시 한 번, 표준 라이브러리는 유형 특성의 형태로 필요한 것을 제공합니다. 그 특성은 std::decay입니다. std::decay<T>::type은 참조와 cv 한정자(즉, const 또는 volatile 한정자)가 제거된다는 점을 제외하고는 T와 동일합니다. (이름에서 알 수 있듯이 std::decay는 배열 및 함수 유형을 포인터로 바꾸기 때문에 여기서 진실을 왜곡하고 있습니다).

([항목 1](#) 참조), 그러나 이 논의의 목적을 위해 std::decay는 내가 설명한 대로 동작합니다. 그러면 생성자가 활성화되었는지 여부를 제어하려는 조건은 다음과 같습니다.

```
!std::is_same<사람, 유형 이름 std::decay<T>::유형>::값
```

즉, Person은 참조 또는 cv 한정자를 무시하고 T와 동일한 유형이 아닙니다. ([항목 9](#) 에서 설명하는 것처럼 std::decay 앞에 "typename"이 필요합니다. std::decay<T>::type 유형은 템플릿 매개변수 T에 의존하기 때문입니다.)

이 조건을 위의 std::enable_if 상용구에 삽입하고 결과 형식을 지정하여 조각이 어떻게 서로 맞는지 더 쉽게 볼 수 있도록 하면 Person의 완전 전달 생성자에 대한 다음 선언이 생성됩니다.

```
class Person
{ public: template<
    typename T,
    typename =
    typename std::enable_if< !std::is_same<Person,
        typename std::decay<T>::type
        >::value

    >>:유형
>
명시적 사람(T&& n);

...
};

};
```

이전에 이와 같은 것을 본 적이 없다면 축복을 세어보십시오. 이 디자인을 마지막으로 저장한 이유가 있습니다. 범용 참조와 오버로딩을 혼합하는 것을 피하기 위해 다른 메커니즘 중 하나를 사용할 수 있는 경우(거의 항상 할 수 있음), 그렇게 해야 합니다. 그래도 함수 구문과 꺥쇠 팔호의 확산에 익숙해지면 그렇게 나쁘지 않습니다. 게다가 이것은 당신이 추구해 온 행동을 제공합니다. 위의 선언이 주어지면 lvalue 또는 rvalue, const 또는 non-const, volatile 또는 non-volatile과 같은 다른 Person에서 Person을 구성하면 범용 참조를 사용하는 생성자가 호출되지 않습니다.

성공, 그렇지? 우리는 끝났어!

음 .. 아니야. 축하합니다. 계속해서 펄럭이는 [항목 26](#) 의 느슨한 끝이 하나 있습니다 . 우리는 그것을 묶을 필요가 있습니다.

Person에서 파생된 클래스가 일반적인 방식으로 복사 및 이동 작업을 구현한다고 가정합니다.

```
class SpecialPerson: public Person { public:  
    SpecialPerson(const SpecialPerson& rhs) // 복  
        사 ctor; Calls : Person(rhs) // 기본 클래스 { ... } // ctor를 전달합니다!
```

```
SpecialPerson(SpecialPerson&& rhs) // ctor 이동; 호출 // 기본 클  
    : 사람(표준::이동(rhs)) { ... } 래스 // 전달 ctor!
```

```
...  
};
```

이것은 주석을 포함하여 [항목 26](#) (206페이지)에서 보여준 것과 동일한 코드입니다. 안타깝게도 이 코드는 여전히 정확합니다. SpecialPerson 개체를 복사하거나 이동할 때 기본 클래스의 복사 및 이동 생성자를 사용하여 기본 클래스 부분을 복사하거나 이동할 것으로 예상하지만 이러한 함수에서 SpecialPerson 개체를 기본 클래스의 생성자에 전달합니다. Person과 동일하지 않고(std::decay 적용 후에도) 기본 클래스의 범용 참조 생성자가 활성화되고 SpecialPerson 인수에 대한 정확한 일치를 수행하기 위해 행복하게 인스턴스화됩니다. 이 정확한 일치는 SpecialPerson 개체를 Person의 복사 및 이동 생성자의 Person 매개변수에 바인딩하는 데 필요한 파생-베이스 변환보다 낫습니다. 따라서 지금 가지고 있는 코드를 사용하면 SpecialPerson 개체를 복사하고 이동하면 Person 기본 클래스 부분을 복사하거나 이동하는 완벽한 전달 생성자! 다시 데자 [항목 26](#) 입니다.

파생 클래스는 파생 클래스 복사 및 이동 생성자를 구현하는 일반적인 규칙을 따르고 있으므로 이 문제에 대한 수정은 기본 클래스, 특히 Person의 범용 참조 생성자가 활성화되었는지 여부를 제어하는 조건에 있습니다. 이제 우리는 Person 이외의 인수 유형에 대해 템플릿화된 생성자를 활성화하고 싶지 않다는 것을 깨달았습니다. 성가신 상속!

표준 유형 특성 중 한 유형이 다른 유형에서 파생되는지 여부를 결정하는 특성이 있다는 사실에 놀라지 마십시오. std::is_base_of라고 합니다. std::is_base_of<T1, T2>::value는 T2가 T1에서 파생된 경우 true입니다. 유형은 자체에서 파생된 것으로 간주되므로 std::is_base_of<T, T>::value가 true입니다.

이는 참조 및 cv 한정자를 제거한 후 유형 T가 Person도 아니고 Person에서 파생된 클래스도 아닌 경우에만 생성자가 활성화되도록 Person의 완전 전달 생성자를 제어하는 조건을 수정하기를 원하기 때문에 편리합니다. std::is_same 대신 std::is_base_of를 사용하면 필요한 것을 얻을 수 있습니다.

```

class Person
{ public: template<
    typename T,
    typename =
    typename std::enable_if<!std::is_base_of<Person,
        typename std::decay<T>::type
    >::value

    >>:유형
}

명시적 사람(T&& n);

...
};


```

이제 드디어 끝났습니다. 우리가 C++11로 코드를 작성한다면 말입니다. C++14를 사용하는 경우 이 코드는 여전히 작동하지만 `std::enable_if` 및 `std::decay`에 대한 별칭 템플릿을 사용하여 "typename" 및 "::type" cruft를 제거할 수 있습니다. 이 다소 더 입맛에 맞는 코드:

```

class Person // C++14
{ public: template<
    typename T,
    typename =
    std::enable_if_t<!
        std::is_base_of<Person,
            std::decay_t<T> // 여기 >::value

        > // 그리고 여기
}

명시적 사람(T&& n);

...
};


```

알겠습니다. 인정합니다. 거짓말을 했습니다. 아직 끝나지 않았습니다. 하지만 우리는 가깝습니다. 감질나게 가깝다. 솔직한.

우리는 `std::enable_if`를 사용하여 클래스의 복사 및 이동 생성자가 처리하고자 하는 인수 유형에 대해 `Person`의 범용 참조 생성자를 선택적으로 비활성화하는 방법을 보았지만 정수를 구별하기 위해 적용하는 방법은 아직 보지 못했습니다. 정수가 아닌 인수. 그것이 결국 우리의 원래 목표였습니다. 생성자 모호성 문제는 우리가 그 과정에서 끌어들인 문제였습니다.

우리가 해야 할 일은 (1) 통합 인수를 처리하기 위해 Person 생성자 오버로드를 추가하고 (2) 템플릿화된 생성자를 추가로 제한하여 이러한 인수에 대해 비활성화되도록 하는 것입니다. 이 자료들을 우리가 논의한 다른 모든 것과 함께 냄비에 붓고 약한 불에서 끓이며 성공의 향기를 음미합니다.

```
클래스 사람 { 공개: 템
    틀릿< typename T,
    typename =
        std::enable_if_t<!std::is_base_of<Person, T>>::value &&
        !std::is_integral<std::remove_reference_t<T>>::값
    >
    >

    명시적 Person(T&& n) : // std::strings 및 // std::strings로 변환 가능
        name(std::forward<T>(n)) { … } // 가능한 // args에 대한 ctor

    명시적 Person(int idx) : // 정수 인수에 대한 ctor
        name(nameFromIdx(idx)) { … }

    … // ctor 등을 복사하고 이동합니다.
}

개인: 표준::
문자열 이름; ];
```

짜잔! 아름다움의 것! 글쎄요, 그 아름다움은 템플릿 메타프로그래밍 패티쉬를 가진 사람들에게 가장 두드러질 것입니다. 그러나 사실은 이 접근 방식이 작업을 완료할 뿐만 아니라 고유한 여유를 가지고 수행한다는 사실입니다. 퍼펙트 포워딩을 사용하기 때문에 효율성이 극대화되고, 범용 참조와 오버로딩의 조합을 금지하지 않고 제어하기 때문에 오버로딩이 불가피한 상황(예: 생성자)에 이 기법을 적용할 수 있다.

절충안

이 항목에서 고려되는 처음 세 가지 기술(오버로딩 포기, const T& 전달, 값 전달)은 호출할 함수의 각 매개변수에 대한 유형을 지정합니다. 마지막 두 가지 기술인 태그 디스패치 및 템플릿 적합성 제한은 완전 전달을 사용하므로 매개변수 유형을 지정하지 않습니다. 유형을 지정하거나 지정하지 않는 이 근본적인 결정에는 결과가 따릅니다.

일반적으로 완전 전달은 매개변수 선언의 유형을 따르기 위한 목적으로만 임시 개체를 생성하지 않기 때문에 더 효율적입니다. Person 생성자의 경우, 완전 전달은 "Nancy"와 같은 문자열 리터럴이 Person 내부의 std::string에 대한 생성자로 전달되는 것을 허용하는 반면, 완전 전달을 사용하지 않는 기술은 임시 std::string을 생성해야 합니다. Person 생성자에 대한 매개 변수 사양을 충족하기 위해 문자열 리터럴에서 개체를 가져옵니다.

그러나 완벽한 포워딩에는 단점이 있습니다. 하나는 특정 유형을 사용하는 함수에 전달할 수 있지만 일부 유형의 인수는 완벽하게 전달할 수 없다는 것입니다.

항목 30은 이러한 완벽한 전달 실패 사례를 탐구합니다.

두 번째 문제는 클라이언트가 잘못된 인수를 전달할 때 오류 메시지를 이해할 수 있다는 것입니다. 예를 들어 Person 객체를 생성하는 클라이언트가 chars(표준: 문자열 구성):

사람 p(u"Konrad Zuse"); // "Konrad Zuse"는 // const char16_t 유형의 문자로
구성됩니다.

이 항목에서 조사한 처음 세 가지 접근 방식을 통해 컴파일러는 사용 가능한 생성자가 int 또는 std::string을 사용하는 것을 볼 수 있으며 const char16_t[12]에서 다음으로 변환의 없음을 설명하는 다소간 직접적인 오류 메시지를 생성합니다. int 또는 std::string.

그러나 완벽한 전달에 기반한 접근 방식을 사용하면 const char16_ts 배열이 불만 없이 생성자의 매개변수에 바인딩됩니다. 거기에서 그것은 Person의 std::string 데이터 멤버의 생성자로 전달되고, 그 시점에서만 호출자가 전달한 것(const char16_t 배열)과 필요한 것(std::string에 허용되는 모든 유형)이 일치하지 않습니다. 생성자)가 발견되었습니다. 결과 오류 메시지는 인상적일 수 있습니다.

내가 사용하는 컴파일러 중 하나를 사용하면 길이가 160줄 이상입니다.

이 예에서 범용 참조는 한 번만 전달되지만(Person 생성자에서 std::string 생성자로), 시스템이 복잡 할수록 범용 참조가 최종 도착하기 전에 여러 계층의 함수 호출을 통해 전달될 가능성이 더 높습니다. 인수 유형이 허용되는지 여부를 결정하는 사이트에서. 범용 참조가 전달되는 횟수가 많을수록 뭔가 잘못되었을 때 오류 메시지가 더 당황스러울 수 있습니다. 많은 개발자들은 이 문제만으로도 성능이 가장 중요한 인터페이스에 대한 범용 참조 매개변수를 예약할 근거라는 것을 알게 됩니다.

Person의 경우 전달 함수의 범용 참조 매개변수가 std::string에 대한 이니셜라이저여야 한다는 것을 알고 있으므로 다음을 사용할 수 있습니다.

`static_assert`를 사용하여 해당 역할을 수행할 수 있는지 확인합니다. `std::is_constructible` 유형 특성은 한 유형의 객체가 다른 유형(또는 유형 집합)의 객체(또는 객체 집합)에서 구성될 수 있는지 여부를 결정하기 위해 컴파일 타임 테스트를 수행하므로 어설션을 쉽게 수행할 수 있습니다. 쓰다:

```
클래스 사람 { 공개: 템
    플랫< typename T,
        typename =
            std::enable_if_t<!std::is_base_of<Person,
                std::decay_t<T>::value && !std::is_integral<std::remove_reference_t<T>>::값
            >
        >
    명시적 Person(T&& n):
        name(std::forward<T>(n)) {
            // std::string이 T 객체에서 생성될 수 있다고 주장합니다.
            static_assert( std::is_constructible<std::string, T>::value,
                "매개변수 n은 std::string을 구성하는 데 사용할 수 없습니다."
            );
            ...
            // 일반적인 ctor 작업은 여기로 이동합니다.
        }
        ...
        // Person 클래스의 나머지 부분(이전과 같이)
    };
}
```

이로 인해 클라이언트 코드가 `std::string`을 구성하는 데 사용할 수 없는 유형에서 `Person`을 만들려고 하면 지정된 오류 메시지가 생성됩니다. 불행히도 이 예에서 `static_assert`는 생성자의 본문에 있지만 멤버 초기화 목록의 일부인 전달 코드가 그 앞에 있습니다. 내가 사용하는 컴파일러를 사용하면 결과적으로 `static_assert`에서 발생하는 훌륭하고 읽기 쉬운 메시지가 일반적인 오류 메시지(최대 160 줄 이상)가 발생한 후에만 나타납니다.

기억해야 할 사항 • 범

용 참조와 오버로딩의 조합에 대한 대안에는 고유한 함수 이름 사용, lvalue 참조를 통해 매개변수를 const로 전달, 값으로 매개변수 전달, 태그 디스패치 사용 등이 있습니다.

- std::enable_if를 통해 템플릿을 제한하면 범용 참조와 오버로딩을 함께 사용할 수 있지만 컴파일러가 범용 참조 오버로드를 사용할 수 있는 조건을 제어합니다.
- 범용 참조 매개변수는 종종 효율성 이점이 있지만 일반적으로 사용성 단점이 있습니다.

항목 28: 참조 축소를 이해하십시오.

[항목 23](#)은 인수가 템플릿 함수에 전달될 때 템플릿 매개변수에 대해 추론된 유형이 인수가 lvalue인지 rvalue인지 여부를 인코딩한다고 말합니다. Item은 이것이 인수가 범용 참조인 매개변수를 초기화하는 데 사용될 때만 발생한다고 언급하지 않았지만 생략에 대한 합당한 이유가 있습니다: 범용 참조는 [항목 24](#) 까지 도입되지 않습니다. 함께, 범용에 대한 이러한 관찰 참조 및 lvalue/rvalue 인코딩은 이 템플릿의 경우,

```
template<typename T>
void func(T&& param);
```

추론된 템플릿 매개변수 T는 param에 전달된 인수가 lvalue인지 rvalue인지 여부를 인코딩합니다.

인코딩 메커니즘은 간단합니다. lvalue가 인수로 전달되면 T는 lvalue 참조로 추론됩니다. rvalue가 전달되면 T는 비참조로 추론됩니다. (비대칭성에 주의하십시오. lvalue는 lvalue 참조로 인코딩되지만 rvalue는 비참조로 인코딩됩니다.) 따라서:

위젯 위젯팩토리();	// rvalue를 반환하는 함수
위젯 w;	// 변수(lvalue)
함수(w);	// lvalue로 func를 호출합니다. T 추론 // Widget&
func(위젯팩토리());	// rvalue로 func를 호출합니다. T 추론 // 위젯 이 됨

func에 대한 두 호출 모두에서 위젯이 전달되지만 하나의 위젯은 lvalue이고 다른 하나는 rvalue이기 때문에 템플릿 매개변수 T에 대해 서로 다른 유형이 추론됩니다. 이것은 곧 보게 되겠지만 범용 참조가 rvalue가 될지 여부를 결정합니다. 참조 또는 lvalue 참조이며 std::forward가 작업을 수행하는 기본 메커니즘이기도 합니다.

std::forward 및 범용 참조를 더 자세히 살펴보기 전에 참조에 대한 참조가 C++에서 불법이라는 점에 유의해야 합니다. 하나를 선언하려고 하면 컴파일러에서 다음과 같이 질책할 것입니다.

```
정수 x;
...
자동& & rx = x; // 오류! 참조에 대한 참조를 선언할 수 없습니다.
```

그러나 범용 참조를 사용하는 함수 템플릿에 lvalue가 전달될 때 어떤 일이 발생하는지 생각해 보십시오.

```
template<typename T>
void func(T&& param);           // 이전과

함수(w);                      // lvalue로 func를 호출합니다.
// T는 Widget&으로 추론
```

T에 대해 추론된 유형(예: Widget&)을 사용하여 템플릿을 인스턴스화하면 다음을 얻습니다.

무효 함수(위젯& && 매개변수);

참조에 대한 참조! 그러나 컴파일러는 항의하지 않습니다. 항목 24에서 범용 참조 매개변수가 lvalue로 초기화되기 때문에 param의 유형은 lvalue 참조여야 하지만 T에 대해 추론된 유형을 가져와 템플릿에 대입한 결과를 컴파일러가 어떻게 얻습니까? 다음 중 궁극적인 가능 서명은 무엇입니까?

무효 함수(위젯& 매개변수);

정답은 참조 축소입니다. 예, 참조에 대한 참조를 선언하는 것은 금지되어 있지만 컴파일러는 템플릿 인스턴스화가 그 중 하나인 특정 컨텍스트에서 참조를 생성할 수 있습니다. 컴파일러가 참조에 대한 참조를 생성할 때 참조 축소는 다음에 일어날 일을 지시합니다.

두 종류의 참조(lvalue 및 rvalue)가 있으므로 네 가지 가능한 참조 참조 조합(lvalue에서 lvalue, lvalue에서 rvalue, rvalue에서 lvalue, rvalue에서 rvalue)이 있습니다. 참조에 대한 참조가 이것을 허용되는 컨텍스트에서 발생하면(예: 템플릿 인스턴스화 동안) 참조는 다음 규칙에 따라 단일 참조로 축소됩니다.

두 참조 중 하나가 lvalue 참조이면 결과는 lvalue 참조입니다.
그렇지 않으면(즉, 둘 다 rvalue 참조인 경우) 결과는 rvalue 참조입니다.
예스.

위의 예에서 추론된 유형 Widget&을 템플릿 func로 대체하면 lvalue 참조에 대한 rvalue 참조가 생성되고 참조 축소 규칙은 결과가 lvalue 참조임을 알려줍니다.

참조 축소는 std::forward 작동의 핵심 부분입니다. 항목 25에서 설명한 것처럼 std::forward는 범용 참조 매개변수에 적용되므로 일반적인 사용 사례는 다음과 같습니다.

템플릿<유형 이름 T> 무효

```
f(T&& fParam) {
```

// 일 좀해라

someFunc(std::forward<T>(fParam)); // fParam을 }로 전달 // someFunc

fParam은 범용 참조이기 때문에 유형 매개변수 T가 f에 전달된 인수(즉, fParam을 초기화하는 데 사용되는 표현식)가 lvalue인지 rvalue인지 여부를 인코딩한다는 것을 알고 있습니다. std::forward의 작업은 f에 전달된 인수가 rvalue인 경우, 즉 T가 비참조 유형인 경우 T가 인코딩하는 경우에만 fParam(lvalue)을 rvalue로 캐스팅하는 것입니다.

이를 위해 `std::forward`를 구현하는 방법은 다음과 같습니다.

이것은 표준에 적합하지 않지만(일부 인터페이스 세부 사항은 생략했습니다), 차이점은 `std::forward`가 어떻게 작동하는지 이해하기 위한 목적과 관련이 없습니다.

f에 전달된 인수가 Widget 유형의 lvalue라고 가정합니다. T는 Widget&으로 추론되고 std::forward에 대한 호출은 std::forward <Widget&>으로 인스턴스화됩니다. Widget&을 std::forward 구현에 연결하면 다음이 생성됩니다.

```
Widget& && forward(typename
    remove_reference<Widget&::type& param> { return
static_cast<Widget& &&>(param); }
```

유형 특성 std::remove_reference<Widget&::type>은 Widget을 생성합니다(참조 항목 9) 따라서 std::forward는 다음과 같이 됩니다.

```
Widget& && forward(Widget& param)
{ return static_cast<Widget& &&>(param); }
```

참조 축소는 반환 유형 및 캐스트에도 적용되며 결과는 호출에 대한 std::forward의 최종 버전입니다.

<pre>Widget& forward(Widget& param)</pre>	// 여전히 // 네임 스페이스 std에 있음
<pre>{ return static_cast<Widget& >(param); }</pre>	

보시다시피 lvalue 인수가 함수 템플릿 f에 전달되면 std::forward가 인스턴스화되어 lvalue 참조를 가져오고 반환합니다. 내부의 캐스트

std::forward는 param의 유형이 이미 Widget&이므로 아무 작업도 수행하지 않으므로 Widget&으로 캐스팅해도 아무 효과가 없습니다. 따라서 std::forward에 전달된 lvalue 인수는 lvalue 참조를 반환합니다. 정의에 따르면 lvalue 참조는 lvalue이므로 lvalue를 std::forward에 전달하면 예상대로 lvalue가 반환됩니다.

이제 f에 전달된 인수가 Widget 유형의 rvalue라고 가정합니다. 이 경우 f의 유형 매개변수 T에 대해 추론된 유형은 단순히 위젯입니다. 따라서 f 내부에서 std::forward에 대한 호출은 std::forward<Widget>에 대한 것입니다. std::forward 구현에서 T를 Widget으로 대체하면 다음과이 제공됩니다.

```
Widget&& forward(typename
    remove_reference<Widget>::type& param) { return
static_cast<Widget&&>(param); }
```

비참조 유형 Widget에 std::remove_reference를 적용하면 (Widget)으로 시작한 것과 동일한 유형이 생성되므로 std::forward는 다음과 같이 됩니다.

```
Widget&& forward(Widget& param)
{ return static_cast<Widget&&>(param); }
```

여기에 참조에 대한 참조가 없으므로 참조 축소가 없으며 이것이 호출에 대한 std::forward의 최종 인스턴스화된 버전입니다.

함수에서 반환된 Rvalue 참조는 rvalue로 정의되므로 이 경우 std::forward는 f의 매개변수 fParam(lvalue)을 rvalue로 바꿉니다. 최종 결과는 f에 전달된 rvalue 인수가 someFunc에 rvalue로 전달된다는 것입니다. 이는 정확히 발생해야 하는 일입니다.

C++14에서 std::remove_reference_t의 존재는 std::forward를 좀 더 간결하게 구현하는 것을 가능하게 합니다:

<pre>템플릿<유형이름 T> T&& 앞으로(remove_reference_t<T>& param) {</pre>	// C++14; 여전히 // 네임스 페이스 std에 있음
<pre>반환 static_cast<T&&>(param);</pre>	
<pre>}</pre>	

참조 축소는 네 가지 상황에서 발생합니다. 첫 번째이자 가장 일반적인 것은 템플릿 인스턴스화입니다. 두 번째는 자동 변수에 대한 유형 생성입니다. 자동 변수에 대한 유형 추론은 기본적으로 템플릿에 대한 유형 추론과 동일하기 때문에 세부 사항은 템플릿의 경우와 본질적으로 동일합니다(항목 2 참조). 항목의 앞부분에서 이 예를 다시 고려하십시오.

<pre>template<typename T> void func(T&& param);</pre>	
<pre>위젯 위젯팩토리();</pre>	// rvalue를 반환하는 함수
<pre>위젯 w;</pre>	// 변수(lvalue)
<pre>함수(w);</pre>	// lvalue로 func를 호출합니다. T 추론 // Widget&
<pre>func(위젯팩토리());</pre>	// rvalue로 func를 호출합니다. T 추론 // 위젯 이 됨

이것은 자동 형태로 모방될 수 있습니다. 선언

자동&& w1 = w;

w1을 lvalue로 초기화하므로 자동으로Widget& 유형을 추론합니다. w1에 대한 선언에서 auto용 Widget&을 연결하면 이 참조 참조 코드가 생성됩니다.

위젯& && w1 = w;

참조 축소 후 다음이 됩니다.

위젯& w1 = w;

결과적으로 w1은 lvalue 참조입니다.

한편 이 선언문은

자동&& w2 = 위젯팩토리();

w2를 rvalue로 초기화하여 비참조 유형 위젯이 자동으로 추론되도록 합니다. Widget을 auto로 대체하면 다음과 같은 결과를 얻을 수 있습니다.

```
위젯&& w2 = 위젯팩토리();
```

여기에 참조에 대한 참조가 없으므로 완료되었습니다. w2는 rvalue 참조입니다.

이제 항목 24에 소개된 범용 참조를 진정으로 이해할 수 있는 위치에 있습니다. 범용 참조는 새로운 종류의 참조가 아니라 실제로 두 가지 조건이 충족되는 컨텍스트에서 rvalue 참조입니다.

- 유형 연역은 lvalue와 rvalue를 구분합니다. 유형 T의 Lvalue는 유형 T&를 갖는 것으로 추론되는 반면 유형 T의 rvalue는 추론 유형으로 T를 산출합니다.
- 참조 축소가 발생합니다.

범용 참조의 개념은 참조 축소 컨텍스트의 존재를 인식하고, lvalue와 rvalue에 대해 다른 유형을 정신적으로 추론하고, 추론된 유형을 발생하는 상황.

네 가지 컨텍스트가 있다고 말했지만 템플릿 인스턴스화와 자동 유형 생성이라는 두 가지만 논의했습니다. 세 번째는 `typedef` 및 별칭 선언의 생성 및 사용입니다(항목 9 참조). `typedef`를 생성하거나 평가하는 동안 참조에 대한 참조가 발생하면 참조 축소가 개입하여 제거합니다. 예를 들어 rvalue 참조 유형에 대한 `typedef`가 포함된 위젯 클래스 템플릿이 있다고 가정합니다.

```
template<typename T> 클
래스 위젯 { public: typedef
    T&& RvalueRefToT;

    ...
};

};
```

lvalue 참조 유형으로 위젯을 인스턴스화한다고 가정합니다.

```
위젯<int&> w;
```

위젯 템플릿에서 T를 `int&`로 대체하면 다음과 같은 `typedef`가 제공됩니다.

```
typedef int& && RvalueRefToT;
```

참조 축소는 이것으로 줄입니다.

```
typedef int& RvalueRefToT;
```

이것은 우리가 `typedef`에 대해 선택한 이름이 우리가 기대했던 것만큼 설명적이지 않을 수 있음을 분명히 합니다. `RvalueRefToT`는 위젯이 lvalue 참조 유형으로 인스턴스화될 때 lvalue 참조에 대한 `typedef`입니다.

참조 축소가 발생하는 마지막 컨텍스트는 decltype을 사용하는 것입니다. decltype과 관련된 유형을 분석하는 동안 참조에 대한 참조가 발생하면 참조 축소가 시작되어 이를 제거합니다. (decltype에 대한 정보는 항목 3을 참조하십시오.)

기억할 사항 • 참조 축소

소는 템플릿 인스턴스화, 자동 유형 생성, typedef 및 별칭 선언의 생성 및 사용, decltype의 네 가지 컨텍스트에서 발생합니다.

- 컴파일러가 참조 축소 컨텍스트에서 참조에 대한 참조를 생성하면 결과가 단일 참조가 됩니다. 원래 참조 중 하나가 lvalue 참조이면 결과는 lvalue 참조입니다. 그렇지 않으면 rvalue 참조입니다.
- 범용 참조는 유형 추론이 lvalue를 rvalue와 구별하고 참조 축소가 발생하는 컨텍스트에서 rvalue 참조입니다.

항목 29: 이동 작업이 존재하지 않고 저렴하지도 않고 사용되지도 않는다고 가정합니다.

이동 의미 체계는 틀림없이 C++11의 최고의 기능입니다. "컨테이너를 옮기는 것이 이제 포인터를 복사하는 것만큼 저렴합니다!" "임시 개체를 복사하는 것이 이제 매우 효율적입니다. 이를 방지하기 위한 코딩은 조기 최적화와 같습니다!" 그러한 감정은 이해하기 쉽습니다. 이동 의미는 정말 중요한 기능입니다. 컴파일러가 값비싼 복사 작업을 비교적 저렴한 이동으로 교체할 수 있을 뿐만 아니라 실제로 수행해야 합니다(적절한 조건이 충족될 때). C++98 코드 기반을 사용하고 C++11 호환 컴파일러 및 표준 라이브러리로 다시 컴파일하면—shazam!—소프트웨어가 더 빠르게 실행됩니다.

이동 시맨틱은 이를 실제로 해낼 수 있으며, 이는 기능에 전설적인 가치가 있는 아우라를 부여합니다. 그러나 전설은 일반적으로 과장의 결과입니다. 이 항목의 목적은 기대치를 유지하는 것입니다.

많은 유형이 이동 의미 체계를 지원하지 못한다는 관찰부터 시작하겠습니다. 이동이 복사보다 빠르게 구현될 수 있는 유형에 대한 이동 작업을 추가하도록 C++11에 대해 전체 C++98 표준 라이브러리가 정밀 검사되었으며 이러한 작업을 활용하도록 라이브러리 구성 요소의 구현이 수정되었습니다. 그러나 C++11을 활용하도록 완전히 수정되지 않은 코드 기반으로 작업하고 있을 가능성이 있습니다. C++11에 대한 수정 사항이 없는 응용 프로그램(또는 사용하는 라이브러리)의 유형의 경우,

컴파일러의 이동 지원은 거의 도움이 되지 않을 것입니다. 사실, C++11은 이동 작업이 없는 클래스에 대해 이동 작업을 기꺼이 생성하지만 복사 작업, 이동 작업 또는 소멸자를 선언하지 않는 클래스에서만 발생합니다([항목 17 참조](#)). 이동을 비활성화한 유형의 데이터 멤버 또는 기본 클래스(예: 이동 작업 삭제 - [항목 11 참조](#))도 컴파일러 생성 이동 작업을 억제합니다. 이동에 대한 명시적 지원이 없고 컴파일러 생성 이동 작업에 적합하지 않은 유형의 경우 C++11이 C++98보다 성능 향상을 제공할 것으로 기대할 이유가 없습니다.

명시적 이동 지원이 있는 유형이라도 원하는 만큼의 이점을 얻지 못할 수 있습니다. 예를 들어 표준 C++11 라이브러리의 모든 컨테이너는 이동을 지원하지만 모든 컨테이너를 이동하는 것이 저렴하다고 가정하는 것은 실수입니다. 일부 컨테이너의 경우 내용물을 이동할 수 있는 정말 저렴한 방법이 없기 때문입니다. 다른 사람들에게는 컨테이너가 제공하는 정말 저렴한 이동 작업에 컨테이너 요소가 만족할 수 없는 경고가 있기 때문입니다.

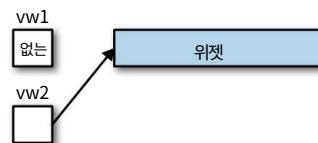
C++11의 새로운 컨테이너인 `std::array`를 고려하십시오. `std::array`는 본질적으로 STL 인터페이스가 있는 내장 배열입니다. 이것은 각각의 내용을 힙에 저장하는 다른 표준 컨테이너와 근본적으로 다릅니다. 이러한 컨테이너 유형의 개체는 개념적으로 컨테이너의 내용을 저장하는 힙 메모리에 대한 포인터만 보유합니다(데이터 멤버로). (실제는 더 복잡하지만 이 분석의 목적상 차이점은 중요하지 않습니다.) 이 포인터가 있으면 전체 컨테이너의 내용을 일정한 시간에 이동할 수 있습니다. 소스 컨테이너를 대상으로 지정하고 소스의 포인터를 `null`로 설정합니다.

```
std::vector<위젯> vw1;
```

// vw1에 데이터를 넣습니다.

...

// vw1을 vw2로 이동합니다. // 일정한 시간
에 실행됩니다. // vw1과 vw2의 ptrs만 수정
됩니다. auto vw2 = std::move(vw1);



`std::array` 객체에는 이러한 포인터가 없습니다. `std::array` 내용에 대한 데이터가 `std::array` 객체에 직접 저장되기 때문입니다.

표준::배열<위젯, 10000> aw1;

// aw1에 데이터를 넣습니다.

...

// aw1을 aw2로 이동합니다. // 선형 시간으로 실행됩니다. // aw1의 모든 요소는 aw2로 이동합니다. auto aw2 = std::move(aw1);



aw1의 요소는 aw2로 이동됩니다. Widget이 복사보다 이동이 빠른 유형이라고 가정하면 Widget의 std::array를 이동하는 것이 동일한 std::array를 복사하는 것보다 빠릅니다. 따라서 std::array는 확실히 이동 지원을 제공합니다.

그러나 std::array를 이동하고 복사하는 것은 컨테이너의 각 요소를 복사하거나 이동해야 하기 때문에 선형 시간 계산 복잡성을 갖습니다. 이것은 "컨테이너를 옮기는 것이 이제 몇 개의 포인터를 할당하는 것 만큼 저렴하다"는 주장과 거리가 멀다.

반면에 std::string은 일정한 시간 이동과 선형 시간 복사를 제공합니다.

따라서 이동이 복사보다 빠른 것처럼 들리지만 그렇지 않을 수도 있습니다.

많은 문자열 구현에서 SSO(소형 문자열 최적화)를 사용합니다. SSO를 사용하면 "작은" 문자열(예: 용량이 15자 이하인 문자열)이 std::string 개체 내의 버퍼에 저장됩니다. 힙 할당 스토리지가 사용되지 않습니다.

SSO 기반 구현을 사용하여 작은 문자열을 이동하는 것은 복사보다 빠르지 않습니다. 일반적으로 복사본을 통한 이동의 성능 이점의 기초가 되는 포인터 전용 복사 트릭이 적용되지 않기 때문입니다.

SSO의 동기는 짧은 문자열이 많은 응용 프로그램의 표준이라는 광범위한 증거입니다. 이러한 문자열의 내용을 저장하기 위해 내부 버퍼를 사용하면 해당 문자열에 대한 메모리를 동적으로 할당할 필요가 없으며 이는 일반적으로 효율성이 높습니다. 그러나 승리의 의미는 이동이 복사보다 빠르지 않다는 것입니다. 그러나 유리 반으로 가득 찬 접근 방식을 취하고 그러한 문자열의 경우 복사가 이동보다 느리지 않다고 말할 수 있습니다.

빠른 이동 작업을 지원하는 유형의 경우에도 확실해 보이는 일부 이동 상황은 결국 복사본을 만들 수 있습니다. [항목 14](#)는 표준 라이브러리의 일부 컨테이너 작업이 강력한 예외 안전 보장을 제공하고 해당 보장에 종속된 레거시 C++98 코드가 C++11로 업그레이드할 때 손상되지 않도록 보장하기 위해 기본 복사 작업을 설명합니다. 이동 작업이 throw되지 않는 것으로 알려진 경우에만 이동 작업으로 대체될 수 있습니다. 결과적으로 유형이 해당 유형보다 더 효율적인 이동 작업을 제공하더라도

복사 작업을 수행하고 코드의 특정 지점에서 이동 작업이 일반적으로 적절하더라도(예: 소스 개체가 rvalue인 경우) 컴파일러는 여전히 복사 작업을 호출해야 할 수 있습니다. 이동 작업은 noexcept로 선언되지 않았습니다.

따라서 C++11의 이동 의미 체계가 도움이 되지 않는 몇 가지 시나리오가 있습니다.

- 이동 작업 없음: 이동할 개체가 이동 작업을 제공하지 못합니다.
일. 따라서 이동 요청은 복사 요청이 됩니다.
- 더 빨리 이동하지 않음: 이동할 개체에 이동 작업이 없는 이동 작업이 있습니다.
복사 작업보다 빠릅니다.
- 이동을 사용할 수 없음: 이동이 발생하는 컨텍스트에는 예외가 발생하지 않는 이동 작업이 필요하지만 해당 작업은 예외 없이 선언되지 않습니다.

이동 의미 체계가 효율성 향상을 제공하지 않는 또 다른 시나리오도 언급할 가치가 있습니다.

- 소스 객체가 lvalue임: 매우 적은 예외(예: 항목 25 참조)를 제외하고는 rvalue만 있습니다.
이동 작업의 소스로 사용될 수 있습니다.

그러나 이 항목의 제목은 이동 작업이 존재하지 않고 저렴하지 않고 사용되지 않는다고 가정하는 것입니다. 이것은 일반적으로 작업 중인 모든 유형을 알지 못하기 때문에 템플릿을 작성할 때와 같은 일반 코드의 경우입니다. 이러한 상황에서는 이동 의미 체계가 존재하기 전에 C++98에서와 같이 객체 복사에 대해 보수적이어야 합니다. 이는 "불안정한" 코드의 경우이기도 합니다. 즉, 사용되는 유형의 특성이 비교적 자주 수정되는 코드입니다.

그러나 종종 코드에서 사용하는 유형을 알고 변경되지 않는 특성(예: 저렴한 이동 작업을 지원하는지 여부)에 의존할 수 있습니다.

그런 경우에는 가정할 필요가 없습니다. 사용 중인 유형에 대한 이동 지원 세부 정보를 간단히 조회할 수 있습니다. 이러한 유형이 저렴한 이동 작업을 제공하고 이러한 이동 작업이 호출되는 컨텍스트에서 개체를 사용하는 경우 이동 의미 체계에 의존하여 복사 작업을 저렴한 이동 대응물로 대체할 수 있습니다.

기억해야 할 사항

- 이동 작업이 존재하지 않고 저렴하지 않으며 사용되지 않는다고 가정합니다. • 알려진 유형 또는 이동 의미 체계를 지원하는 코드에서는 다음이 필요하지 않습니다.
가정.

항목 30: 완벽한 전달 실패 사례를 숙지하십시오.

C++11 상자에 가장 두드러지게 새겨진 기능 중 하나는 완벽한 전달입니다. 완벽한 전달. 그것은 완벽! 아아, 상자를 열면 "완벽한"(이상)이 있고 "완벽한"(현실)이 있다는 것을 알게 될 것입니다. C++11의 완벽한 포워딩은 매우 훌륭하지만, 한두 앱실론을 간과할 수 있는 경우에만 진정한 완벽함을 얻을 수 있습니다. 이 항목은 앱실론에 익숙해지기 위한 것입니다.

애플론 탐색을 시작하기 전에 "완벽한 전달"이 무엇을 의미하는지 검토하는 것이 좋습니다. "전달"은 한 함수가 매개변수를 다른 함수로 전달하는 것을 의미합니다. 목표는 두 번째 함수(전달되는 함수)가 첫 번째 함수(전달을 수행하는 함수)가 받은 것과 동일한 객체를 받는 것입니다. 원래 호출자가 전달한 것의 복사본이기 때문에 값에 의한 매개변수를 배제합니다. 우리는 전달된 함수가 원래 전달된 객체와 함께 작동할 수 있기를 원합니다. 호출자가 강제로 포인터를 전달하도록 하고 싶지 않기 때문에 포인터 매개변수도 제외됩니다. 범용 전달에 관해서는 참조 매개변수를 다룰 것입니다.

완벽한 전달은 우리가 객체를 전달할 뿐만 아니라 객체의 유형, lvalue이든 rvalue이든, const 또는 volatile인지 여부와 같은 두드러진 특성도 전달한다는 것을 의미합니다. 참조 매개변수를 다룰 것이라는 관찰과 함께 이것은 범용 참조를 사용할 것임을 의미합니다(항목 24 참조). 왜냐하면 범용 참조 매개변수만 그들을.

어떤 함수 *f*가 있고 그것을 전달하는 함수(사실, 함수 템플릿)를 작성하고 싶다고 가정해 봅시다. 우리에게 필요한 핵심은 다음과 같습니다.

```
템플릿<유형명 T> 무효
fwd(T&& param) // 모든 인수 수락
{ f(std::forward<T>(param)); }
// f로 전달
```

전달 기능은 본질적으로 일반적입니다. 예를 들어, *fwd* 템플릿은 모든 유형의 인수를 허용하고 받는 모든 것을 전달합니다. 이 일반성의 논리적 확장은 전달 함수가 단순한 템플릿이 아니라 가변 템플릿이 되도록 하여 임의의 수의 인수를 허용하는 것입니다. *fwd*의 가변 형식은 다음과 같습니다.

```
template<typename... Ts> 무효
fwd(Ts&&... params) // 모든 인수 수락
```

```
f(std::forward<Ts>(params)...); } // f로 전달
```

이것은 표준 컨테이너의 배치 함수([항목 42 참조](#)) 와 스마트 포인터 팩토리 함수인 std::make_shared 및 std::make_unique([항목 21 참조](#))에서 볼 수 있는 형식입니다.

대상 함수 f와 전달 함수 fwd가 주어지면 특정 인수로 f를 호출하면 한 가지 작업을 수행하지만 동일한 인수로 fwd를 호출하면 다른 작업이 수행되면 완벽한 전달이 실패합니다.

<pre>f(식); fwd(표현식);</pre>	<pre>// 이것이 한 가지 일을 하는 경우 // 그 레나 이것이 다른 일을 하는 경우 fwd는 // 표현식 을 f로 완벽하 게 전달하는 데 실패합니다.</pre>
--------------------------------	--

여러 종류의 논쟁이 이러한 종류의 실패로 이어집니다. 그것들이 무엇인지 그리고 어떻게 해결해야 하지 아는 것이 중요하므로 완벽하게 전달할 수 없는 논쟁의 종류를 살펴보겠습니다.

중괄호 이니셜라이저

f가 다음과 같이 선언되었다고 가정합니다.

```
무효 f(const std::vector<int>& v);
```

이 경우 중괄호 이니셜라이저를 사용하여 f를 호출하면 컴파일됩니다.

<pre>f({ 1,2,3 });</pre>	<pre>// 좋아요, "{1, 2, 3}" 암시적으로 //</pre>
--------------------------	---

std::vector<int>로 변환됨

그러나 fwd에 동일한 중괄호 이니셜라이저를 전달하면 컴파일되지 않습니다.

<pre>fwd({ 1,2,3 });</pre>	<pre>// 오류! 컴파일하지 않는다</pre>
----------------------------	-----------------------------

이는 중괄호 이니셜라이저를 사용하는 것이 완벽한 전달 실패 사례이기 때문입니다.

이러한 모든 실패 사례에는 동일한 원인이 있습니다. f(예: f({ 1, 2, 3 }))에 대한 직접 호출에서 컴파일러는 호출 사이트에서 전달된 인수를 보고 f에 의해 선언된 매개변수의 유형을 봅니다. 호출 사이트의 인수를 매개변수 선언과 비교하여 호환되는지 확인하고 필요한 경우 암시적 변환을 수행하여 호출을 성공시킵니다. 위의 예에서 그들은 f의 매개변수 v가 바인딩할 std::vector<int> 객체를 갖도록 { 1, 2, 3 }에서 임시 std::vector<int> 객체를 생성합니다.

전달 함수 템플릿 fwd를 통해 간접적으로 f를 호출할 때 컴파일러는 더 이상 fwd의 호출 사이트에서 전달된 인수를 f의 매개변수 선언과 비교하지 않습니다. 대신 fwd에 전달되는 인수의 유형을 추론하고

그들은 추론된 유형을 f의 매개변수 선언과 비교합니다. 다음 중 하나가 발생하면 퍼펙트 포워딩이 실패합니다.

- 컴파일러는 하나 이상의 fwd 매개변수에 대한 유형을 추론할 수 없습니다. 이 경우 코드가 컴파일되지 않습니다.
- 컴파일러는 하나 이상의 fwd 매개변수에 대해 "잘못된" 유형을 추론합니다. 여기서 "잘못된"은 fwd의 인스턴스화가 추론된 유형으로 컴파일되지 않음을 의미할 수 있지만 fwd의 추론된 유형을 사용하여 f에 대한 호출이 fwd. 이러한 분기 동작의 한 가지 원인은 f가 오버로드된 함수 이름이고 "잘못된" 형식 추론으로 인해 fwd 내부에서 호출된 f의 오버로드가 f가 직접 호출되는 경우 호출되는 오버로드와 다른 경우입니다.

위의 "fwd({ 1, 2, 3 })" 호출에서 문제는 std::initializer_list로 선언되지 않은 함수 템플릿 매개변수에 중괄호 이니셜라이저를 전달하는 것이 표준에 따르면 그것은 "연역되지 않은 컨텍스트"입니다. 일반 영어에서 이는 fwd의 매개변수가 std::initializer_list로 선언되지 않았기 때문에 컴파일러가 fwd에 대한 호출에서 표현식 { 1, 2, 3 }에 대한 유형을 추론하는 것이 금지되어 있음을 의미합니다. fwd의 매개변수에 대한 유형을 추론하는 것을 방지하기 위해 컴파일러는 당연히 호출을 거부해야 합니다.

흥미롭게도 항목 2 는 중괄호 이니셜라이저로 초기화된 자동 변수에 대해 유형 추론이 성공한다고 설명합니다. 이러한 변수는 std::initializer_list로 간주됩니다.

initializer_list 자체를 사용하여 전달 함수가 추론해야 하는 유형이 std::initializer_list인 경우에 간단한 해결 방법을 제공합니다. auto를 사용하여 로컬 변수를 선언한 다음 전달 함수에 로컬 변수를 전달합니다.

자동 il = { 1, 2, 3 };	// il의 유형은 // std::initializer_list<int> 로 추론됨
fwd(il);	// il에서 f로 완벽하게 전달

널 포인터로 0 또는 NULL

항목 8 은 템플릿에 대한 널 포인터로 0 또는 NULL을 전달하려고 하면 유형 추론이 잘못되어 전달하는 인수에 대한 포인터 유형 대신 정수 유형(일반적으로 int)을 추론한다고 설명합니다. 결과는 0이나 NULL 모두 널 포인터로 완벽하게 전달할 수 없다는 것입니다. 그러나 수정은 쉽습니다. 0 또는 NULL 대신 nullptr을 전달하십시오. 자세한 내용은 항목 8을 참조하십시오.

선언 전용 정수형 정적 const 데이터 멤버 일반적으로 클래

스에 정수형 정적 const 데이터 멤버를 정의할 필요가 없습니다. 선언만으로도 충분합니다. 이는 컴파일러가 그러한 멤버의 값에 대해 const 전파를 수행하여 메모리를 따로 확보할 필요가 없기 때문입니다. 예를 들어 다음 코드를 고려하십시오.

```
클래스 위젯 { 공개: 정
    적 const std::size_t
        MinVals = 28; // MinVals' 선언
        ...
    };
    ...
    표준::벡터<int> 위젯데이터;
    widgetData.reserve(위젯::MinVals); // MinVals 사용
    ...
}
```

// defn이 없습니다. MinVals용

여기에서는 MinVals에 정의가 부족하더라도 위젯 Data의 초기 용량을 지정하기 위해 Widget::MinVals(이하 간단히 MinVals)를 사용합니다. 컴파일러는 MinVals가 언급된 모든 위치에 값 28을 배치하여 누락된 정의를 해결합니다(필요한 대로). MinVals의 값을 위해 따로 할당된 스토리지가 없다는 사실은 문제가 되지 않습니다. MinVals의 주소를 사용하는 경우(예: 누군가 MinVals에 대한 포인터를 만든 경우) MinVals는 스토리지가 필요하고(포인터가 가리킬 것이 있도록) 위의 코드는 컴파일되지만 다음에서 실패합니다. MinVals에 대한 정의가 제공될 때까지의 링크 시간입니다.

이를 염두에 두고 f(fwd가 인수를 전달하는 함수)가 다음과 같이 선언된다고 상상해 보십시오.

무효 f(std::size_t val);

컴파일러가 MinVals를 해당 값으로 대체하기 때문에 MinVals로 f를 호출하는 것은 괜찮습니다.

f(위젯::MinVals); // 괜찮음, "f(28)"로 처리됨

아아, fwd를 통해 f를 호출하려고 하면 일이 순조롭게 진행되지 않을 수 있습니다.

fwd(위젯::MinVals); // 오류! 링크하면 안됩니다

이 코드는 컴파일되지만 링크되어서는 안 됩니다. MinVals의 주소를 사용하는 코드를 작성하면 어떤 일이 발생하는지 생각나게 하는 경우 기본 문제가 동일하기 때문에 괜찮습니다.

소스 코드의 어떤 것도 MinVals의 주소를 사용하지 않지만 fwd의 매개변수는 범용 참조이며 컴파일러에서 생성된 코드의 참조는 일반적으로 포인터처럼 취급됩니다. 프로그램의 기본 이진 코드(및 하드웨어)에서

포인터와 참조는 본질적으로 같은 것입니다. 이 수준에서 참조는 자동으로 역참조되는 포인터라는 격언이 사실입니다.

이 경우 MinVals를 참조로 전달하는 것은 포인터로 전달하는 것과 사실상 동일하므로 포인터가 가리킬 메모리가 있어야 합니다.

통합 정적 const 데이터 멤버를 참조로 전달하려면 일반적으로 해당 멤버를 정의해야 하며 이 요구사항으로 인해 완전 전달이 없는 동등한 코드가 성공한 경우 완전 전달을 사용하는 코드가 실패할 수 있습니다.

그러나 앞의 토론에서 내가 뿐만 족제비 단어를 눈치채셨을 것입니다. 코드는 링크하지 않아야 합니다. 참조는 "보통" 포인터처럼 취급됩니다. "일반적으로" 참조로 통합 정적 const 데이터 멤버를 전달하려면 해당 멤버를 정의해야 합니다. 너에게 별로 말하고 싶지 않은 것을 아는 것 같아…

내가 하기 때문이다. 표준에 따르면 MinVals를 참조로 전달하려면 정의해야 합니다. 그러나 모든 구현이 이 요구 사항을 적용하는 것은 아닙니다. 따라서 컴파일러와 링커에 따라 정의되지 않은 통합 정적 const 데이터 멤버를 완벽하게 전달할 수 있음을 알 수 있습니다. 그렇게 하면 축하하지만 그러한 코드가 이식될 것이라고 기대할 이유가 없습니다. 이식성 있게 만들려면 문제의 정수 static const 데이터 멤버에 대한 정의를 제공하기만 하면 됩니다. MinVals의 경우 다음과 같습니다.

```
const std::size_t 위젯::MinVals; // 위젯의 .cpp 파일에서
```

정의는 이니셜라이저(MinVals의 경우 28)를 반복하지 않습니다.

그러나 이 세부 사항에 대해 강조하지 마십시오. 이니셜라이저를 잊어버리고 두 위치에 모두 제공하면 컴파일러에서 불평하므로 한 번만 지정하라는 메시지가 표시됩니다.

오버로드된 함수 이름 및 템플릿 이름 fwd를 통해 인수를 계속 전달하는 함수 f가 일부 작업을 수행하는 함수를 전달하여 동작을 사용자 정의할 수 있다고 가정합니다.

이 함수가 int를 취하고 반환한다고 가정하면 f는 다음과 같이 선언될 수 있습니다.

```
무효 f(int* pf)(int); // pf = "함수 처리"
```

f는 포인터가 아닌 더 간단한 구문을 사용하여 선언할 수도 있습니다. 이러한 선언은 위의 선언과 동일한 의미를 가지지만 다음과 같습니다.

```
무효 f(int pf(int)); // 위와 같은 f를 선언
```

어느 쪽이든, 이제 오버로드된 함수 processVal이 있다고 가정합니다.

```
int processVal(int 값); int  
processVal(int 값, int 우선순위);
```

f에 processVal을 전달할 수 있습니다.

f(프로세스 밸); // 좋아

하지만 우리가 할 수 있다는 것은 놀라운 일입니다. f는 함수에 대한 포인터를 인수로 요구하지만 processVal은 함수 포인터나 함수가 아니라 두 개의 서로 다른 함수의 이름입니다. 그러나 컴파일러는 어떤 processVal이 필요한지 알고 있습니다. f의 매개변수 유형과 일치하는 것입니다. 따라서 그들은 하나의 int를 취하는 processVal을 선택하고 해당 함수의 주소를 f에 전달합니다.

이 작업을 수행하는 이유는 f의 선언을 통해 컴파일러에서 필요한 processVal 버전을 파악할 수 있다는 것입니다. 그러나 fwd는 함수 템플릿이므로 필요한 유형에 대한 정보가 없으므로 컴파일러가 전달해야 하는 오버로드를 결정할 수 없습니다.

fwd(프로세스 밸); // 오류! 어떤 processVal?

processVal에는 유형이 없습니다. 유형이 없으면 유형 연역이 있을 수 없고 유형 연역이 없으면 또 다른 완벽한 전달 실패 사례가 남습니다.

오버로드된 함수 이름 대신(또는 추가로) 함수 템플릿을 사용하려고 하면 동일한 문제가 발생합니다. 기능 템플릿은 하나의 기능을 나타내는 것이 아니라 많은 기능을 나타냅니다.

```
template<typename T> T
workOnVal(T 매개변수) { … } // 값 처리를 위한 템플릿
```

fwd(workOnVal); // 오류! 어떤 workOnVal // 인스턴스화?

오버로드된 함수 이름이나 템플릿 이름을 수락하기 위해 fwd와 같은 완벽한 전달 함수를 얻는 방법은 전달하려는 오버로드 또는 인스턴스화를 수동으로 지정하는 것입니다. 예를 들어, f의 매개변수와 동일한 유형의 함수 포인터를 만들고 processVal 또는 workOnVal로 해당 포인터를 초기화하고 (따라서 적절한 버전의 processVal이 선택되거나 workOnVal의 적절한 인스턴스가 생성되도록 함) 다음을 전달할 수 있습니다. fwd에 대한 포인터:

```
ProcessFuncType 사용 = int (*) // typedef를 만든다. //
(int); // 항목 9 참조
```

```
ProcessFuncType processValPtr = processVal; // processVal에 // 필요
                                            한 서명을 지정합니다.
```

fwd(processValPtr); // 좋아

fwd(정적_캐스트<ProcessFuncType>(workOnVal)); // 역시 좋다

물론 이를 위해서는 fwd가 전달하는 함수 포인터의 유형을 알아야 합니다. 완전 전달 기능이 이를 문서화할 것이라고 가정하는 것은 불합리하지 않습니다. 결국, 완전 전달 함수는 무엇이든 받아들이도록 설계되었으므로 전달할 내용을 알려주는 문서가 없으면 어떻게 알 수 있습니까?

비트필드

완벽한 전달의 마지막 실패 사례는 비트 필드가 함수 인수로 사용되는 경우입니다. 이것이 실제로 무엇을 의미하는지 보려면 IPv4 헤더가 다음과 같이 모델링될 수 있음을 관찰하십시오.³

```
struct IPv4Header
{
    std::uint32_t 버전:4,
    IHL:4,
    DSCP:6,
    ECN:2,
    총 길이:16;
    ...
};
```

오래 참는 함수 f(포워딩 함수 fwd의 영원한 목표)가 std::size_t 매개변수를 취하도록 선언된 경우 이를 호출하면 IPv4Header 객체의 totalLength 필드가 소란 없이 컴파일됩니다.

```
무효 f(std::size_t sz); // 호출할 함수
```

```
IPv4헤더 h;
...
f(h.totalLength); // 좋아
```

그러나 fwd를 통해 h.totalLength를 f로 전달하는 것은 다른 이야기입니다.

```
fwd(h.totalLength); // 오류!
```

문제는 fwd의 매개변수가 참조이고 h.totalLength가 비 const 비트필드라는 것입니다. 그렇게 나쁘게 들리지 않을 수도 있지만 C++ 표준은 비정상적으로 명확한 산문에서 조합을 비난합니다. "비 const 참조는 비트 필드에 바인딩되지 않습니다." 금지에 대한 훌륭한 이유가 있습니다. 비트필드는 기계어의 임의의 부분(예: 32비트 int의 비트 3-5)으로 구성될 수 있지만 이러한 것을 직접 처리할 방법은 없습니다. 앞서 언급한 참조와 포인터는 하드웨어 수준에서 동일한 것이며 포인터를 생성할 방법이 없는 것과 마찬가지입니다.

³ 이것은 비트 필드가 lsb(최하위 비트)에서 msb(최상위 비트)로 배치된다고 가정합니다. C++는 이를 보장하지 않지만 컴파일러는 프로그래머가 비트필드 레이아웃을 제어할 수 있는 메커니즘을 제공하는 경우가 많습니다.

임의의 비트(C++에서는 가리킬 수 있는 가장 작은 것이 char임을 나타냄)에서 참조를 임의의 비트에 바인딩할 방법도 없습니다.

비트필드를 인수로 받아들이는 모든 함수가 비트필드 값의 복사본을 받는다는 것을 깨닫고 나면 비트필드를 완벽하게 전달하는 것이 불가능하다는 문제를 해결하는 것은 쉽습니다. 결국 어떤 함수도 비트필드에 대한 참조를 바인딩할 수 없으며 비트필드에 대한 포인터가 존재하지 않기 때문에 어떤 함수도 비트필드에 대한 포인터를 수락할 수 없습니다. 비트필드가 전달될 수 있는 유일한 종류의 매개변수는 값에 의한 매개변수와 흥미롭게도 const에 대한 참조입니다. 값에 의한 매개변수의 경우 호출된 함수는 분명히 비트필드에 있는 값의 복사본을 수신하며, const 매개변수에 대한 참조의 경우 표준은 참조가 실제로 일부 표준 정수 유형(예: int)의 개체에 저장된 비트 필드 값의 복사본. const에 대한 참조는 비트 필드에 바인딩되지 않으며 비트 필드 값이 복사된 "일반" 개체에 바인딩됩니다.

비트 필드를 완전 전달 함수에 전달하는 핵심은 전달 대상 함수가 항상 비트 필드 값의 복사본을 수신한다는 사실을 이용하는 것입니다. 따라서 직접 복사본을 만들고 복사본으로 전달 기능을 호출할 수 있습니다. IPv4Header가 있는 예제의 경우 이 코드가 트릭을 수행합니다.

```
// 비트 필드 값 복사; init에 대한 정보는 항목 6 을 참조하십시오 . 양식 자동 길이 =
static_cast<std::uint16_t>(h.totalLength);

fwd(길이); // 복사를 전달
```

결론 대부

분의 경우 퍼펙트 포워딩은 광고된 대로 정확하게 작동합니다. 그것에 대해 생각할 필요가 거의 없습니다. 그러나 작동하지 않을 때(합리적으로 보이는 코드가 컴파일에 실패하거나 더 심하게는 컴파일되지만 예상대로 동작하지 않을 때) 완벽한 포워딩의 불완전성에 대해 아는 것이 중요합니다. 마찬가지로 중요한 것은 문제를 해결하는 방법을 아는 것입니다. 대부분의 경우 이것은 간단합니다.

기억할 사항 • 템플릿

유형 추론 실패 또는 추론 시 완전 전달 실패
잘못된 유형.

- 완벽한 전달 실패로 이어지는 인수의 종류는 중괄호 이니셜라이저, 0 또는 NULL로 표현되는 null 포인터, 선언 전용 정수 const 정적 데이터 멤버, 템플릿 및 오버로드된 함수 이름, 비트 필드입니다.

6장

람다 표현식

람다 표현식(lambdas)은 C++ 프로그래밍의 판도를 바꾸는 도구입니다. 그것은 언어에 새로운 표현력을 가져오지 않기 때문에 다소 놀립습니다.

람다가 할 수 있는 모든 것은 타이핑을 조금 더 하면 손으로 할 수 있는 것입니다.

그러나 람다는 함수 개체를 생성하는 매우 편리한 방법이므로 일상적인 C++ 소프트웨어 개발에 미치는 영향은 엄청납니다. 람다가 없으면 STL "_if" 알고리즘(예: std::find_if, std::remove_if, std::count_if 등)은 가장 사소한 술어와 함께 사용되는 경향이 있지만 람다가 사용할 수 있는 경우 다음을 사용합니다. 사소하지 않은 조건을 가진 알고리즘이 꽂을 피웁니다. 비교 기능(예: std::sort, std::nth_element, std::lower_bound 등)으로 사용자 정의할 수 있는 알고리즘도 마찬가지입니다. STL 외부에서 람다는 std::unique_ptr 및 std::shared_ptr(항목 18 및 19 참조)에 대한 사용자 지정 삭제자를 빠르게 생성할 수 있게 하고 스레딩 API에서 조건 변수에 대한 조건자 사양을 똑같이 간단하게 만듭니다(항목 39 참조). 표준 라이브러리 외에도 람다는 콜백 함수, 인터페이스 적응 함수 및 일회성 호출에 대한 컨텍스트별 함수의 즉석 사양을 용이하게 합니다. Lambda는 실제로 C++를 더 즐거운 프로그래밍 언어로 만듭니다.

람다와 관련된 어휘는 혼란스러울 수 있습니다. 다음은 간단한 복습입니다.

- 람다 표현식은 표현식입니다. 소스 코드의 일부입니다. ~ 안에

```
std::find_if(container.begin(), container.end(), [](int val) { return  
    0 < val && val < 10; });
```

강조 표시된 표현식은 람다입니다.

- 클로저는 람다가 생성한 런타임 객체입니다. 캡처 모드에 따라 클로저는 캡처된 데이터의 복사본 또는 참조를 보유합니다. 예 대한 호출에서

위의 `std::find_if`에서 클로저는 런타임에 `std::find_if`에 대한 세 번째 인수로 전달되는 객체입니다.

- 클로저 클래스는 클로저가 인스턴스화되는 클래스입니다. 각 람다는 컴파일러가 고유한 클로저 클래스를 생성하도록 합니다. 람다 내부의 명령문은 클로저 클래스의 멤버 함수에서 실행 가능한 명령어가 됩니다.

람다는 함수에 대한 인수로만 사용되는 클로저를 만드는 데 자주 사용됩니다. 위의 `std::find_if` 호출의 경우입니다. 그러나 클로저는 일반적으로 복사될 수 있으므로 일반적으로 단일 람다에 해당하는 클로저 유형의 여러 클로저를 가질 수 있습니다. 예를 들어 다음 코드에서

```
{
    정수 x;                                     // x는 지역변수
    ...
    자동 c1 =                                     // c1은 y > 55의 복사본입니다. };//*
        [x](int y) { x를 반환                   * 클로저 생성 // 람다에 의해 생성됨
    ...
    자동 c2 = c1;                                // c2는 c1의 복사본입니다.
    자동 c3 = c2;                                // c3은 c2의 복사본입니다.
    ...
}
```

`c1`, `c2` 및 `c3`은 모두 람다가 생성한 클로저의 복사본입니다.

비공식적으로 람다, 클로저 및 클로저 클래스 사이의 경계를 흐리게 하는 것은 완벽하게 허용됩니다. 그러나 이어지는 항목에서는 컴파일 중에 존재하는 것(람다 및 클로저 클래스), 런타임에 존재하는 것(클로저), 서로 어떻게 관련되는지를 구별하는 것이 종종 중요합니다.

항목 31: 기본 캡처 모드를 피하세요.

C++11에는 참조 기준과 값 기준의 두 가지 기본 캡처 모드가 있습니다. 기본 참조별 캡처는 대그리고 참조로 이어질 수 있습니다. 기본 값에 의한 캡처는 당신이 그 문제에 면역이 있다고 생각하게 만들고(당신은 그렇지 않습니다), 당신의 클로저가 독립적이라고 생각하게 만듭니다(그렇지 않을 수도 있습니다).

이 항목에 대한 요약입니다. 당신이 임원보다 엔지니어에 더 가깝다면 그 뼈에 약간의 고기를 원할 것이므로 참조 캡처로 인한 디폴트 위험부터 시작하겠습니다.

참조에 의한 캡처는 클로저가 지역 변수에 대한 참조를 포함하도록 합니다.
 람다가 정의된 범위에서 사용할 수 있는 매개변수입니다. 만약 평생
 해당 람다에서 생성된 클로저가 지역 변수의 수명을 초과하거나
 매개변수가 없으면 클로저의 참조가 매달려 있습니다. 예를 들어,
 각각 int를 취하고 bool indi를 반환하는 필터링 함수의 컨테이너
 전달된 값이 필터를 충족하는지 여부 지정:

```
FilterContainer 사용 =
    std::vector<std::function<bool(int)>>;
// 항목 9 참조
// "사용", 항목 2
// 표준::함수의 경우
```

```
FilterContainer 필터; // 필터링 함수
```

다음과 같이 5의 배수에 대한 필터를 추가할 수 있습니다.

```
filter.emplace_back( [](int
    value) { 반환 값 % 5 == 0; } // 정보
);
// emplace_back
```

그러나 런타임에 제수를 계산해야 할 수도 있습니다.

5를 람다에 하드 코딩하십시오. 따라서 필터를 추가하면 다음과 같이 보일 수 있습니다.

```
무효 addDivisorFilter()
{
    자동 계산1 = 계산SomeValue1();
    자동 calc2 = 계산SomeValue2();

    자동 제수 = computeDivisor(calc1, calc2);

    필터.emplace_back(
        [&](int 값) { 반환 값 % 제수 == 0; } // 참조
    );
}
// 위험!
// 제수
// 할 것이다
// 매달아!
```

이 코드는 발생하기를 기다리는 문제입니다. 람다는 지역 변수를 나타냅니다.
 제수이지만 해당 변수는 addDivisorFilter가 반환될 때 더 이상 존재하지 않습니다. 그건
 filters.emplace_back이 반환된 직후에
 필터는 도착 시 기본적으로 작동하지 않습니다. 해당 필터를 사용하면 정의되지 않은 동작이 발생합니다.
 생성되는 순간부터.

이제 제수의 참조 기준 캡처가 명시적이면 동일한 문제가 존재합니다.

```
필터.emplace_back(
    [&divisor](int 값) { 반환 값 %
        제수 == 0; } );
// 위험! 참조
// 제수는
// 여전히 매달려 있습니다!
```

그러나 명시적인 캡처를 사용하면 람다의 실행 가능성이 의존한다는 것을 더 쉽게 알 수 있습니다. 제수의 수명에 덴트. 또한 "제수"라는 이름을 쓰는 것은 우리에게 다음을 상기시킵니다. 제수가 적어도 람다의 클로저만큼 살아 있는지 확인하십시오. 그게 더 특별한거야- 일반적인 "어떤 것도 매달려 있지 않은지 확인하십시오"라는 일반적인 훈계보다 "[&]"를 전달합니다.

클로저가 즉시 사용된다는 것을 알고 있는 경우(예: STL에 전달되어 알고리즘) 복사되지 않으며 보유하고 있는 참조가 오래 지속될 위험이 없습니다. 람다가 생성되는 환경의 로컬 변수 및 매개변수. ~ 안에 이 경우 참조가 매달릴 위험이 없으므로 기본 참조에 의한 캡처 모드를 피하십시오. 예를 들어 필터링 람다는 C++11의 std::all_of에 대한 인수로만 사용되며 모든 요소가 범위의 항목은 다음 조건을 충족합니다.

```
템플릿<유형 이름 C>
무효 workWithContainer(const C& 컨테이너)
{
    자동 계산1 = 계산SomeValue1(); 자동 calc2 = 계
    산SomeValue2();                                // 위와 같이
                                                // 위와 같이

    자동 제수 = computeDivisor(calc1, calc2); // 위와 같이

    ContElemT 사용 = typename C::value_type;      // 유형
                                                // 요소
                                                // 컨테이너

    std :: 시작을 사용하여;                      // 을 위한
    std::end를 사용하여;                        // 일반성;
                                                // 항목 13 참조

    if (std::all_of(
        begin(컨테이너), end(컨테이너), [&](const
        ContElemT& 값) { 반환 값 % 제수 == 0; } ) { // 모든 값이 있는 경우
                                                // 컨테이너에
                                                // 배수이다
                                                // 제수...
        ...
    } 또 다른 {                                // 그들은...
        ...
    }
}
```

이것은 안전하지만 안전이 다소 불안정한 것은 사실입니다. 람다가 발견되면 다른 컨텍스트에서 유용합니다(예: 필터에 추가될 기능으로 tainer) 및 해당 클로저가 div보다 오래 지속될 수 있는 컨텍스트에 복사하여 붙여넣었습니다.

그래서, 당신은 dangle-city로 돌아갈 것이고, 제수에 대한 평생 분석을 수행하도록 특별히 상기시키는 캡처 절에 아무 것도 없을 것입니다.

장기적으로 람다가 의존하는 지역 변수와 매개변수를 명시적으로 나열하는 것이 더 나은 소프트웨어 엔지니어링입니다.

그건 그렇고, C++14 람다 매개변수 사양에서 auto를 사용할 수 있다는 것은 C++14에서 위의 코드를 단순화 할 수 있다는 것을 의미합니다. ContElemT typedef는 제거될 수 있고 if 조건은 다음과 같이 수정될 수 있습니다:

```
if (std::all_of(begin(container), end(container), [&](const auto&
    value) { 반환 값 % 제수 == 0; })) // C++14
```

제수 문제를 해결하는 한 가지 방법은 기본 값별 캡처 모드입니다. 즉, 다음과 같이 필터에 람다를 추가할 수 있습니다.

<pre>필터.emplace_back([=](int 값) { 반환 값 % 제수 == 0; });</pre>	// 지 금 // 제수 // 할 수 없음 // 매달릴 수 없음
--	---

이 예에서는 이것으로 충분하지만 일반적으로 기본 값별 캡처는 여러분이 상상할 수 있는 안티 땅글링 엘리서가 아닙니다. 문제는 값으로 포인터를 캡처하는 경우 포인터를 람다에서 발생하는 클로저에 복사하지만 람다 외부의 코드에서 포인터를 삭제하고 복사본이 매달리는 것을 방지하지 못한다는 것입니다.

"절대 그런 일이 있을 수 없어!" 당신은 항의. "4 장 을 읽고 나는 스마트 포인터의 집에서 예배를 드린다. 때자 C++98 프로그래머만 원시 포인터를 사용하고 삭제합니다." 그것이 사실일 수도 있지만 실제로 원시 포인터를 사용하고 실제로는 사용자 아래에서 삭제될 수 있기 때문에 관련이 없습니다. 현대 C++ 프로그래밍 스타일에서는 소스 코드에 거의 표시되지 않는 경우가 많습니다.

위젯이 할 수 있는 일 중 하나가 필터 컨테이너에 항목을 추가하는 것이라고 가정합니다.

클래스 위젯 { 공개:

```
...
무효 addFilter() const; // ctor 등 // 필터에  
항목 추가
```

```
private: 정
수 제수; }; // 위젯의 필터에서 사용
```

Widget::addFilter는 다음과 같이 정의할 수 있습니다.

```
무효 위젯::addFilter() const { 필
터.emplace_back(
    [=](int 값) { 반환 값 % 제수 == 0; } );
}
```

행복한 초심자에게 이것은 안전한 코드처럼 보입니다. 람다는 제수에 의존하지만 기본 값별 캡처 모드는 제수가 람다에서 발생하는 모든 클로저에 복사되도록 보장합니다. 맞죠?

잘못된. 완전히 틀렸어. 끔찍하게 잘못되었습니다. 치명적으로 틀립니다.

캡처는 람다가 생성된 범위에서 볼 수 있는 비정적 로컬 변수(매개변수 포함)에만 적용됩니다. Widget::addFilter의 본문에서 divisor는 지역 변수가 아니라 Widget 클래스의 데이터 멤버입니다. 캡처할 수 없습니다. 그러나 기본 캡처 모드가 제거되면 코드가 컴파일되지 않습니다.

```
void Widget::addFilter() const
{ filters.emplace_back( // 오류! [](int
    value) { 반환 값 % divisor == 0; } // divisor );
    // 사용
    // 할 수 없습니다
}
```

더욱이, 명시적으로 제수를 캡처하려고 하면(값으로든 참조로든 상관없습니다), 제수가 지역 변수나 매개변수가 아니기 때문에 캡처가 컴파일되지 않습니다.

```
무효 위젯::addFilter() const
{ filters.emplace_back( [제수](int 값) { 반
    환 값 % 제수 == 0; } );
    // 오류! 캡처할 로컬 // 제수
    // 없음
}
```

따라서 기본 값별 캡처 절이 제수를 캡처하지 않고 기본 값별 캡처 절이 없으면 코드가 컴파일되지 않습니다. 무슨 일이 일어나고 있습니까?

설명은 원시 포인터의 암시적 사용에 달려 있습니다. this. 모든 비정적 멤버 함수에는 this 포인터가 있으며 클래스의 데이터 멤버를 언급할 때마다 이 포인터를 사용합니다. 예를 들어 모든 위젯 멤버 함수 내에서 컴파일러는 내부적으로 제수 사용을 this->divisor로 바꿉니다. 기본 값별 캡처가 있는 Widget::addFilter 버전에서,

```
무효 위젯::addFilter() const { 필
터.emplace_back(
    [=](int 값) { 반환 값 % 제수 == 0; } );
}
```

캡처되는 것은 제수가 아니라 위젯의 this 포인터입니다. 컴파일러는 코드가 다음과 같이 작성된 것처럼 취급합니다.

```
무효 위젯::addFilter() const {
    자동 currentObjectPtr = 이것;
    필터.emplace_back(
        [currentObjectPtr](int 값) { 반환 값 %
            currentObjectPtr->제수 == 0; } );
}
```

이것을 이해하는 것은 이 람다에서 발생하는 클로저의 실행 가능성이 이 포인터에 복사본이 포함된 위젯의 수명과 관련이 있다는 것을 이해하는 것과 같습니다. 특히 4 장에 따라 스마트 품종의 포인터만 사용하는 다음 코드를 고려하십시오.

```
FilterContainer 사용 =                                     // 이전과
    std::vector<std::function<bool(int)>>;
FilterContainer 필터;                                     // 이전과

무효 doSomeWork() {
    자동 pw =                                         // 위젯 생성; 보다
        std::make_unique<위젯>(); // 항목 21 for // std::make_unique

    pw->addFilter();                                // 사용하는 필터 추가
    // 위젯::제수
    ...
}
// 위젯을 파괴합니다. 필터 // 이제 댱글링
포인터를 보유합니다!
```

doSomeWork에 대한 호출이 만들어지면 std::make_unique에 의해 생성된 Widget 객체, 즉 해당 Widget에 대한 포인터의 복사본을 포함하는 필터에 의존하는 필터가 생성됩니다. Widget의 this 포인터입니다. 이 필터는 필터에 추가되지만 doSomeWork가 완료되면 위젯을 관리하는 std::unique_ptr에 의해 위젯이 파괴됩니다.

수명(항목 18 참조). 그 시점부터 필터에는 데人格링 포인터가 있는 항목이 포함됩니다.

이 특정 문제는 캡처하려는 데이터 멤버의 로컬 복사본을 만든 다음 복사본을 캡처하여 해결할 수 있습니다.

```
무효 위젯::addFilter() const {
    자동 제수복사 = 제수;                                     // 데이터 멤버 복사
    필터.emplace_back(
        [divisorCopy](int 값) { 반환 값 % divisorCopy == 0; } );
}
}
```

솔직히 말해서 이 접근 방식을 취하면 기본 값별 캡처도 작동합니다.

```
무효 위젯::addFilter() const { auto divisorCopy = 제수;
    filter.emplace_back( [=](int 값) { 반환 값 % divisorCopy == 0; } );
}
}
```

하지만 왜 운명을 유혹하는가? 기본 캡처 모드는 처음에 제수를 캡처한다고 생각했을 때 실수로 캡처할 수 있게 해 주는 모드입니다.

C++14에서 데이터 멤버를 캡처하는 더 좋은 방법은 일반화된 람다 캡처를 사용하는 것입니다(항목 32 참조).

```
void Widget::addFilter() const
{ filters.emplace_back( [divisor = divisor]
    (int value) { 반환 값 % divisor == 0; } // 클로저에 제수 복사
복사본 사용 );
}
}
```

그러나 일반화된 람다 캡처를 위한 기본 캡처 모드와 같은 것은 없으므로 C++14에서도 기본 캡처 모드를 피하기 위한 이 항목의 조언은 유효합니다.

기본 값 기반 캡처의 또 다른 단점은 해당 클로저가 자체 포함되고 외부 데이터 변경으로부터 격리 된다는 것을 제안할 수 있다는 것입니다.

폐쇄. 일반적으로 람다는 로컬 변수와 매개변수(캡처될 수 있음)뿐만 아니라 정적 저장 기간이 있는 객체에도 종속될 수 있기 때문에 사실이 아닙니다. 이러한 개체는 전역 또는 네임스페이스 범위에서 정의되거나 클래스, 함수 또는 파일 내에서 정적으로 선언됩니다. 이러한 개체는 람다 내에서 사용할 수 있지만 캡처할 수는 없습니다. 그러나 기본 값별 캡처 모드를 지정하면 실제와 같은 인상을 줄 수 있습니다. 이전에 본 add DivisorFilter 함수의 수정된 버전을 고려하십시오.

```
무효 addDivisorFilter() {
    정적 자동 calc1 = 계산SomeValue1(); 정적 자동 calc2 = 계산
    SomeValue2();

    정적 자동 제수 =
        계산제수(calc1, calc2);

    filter.emplace_back( [=](int
        value) // 아무것도 캡처하지 않습니다! { return value % divisor == 0; } // 위를 참조
        static );

    ++제수; // 제수 수정
}
```

이 코드의 일반 독자는 "[=]"를 보고 "좋아, 람다가 사용하는 모든 개체의 복사본을 만들고 따라서 독립적입니다."라고 생각하는 것을 용서할 수 있습니다. 그러나 그것은 자급 자족하지 않습니다. 이 람다는 비정적 지역 변수를 사용하지 않으므로 아무 것도 캡처되지 않습니다. 오히려, 람다에 대한 코드는 정적 변수 제수를 참조합니다. addDivisorFilter 호출이 끝날 때마다 제수가 증가하면 이 함수를 통해 필터에 추가된 람다가 새로운 동작(새로운 제수의 값에 해당)을 나타냅니다. 실질적으로 말해서, 이 람다는 참조로 제수를 캡처합니다. 이는 기본 값 캡처 절이 의미하는 것과 직접적인 모순입니다. 기본값으로 값 캡처 절을 사용하지 않으면 이러한 방식으로 코드를 잘못 읽을 위험이 없습니다.

기억해야 할 사항

- 기본 참조별 캡처는 맹글링 참조로 이어질 수 있습니다. • 기본 값별 캡처는 맹글링 포인터(특히 이것)에 취약합니다.
- 그리고 그것은 람다가 독립적이라고 오해의 소지가 있습니다.

항목 32: 객체를 클로저로 이동하려면 초기화 캡처를 사용하세요.

때로는 값에 의한 캡처도 참조에 의한 캡처도 원하는 것이 아닙니다. 클로저에 들어가고 싶은 이동 전용 객체(예: std::unique_ptr 또는 std::future)가 있는 경우 C++11은 이를 수행할 방법이 없습니다. 복사 비용은 비싸지만 이동 비용은 저렴한 객체(예: 표준 라이브러리의 대부분의 컨테이너)가 있고 해당 객체를 클로저에 넣고 싶다면 복사하는 것보다 옮기는 것이 훨씬 좋습니다. 그러나 C++11에서 이를 수행할 방법이 없습니다.

그러나 그것은 C++11입니다. C++14는 다른 이야기입니다. 객체를 클로저로 이동하는 것을 직접 지원합니다. 컴파일러가 C++14와 호환되는 경우 기뻐하고 계속 읽으십시오. 여전히 C++11 컴파일러로 작업하고 있다면 C++11에 대략적인 이동 캡처 방법이 있기 때문에 계속해서 읽어야 합니다.

C++11을 도입하면서도 움직임 캡처가 없다는 점은 단점으로 인식됐다. 직접적인 해결책은 C++14에 추가하는 것이지만 표준화 위원회는 다른 경로를 선택했습니다. 그들은 매우 유연한 새로운 캡처 메커니즘을 도입했습니다. 움직임에 따른 캡처는 수행할 수 있는 트릭 중 하나일 뿐입니다. 새로운 기능을 초기화 캡처라고 합니다. C++11 캡처 형식이 수행할 수 있는 거의 모든 작업과 그 이상을 수행할 수 있습니다. init capture로 표현할 수 없는 한 가지는 기본 캡처 모드이지만, 항목 31에서는 어쨌든 그런 방식을 피해야 한다고 설명합니다. (C++11 캡처로 처리되는 상황의 경우 init capture의 구문이 좀 더 많이 많으므로 C++11 캡처가 작업을 완료하는 경우 사용하는 것이 완벽합니다.)

초기화 캡처를 사용하면 다음을 지정할 수 있습니다.

1. 람다에서 생성된 클로저 클래스의 데이터 멤버 이름 및 2. 해당 데이터 멤버를 초기화 하는 표현식 .

다음은 init capture를 사용하여 std::unique_ptr를 클로저로 이동하는 방법입니다.

클래스 위젯 { 공개: // 유용한 유형

...

```
부울 isValidated() const; 부울
isProcessed() const; 부울
isArchived() const;
```

사적인:

...

};

```

자동 pw = std::make_unique<위젯>(); // 위젯 생성; 보다
                                                // std::make_unique에 대한 정
                                                // 보를 위한 // 항목 21

...
                                                // 구성 *pw

auto func = [pw = std::move(pw)] // 데이터 초기화 mbr { return pw->isValidated() // 클로
저 w/ && pw->isArchived(); }; // 표준::이동(pw)

```

강조 표시된 텍스트는 초기화 캡처를 구성합니다. "="의 왼쪽에는 지정하는 클로저 클래스의 데이터 멤버 이름이 있고 오른쪽에는 초기화 표현식이 있습니다. 흥미롭게도 "=" 왼쪽의 범위는 오른쪽의 범위와 다릅니다. 왼쪽의 범위는 클로저 클래스의 범위입니다. 오른쪽의 범위는 람다가 정의되는 위치와 동일합니다. 위의 예에서 "="의 왼쪽에 있는 이름 pw는 클로저 클래스의 데이터 멤버를 참조하는 반면 오른쪽에 있는 이름 pw는 람다 위에 선언된 개체, 즉 호출에 의해 초기화된 변수를 나타냅니다. std::make_unique로. 따라서 "pw = std::move(pw)"는 "클로저에 데이터 멤버 pw를 만들고 std::move를 로컬 변수 pw에 적용한 결과로 해당 데이터 멤버를 초기화합니다."를 의미합니다.

평소와 같이 람다 본문의 코드는 클로저 클래스의 범위에 있으므로 여기서 pw를 사용하면 클로저 클래스 데이터 멤버를 참조합니다.

이 예에서 "configure *pw" 주석은 위젯이 std::make_unique에 의해 생성된 후 해당 위젯에 대한 std::unique_ptr이 람다에 의해 캡처되기 전에 위젯이 어떤 식으로든 수정되었음을 나타냅니다. 그러한 구성이 필요하지 않은 경우, 즉 std::make_unique에 의해 생성된 위젯이 람다에 의해 캡처하기에 적합한 상태에 있는 경우 클로저 클래스의 데이터 멤버가 std에 의해 직접 초기화될 수 있기 때문에 로컬 변수 pw가 필요하지 않습니다. :make_unique:

```

auto func = [pw = std::make_unique<Widget>()] // 데이터 초기화 mbr { return pw-
                                                //isValidated() // 클로저 w/ && pw->isArchived(); // 표준::make_unique 호

```

이것은 C++11에서 표현식의 결과를 캡처하는 것이 불가능하기 때문에 C++14에서 "캡처"라는 개념이 C++11에서 상당히 일반화되었음을 분명히 해야 합니다. 결과적으로 초기화 캡처의 또 다른 이름은 일반화된 람다 캡처입니다.

그러나 사용하는 컴파일러 중 하나 이상이 C++14의 초기화 캡처를 지원하지 않는다면 어떻게 될까요? 이동 캡처를 지원하지 않는 언어로 어떻게 이동 캡처를 수행할 수 있습니까?

람다 식은 단순히 클래스가 생성되고 해당 유형의 객체가 생성되도록 하는 방법이라는 것을 기억하십시오. 손으로 할 수 없는 일을 람다로 할 수 있는 일은 없습니다. 예를 들어 방금 본 예제 C++14 코드는 다음과 같이 C++11로 작성할 수 있습니다.

```
클래스 IsValAndArch { 공개:
    사용 DataType =
        std::unique_ptr<Widget>; // "검증됨" 및 보관됨"

    명시적 IsValAndArch(DataType&& ptr) :
        pw(std::move(ptr)) // 항목 25 는 // std::move
                           사용을 설명합니다.

    부울 연산자() const { 반환 pw->isValidated() && pw->isArchived(); }

    사적인:
    데이터 유형 pw; };

```

자동 기능 = IsValAndArch(std::make_unique<위젯>());

그것은 람다를 작성하는 것보다 더 많은 작업이지만 데이터 멤버의 이동 초기화를 지원하는 C++11의 클래스를 원한다면 당신과 당신의 욕망 사이에 유일한 것은 약간의 시간이라는 사실은 변하지 않습니다. 당신의 키보드로.

람다를 고수하고 싶다면(그리고 편의상 그렇게 할 것입니다), 이동 캡처는 다음과 같이 C++11에서 에뮬레이트될 수 있습니다.

1. 캡처할 객체를 다음으로 생성된 함수 객체로 이동
std::bind 및
2. "캡처된" 객체에 대한 참조를 람다에 제공합니다.

std::bind에 익숙하다면 코드는 매우 간단합니다. std::bind에 익숙하지 않은 경우 코드에 익숙해지는 데 시간이 조금 걸리지만 문제가 될만한 가치가 있습니다.

로컬 std::vector를 만들고 적절한 값 집합을 넣은 다음 클로저로 이동한다고 가정합니다. C++14에서는 다음과 같이 쉽습니다.

```
std::vector<더블> 데이터; // 이동할 객체 // 클로저로

...
// 데이터 채우기

auto func = [data = std::move(data)] /* 데이터 사용 */; // C++14 초기화 캡처
```

이 코드의 핵심 부분을 강조했습니다. 이동하려는 객체 유형(`std::vector<double>`), 해당 객체의 이름(`data`), 초기화 캡처(`std::move(data)`). C++11에 해당하는 내용은 다음과 같습니다. 여기서 동일한 핵심 사항을 강조 표시했습니다.

```
std::vector<더블> 데이터; // 위와 같이

...
// 위와 같이

자동 기능 =
표준::바인드( // C++11 에뮬레이션 []
    (const std::vector<double>& data) // 초기화 캡처 { /* 데이터 사용 */ },
    std::move(data) );
```

람다 식과 마찬가지로 `std::bind`는 함수 개체를 생성합니다. `std::bind` 바인드 개체에서 반환된 함수 개체를 호출합니다. `std::bind`에 대한 첫 번째 인수는 호출 가능한 개체입니다. 후속 인수는 해당 개체에 전달할 값을 나타냅니다.

바인드 객체는 `std::bind`에 전달된 모든 인수의 복사본을 포함합니다. 각 lvalue 인수에 대해 `bind` 개체의 해당 개체는 복사 생성됩니다. 각 rvalue에 대해 이동이 구성됩니다. 이 예에서 두 번째 인수는 `rvalue(std::move의 결과 - 항목 23 참조)`이므로 데이터가 바인드 객체로 생성됩니다. 이 이동 구성은 이동 캡처 에뮬레이션의 핵심입니다. `rvalue`를 `bind` 개체로 이동하는 것이 `rvalue`를 C++ 11 클로저로 이동할 수 없는 문제를 해결하는 방법이기 때문입니다.

바인드 객체가 "호출"되면(즉, 함수 호출 연산자가 호출될 때) 이 객체가 저장하는 인수는 원래 `std::bind`에 전달된 호출 가능한 객체로 전달됩니다. 이 예에서 이는 `func(바인드 개체)`가 호출될 때 `func` 내부의 이동 생성된 데이터 복사본이 `std::bind`에 전달된 람다에 인수로 전달됨을 의미합니다.

이 람다는 의사 이동 캡처 개체에 해당하도록 매개 변수 `data`가 추가되었다는 점을 제외하고는 C++ 14에서 사용하는 람다와 동일합니다. 이 매개변수는 바인드 개체의 데이터 복사본에 대한 lvalue 참조입니다. (데이터 복사본("std::move(data)")을 초기화하는 데 사용되는 표현식이 `rvalue`이지만 데이터 복사본 자체가 `lvalue`이기 때문에 `rvalue` 참조가 아닙니다. 따라서 람다는 바인드 개체 내부의 이동 생성된 데이터 복사본에서 작동합니다.

기본적으로 람다에서 생성된 클로저 클래스 내부의 `operator()` 멤버 함수는 `const`입니다. 이는 클로저의 모든 데이터 멤버를 렌더링하는 효과가 있습니다.

람다의 본문 내에서 const. 바인드 객체 내부의 이동 생성 데이터 복사본은 const가 아니므로 해당 데이터 복사본이 람다 내부에서 수정되는 것을 방지하기 위해 람다의 매개 변수는 const에 대한 참조로 선언됩니다. 람다가 변경 가능하다고 선언되면 클로저 클래스의 operator()는 const로 선언되지 않으며 람다의 매개변수 선언에서 const를 생략하는 것이 적절할 것입니다.

자동 기능 =

```
표준::바인드( // C++11 에뮬레이션 []
    (std::vector<double>& data) mutable // of init capture { /* 데이터 사용 */ },
    std::move(data) ); // 가변 람다의 경우
```

bind 객체는 std::bind에 전달된 모든 인수의 복사본을 저장하기 때문에 이 예제의 bind 객체에는 첫 번째 인수인 람다가 생성한 클로저 복사본이 포함되어 있습니다. 그러므로 클로저의 수명은 바인드 객체의 수명과 같습니다. 클로저가 존재하는 한 의사 이동 캡처 객체를 포함하는 바인드 객체도 존재한다는 것을 의미하기 때문에 중요합니다.

이것이 std::bind에 대한 첫 번째 노출인 경우 앞의 논의에 대한 모든 세부 사항이 자리를 잡기 전에 좋아하는 C++11 참조를 참조해야 할 수도 있습니다. 이 경우에도 다음과 같은 기본 사항이 명확해야 합니다.

- 객체를 C++11 클로저로 이동 구성하는 것은 불가능하지만 가능합니다.
객체를 C++11 바인드 객체로 이동 구성할 수 있습니다.
- C++11에서 이동 캡처를 에뮬레이트하는 것은 객체를 바인드 객체로 이동 구성한 다음 참조를 통해 이동 구성 객체를 람다에 전달하는 것으로 구성됩니다.
엔스.
- bind 객체의 수명은 클로저의 수명과 같기 때문에 가능하다.
바인드 객체의 객체를 마치 클로저에 있는 것처럼 취급할 수 있습니다.

이동 캡처를 에뮬레이트하기 위해 std::bind를 사용하는 두 번째 예로서 클로저에 std::unique_ptr을 생성하기 위해 앞에서 본 C++14 코드가 있습니다.

```
자동 기능 = [pw = std::make_unique<위젯>()] { 반환 pw->isValidated() && pw->isArchived(); }; // 이전과 같이 // pw 를 // 생성합니다.
```

다음은 C++11 에뮬레이션입니다.

```
자동 기능 = std::bind(
    [](const std::unique_ptr<위젯>& pw) { return pw->isValidated() && pw->isArchived(); },
```

```
std::make_unique<위젯>() );
```

항목 34에서 std::bind보다 람다 사용을 옹호하기 때문에 std::bind를 사용하여 C++11 람다의 제한 사항을 해결하는 방법을 보여주고 있다는 것은 아닙니다.

그러나 해당 항목은 C++11에서 std::bind가 유용할 수 있는 몇 가지 경우가 있으며 이것이 그 중 하나라고 설명합니다. (C++14에서는 초기화 캡처 및 자동 매개변수와 같은 기능이 이러한 경우를 제거합니다.)

기억할 사항 • C++14의

초기화 캡처를 사용하여 개체를 클로저로 이동합니다. • C++11에서는

손으로 작성한 클래스 또는 std::bind를 통해 초기화 캡처를 에뮬레이트합니다.

항목 33: auto&& 매개변수에 decltype을 사용하여 std::forward를 전달 하십시오.

C++14의 가장 흥미로운 기능 중 하나는 매개변수 사양에서 auto를 사용하는 람다인 일반 람다입니다. 이 기능의 구현은 간단합니다. 람다의 클로저 클래스에 있는 operator()는 템플릿입니다. 예를 들어, 이 람다가 주어졌을 때,

```
자동 f = [](자동 x){ return func(normalize(x)); };
```

클로저 클래스의 함수 호출 연산자는 다음과 같습니다.

```
클래스 SomeCompilerGeneratedClassName { 공개:
    템플릿<유형 이름 T> 자동 연산자()(T x) const { return
        func(normalize(x)); } // 자동 반환 유형은 // 항
    목 3 참조
}
```

```
... // 다른 클로저 클래스 // 가능
};
```

이 예에서 람다가 매개변수 x로 수행하는 유일한 작업은 정규화를 위해 전달하는 것입니다. normalize가 lvalue를 rvalue와 다르게 취급하면 람다에 전달된 인수가 rvalue인 경우에도 정규화를 위해 항상 lvalue(매개변수 x)를 전달하기 때문에 이 람다가 제대로 작성되지 않았습니다.

람다를 작성하는 올바른 방법은 정규화하기 위해 완전 순방향 x를 사용하는 것입니다.

그렇게 하려면 코드를 두 번 변경해야 합니다. 첫째, x는 보편적 참조가 되어야 합니다.

(**항목 24 참조**) 두 번째로 std::forward를 통해 정규화하기 위해 전달해야 합니다(**항목 25 참조**). 개념적으로는 다음과 같은 사소한 수정 사항입니다.

```
자동 f = [](자동&& x) { return
    func(normalize(std::forward<???(x)); };
```

그러나 개념과 실현 사이에는 std::forward에 전달할 유형의 문제가 있습니다. 위에.

일반적으로 완벽한 전달을 사용하면 형식 매개변수 T를 사용하는 템플릿 함수에 있으므로 std::forward<T>를 작성하면 됩니다. 그러나 일반 람다에서는 사용할 수 있는 형식 매개 변수 T가 없습니다. 람다에 의해 생성된 클로저 클래스 내부의 템플릿화된 operator()에 T가 있지만 람다에서 이를 참조할 수 없으므로 아무 소용이 없습니다.

항목 28은 lvalue 인수가 범용 참조 매개변수에 전달되면 해당 매개변수의 유형이 lvalue 참조가 된다고 설명합니다. rvalue가 전달되면 매개변수는 rvalue 참조가 됩니다. 이는 람다에서 매개변수 x의 유형을 검사하여 전달된 인수가 lvalue인지 rvalue인지 결정할 수 있음을 의미합니다. decltype은 그렇게 하는 방법을 제공합니다(**항목 3 참조**). lvalue가 전달된 경우 decltype(x)는 lvalue 참조인 유형을 생성합니다. rvalue가 전달되면 decltype(x)는 rvalue 참조 유형을 생성합니다.

항목 28은 또한 std::forward를 호출할 때 형식 인수가 lvalue를 나타내는 lvalue 참조이고 rvalue를 나타내는 비참조가 되어야 한다고 규정하고 있다고 설명합니다. 람다에서 x가 lvalue에 바인딩되면 decltype(x)는 lvalue 참조를 생성합니다. 그것은 관례에 따릅니다. 그러나 x가 rvalue에 바인딩된 경우 decltype(x)는 관례적인 비 참조 대신 rvalue 참조를 생성합니다.

그러나 **항목 28**의 std::forward에 대한 샘플 C++14 구현을 보십시오 .

```
템플릿<유형이름 T>                                // 네임스페이스에서 //
T&& 앞으로(remove_reference_t<T>& param) {           표준
    반환 static_cast<T&&>(param);
```

클라이언트 코드가 Widget 유형의 rvalue를 완벽하게 전달하려는 경우 일반적으로 Widget 유형(즉, 비참조 유형)으로 std::forward를 인스턴스화하고 std::forward 템플릿은 다음 기능을 생성합니다.

```
위젯&& 앞으로(위젯& 매개변수) {                   // 인스턴스화 // std::forward
    때
    return static_cast<Widget&&>(param);            // T는 위젯
}
```

그러나 클라이언트 코드가 Widget 유형의 동일한 rvalue를 완벽하게 전달하기를 원했지만 T를 비참조 유형으로 지정하는 규칙을 따르는 대신 rvalue 참조로 지정했다면 어떤 일이 일어날지 생각해 보십시오. 즉, T가 Widget&&로 지정되면 어떻게 되는지 생각해 보십시오. std::forward의 초기 인스턴스화 및 std::remove_reference_t의 적용 후, 그러나 참조가 축소되기 전(다시 한 번 항목 28 참조), std::forward는 다음과 같이 보일 것입니다.

```
위젯&& && 앞으로(위젯& 매개변수) {
    // 인스턴스화 // std::forward
    when return static_cast<Widget&&
        &&>(param); // T는 Widget&&입니다 . // (참조 전- // 축소)
    }
}
```

rvalue 참조에 대한 rvalue 참조가 단일 rvalue 참조가 된다는 참조 축소 규칙을 적용하면 이 인스턴스화가 나타납니다.

```
위젯&& 앞으로(위젯& 매개변수) {
    // 인스턴스화 // std::forward 때
    return static_cast<Widget&&>(param);
}
// T는 Widget&&입니다 . //
// (참조 후- // 축소)
```

이 인스턴스화를 T가 Widget으로 설정하고 std::forward가 호출될 때 생성된 인스턴스화와 비교하면 동일하다는 것을 알 수 있습니다. 즉, rvalue 참조 유형으로 std::forward를 인스턴스화하면 비참조 유형으로 인스턴스화하는 것과 동일한 결과가 생성됩니다.

rvalue가 람다의 매개변수 x에 대한 인수로 전달될 때 decltype(x)이 rvalue 참조 유형을 생성하기 때문에 이는 놀라운 소식입니다. 위에서 우리는 lvalue가 람다에 전달될 때 decltype(x)이 std::forward에 전달할 관습적 유형을 산출한다는 것을 설정했고 이제 rvalue에 대해 decltype(x) 가 std에 전달할 유형을 산출한다는 것을 깨달았습니다. ::forward 그것은 관습적이지 않지만 그럼에도 불구하고 전통적인 유형과 동일한 결과를 산출합니다. 따라서 lvalue와 rvalue 모두에 대해 decltype(x)를 std::forward에 전달하면 원하는 결과를 얻을 수 있습니다. 따라서 완벽한 전달 람다는 다음과 같이 작성할 수 있습니다.

```
자동 f = []
(자동&& 매개변수) {
    반품
    func(normalize(std::forward<decltype(param)>(param)));
}
```

거기에서 C++14 람다도 가변적일 수 있기 때문에 단일 매개변수뿐만 아니라 여러 매개변수를 허용하는 완전 전달 람다에 대한 흡, 건너뛰기 및 6개의 점입니다.

```

자동 f = [](자
동&&... 매개변수) {
    반품
    func(normalize(std::forward<decltype(params)>(params...)));
};

```

기억해야 할 사항

- auto&& 매개변수에 decltype을 사용하여 std::forward를 전달합니다.

항목 34: std::bind보다 람다를 선호하십시오.

std::bind는 C++98의 std::bind1st 및 std::bind2nd의 C++11 후속 제품이지만 비공식적으로는 2005년부터 표준 라이브러리의 일부였습니다. 이때 표준화 위원회에서 다음으로 알려진 문서를 채택했습니다. bind의 사양이 포함된 TR1. (TR1에서는 바인드가 다른 네임스페이스에 있었기 때문에 std::bind가 아니라 std::tr1::bind였고, 인터페이스 세부사항이 약간 달랐습니다.) 이 역사는 일부 프로그래머가 10년 이상의 경험을 가지고 있음을 의미합니다. std::bind를 사용합니다. 당신이 그들 중 하나라면 당신에게 잘 맞는 도구를 포기하기를 꺼릴 수 있습니다. 이해할 수 있지만 이 경우 변경이 좋습니다. C++11에서 람다가 거의 항상 std::bind보다 더 나은 선택이기 때문입니다. C++14에서 람다의 경우는 더 강력할 뿐만 아니라 완전히 철통처럼 되어 있습니다.

이 항목은 std::bind에 익숙하다고 가정합니다. 그렇지 않은 경우 계속하기 전에 기본적인 이해를 얻고 싶을 것입니다. 이러한 이해는 어떤 경우에도 가치가 있습니다. 읽거나 유지해야 하는 코드 기반에서 언제 std::bind를 사용하게 될지 알 수 없기 때문입니다.

항목 32에서와 같이 std::bind에서 반환된 함수 개체를 바인드 개체로 참조합니다.

std::bind보다 람다를 선호하는 가장 중요한 이유는 람다가 더 읽기 쉽다는 것입니다. 예를 들어 가정 경보를 설정하는 기능이 있다고 가정합니다.

```

// Time = std::chrono::steady_clock::time_point를 사용하여 특정 시점에 대한 typedef( 구
문 은 항목 9 참조);
// "enum class" 항목 10 참조 enum class Sound
{ Beep, Siren, Whistle };

// 일정 시간 동안 typedef

```

Duration = std::chrono::steady_clock::duration 사용

// 시간 t에서 지속 시간 d 동안 소리 s를 만듭니다. void setAlarm(시간 t, 소리 s, 지속 시간 d);

또한 프로그램의 어느 시점에서 설정한 후 1시간 후에 울리고 30초 동안 계속 켜져 있는 알람을 원한다고 결정했다고 가정합니다. 다만 경보음은 미정이다. 소리만 지정하면 되도록 setAlarm의 인터페이스를 수정하는 람다를 작성할 수 있습니다.

```
// setSoundL("lambda"의 경우 "L")은 // 30초 알람이 울리도록 지정되는 소리를 허용하는 함수 객체입니다. // 설정된 후 // 자동 setSoundL =
```

```
[](소리) {
    // 네임스페이스를 사용하여 자격 없이 std::chrono 구성 요소를 사용할 수 있도록 합니다.
    std::chrono;
    setAlarm(steady_clock::now() + hours(1), // 알람 해제
        // 1시간 동안 // 30초 동안
        초, 초(30));
};
```

람다 내에서 setAlarm 호출을 강조 표시했습니다. 이것은 정상적인 함수 호출이며, 람다 경험이 거의 없는 독자도 람다에 전달된 매개변수가 setAlarm에 대한 인수로 전달된다는 것을 알 수 있습니다.

사용자 정의 리터럴에 대한 C++11의 지원을 기반으로 하는 초(s), 밀리초(ms), 시간(h) 등에 대한 표준 접미사를 활용하여 C++14에서 이 코드를 간소화할 수 있습니다. 이러한 접미사는 std::literals 네임스페이스에서 구현되므로 위의 코드는 다음과 같이 다시 작성할 수 있습니다.

```
자동 setSoundL =
[](Sound s) { 네
    임스페이스
        std::chrono 사용; 네임스페이스 std::literals
        사용
        // C++14 접미사의 경우
        setAlarm(steady_clock::now() + 1h,
            // C++14이지만 // 위
            // 같은 의미입니다.
            예스,
            30초);
};
```

해당 std::bind 호출을 작성하려는 첫 번째 시도는 아래와 같습니다. 잠시 후에 수정할 오류가 있지 만 올바른 코드는 더 복잡하고 이 단순화된 버전에서도 몇 가지 중요한 문제가 발생합니다.

```
네임스페이스 사용 std::chrono; 네임스페이스 std::literals 사용           // 위와 같이
네임스페이스 std::placeholders 사용                                         // "_1" 사용에 필요
auto setSoundB =                                                       // "바인드"의 경우 "B"
    std::bind(setAlarm, steady_clock::now() + 1h, // 틀립니다! 아래 참조 _1, 30s);
```

람다에서 했던 것처럼 여기에서 setAlarm 호출을 강조 표시하고 싶지만 강조 표시할 호출이 없습니다. 이 코드를 읽는 사람은 setSoundB를 호출하면 std::bind에 대한 호출에 지정된 시간과 지속 시간으로 setAlarm이 호출된다는 것을 알아야 합니다. 초심자에게 자리 표시자 "_1"은 본질적으로 마법이지만, 알고 있는 독자도 해당 자리 표시자의 숫자에서 std::bind 매개변수 목록의 위치로 정신적으로 매팅해야 setSoundB에 대한 호출은 setAlarm에 대한 두 번째 인수로 전달됩니다. 이 인수의 유형은 std::bind에 대한 호출에서 식별되지 않으므로 독자는 setAlarm 선언을 참조하여 setSoundB에 전달할 인수의 종류를 결정해야 합니다.

그러나 내가 말했듯이 코드는 정확하지 않습니다. 람다에서 "steady_clock::now() + 1h"라는 표현이 setAlarm에 대한 인수임이 분명합니다. setAlarm이 호출될 때 평가됩니다. 그것은 의미가 있습니다. 우리는 setAlarm을 호출한 후 한 시간 후에 알람이 울리기를 원합니다. 그러나 std::bind 호출에서 "steady_clock::now() + 1h"는 setAlarm이 아닌 std::bind에 인수로 전달됩니다. 즉, std::bind가 호출될 때 표현식이 평가되고 해당 표현식으로 인한 시간이 결과 바인드 객체 내부에 저장됩니다. 결과적으로 알람은 setAlarm! 호출 후 1시간이 아니라 std::bind 호출 후 1시간 후에 울리도록 설정됩니다.

문제를 해결하려면 setAlarm이 호출될 때까지 표현식 평가를 연기하도록 std::bind에 지시해야 합니다. 그렇게 하는 방법은 첫 번째 호출 안에 std::bind에 대한 두 번째 호출을 중첩하는 것입니다.

```
자동 setSoundB =
    std::bind(setAlarm,
        std::bind(std::plus<>(), stable_clock::now(), _1, 30s);
```

C++98의 std::plus 템플릿에 익숙하다면 이 코드에서 꺼쇠 괄호 사이에 유형이 지정되지 않은 것을 보고 놀랄 수 있습니다. 즉, 코드에 "std::plus<>"가 포함되어 있습니다. ", "std::plus<type>"이 아닙니다. C++14에서는 일반적으로 표준 연산자 템플릿에 대한 템플릿 유형 인수를 생략할 수 있으므로 여기에 제공할 필요가 없습니다. C++11은 이러한 기능을 제공하지 않으므로 람다에 해당하는 C++11 std::bind는 다음과 같습니다.

네임스페이스 사용 std::chrono; 네임스페이스 std::placeholders 사용 // 위와 같이

```
자동 setSoundB =
    std::bind(setAlarm,
        std::bind(std::plus<steady_clock::time_point>(), stable_clock::now(),
            시간(1)),
        _1,
        초(30));
```

이 시점에서 람다가 훨씬 더 매력적으로 보이지 않는다면 아마도 시력 검사를 받아야 합니다.

setAlarm이 오버로드되면 새로운 문제가 발생합니다. 알람 볼륨을 지정하는 네 번째 매개변수를 사용하는 과부하가 있다고 가정합니다.

열거형 클래스 볼륨 { 보통, 크게, LoudPlusPlus };

void setAlarm(시간 t, 소리 s, 지속 시간 d, 볼륨 v);

과부하 해결은 setAlarm의 3개 인수 버전을 선택하기 때문에 람다는 계속해서 이전과 같이 작동합니다.

```
자동 setSoundL =
    [](Sound s) { 네임스페이스 std::chrono 사용;
        setAlarm(steady_clock::now() + 1h,
            예스,
            30대);
    };
// 이전과 동일
```

// 좋아요, // setAlarm
의 // 3-arg 버전을 호출합니다.

반면에 std::bind 호출은 이제 컴파일에 실패합니다.

```
자동 setSoundB =
    std::bind(setAlarm,
        std::bind(std::plus<>(),
            꾸준한 시계::지금()),
```

// 오류! 어느 // setAlarm?

```
1시간),
_1,
30초);
```

문제는 컴파일러가 두 개의 setAlarm 함수 중 std::bind에 전달해야 하는 함수를 결정할 방법이 없다는 것입니다. 함수 이름만 있고 이름만으로도 모호합니다.

컴파일을 위해 std::bind 호출을 가져오려면 setAlarm을 적절한 함수 포인터 유형으로 캐스팅해야 합니다.

```
SetAlarm3ParamType = void(*)(시간 t, 소리 s, 지속 시간 d) 사용;
```

```
자동 setSoundB =
std::bind(static_cast<SetAlarm3ParamType>(setAlarm), // 좋습니다
          std::bind(std::plus<>(),
                    꾸준한 시계::지금(), 1h),
_1,
30초);
```

그러나 이것은 람다와 std::bind 사이에 또 다른 차이점을 나타냅니다. setSoundL의 함수 호출 연산자(즉, 람다의 클로저 클래스의 함수 호출 연산자) 내에서 setAlarm에 대한 호출은 일반적인 방식으로 컴파일러에 의해 인라인될 수 있는 일반 함수 호출입니다.

```
setSoundL(사운드::사이렌); // setAlarm의 본문은 // 여기에 잘 인
                             라인될 수 있습니다.
```

그러나 std::bind에 대한 호출은 setAlarm에 대한 함수 포인터를 전달하며, 이는 setSoundB에 대한 함수 호출 연산자(즉, 바인드 개체에 대한 함수 호출 연산자) 내부에서 setAlarm에 대한 호출이 함수를 통해 발생함을 의미합니다. 바늘. 컴파일러는 함수 포인터를 통해 함수 호출을 인라인할 가능성이 적습니다. 즉, setSoundB를 통한 setAlarm 호출은 setSoundL을 통한 호출보다 완전히 인라인될 가능성이 적습니다.

```
setSoundB(사운드::사이렌); // setAlarm의 본문은 // 여기에 인라인될 가
                             능성이 적습니다.
```

따라서 람다를 사용하면 std::bind를 사용하는 것보다 더 빠른 코드가 생성될 수 있습니다.

setAlarm 예제에는 간단한 함수 호출만 포함됩니다. 더 복잡한 작업을 수행하려면 저율이 람다 쪽으로 훨씬 더 기울어집니다. 예를 들어, 인수가 최소값(lowVal)과 최대값(highVal) 사이에 있는지 여부를 반환하는 C++14 람다를 생각해 보십시오. 여기서 lowVal과 highVal은 지역 변수입니다.

```
auto betweenL =
    [lowVal, highVal] (const
        auto& val) // C++14
    { return lowVal <= val && val <= highVal; };
```

`std::bind`는 같은 것을 표현할 수 있지만 구조는 코드 모호성을 통한 직업 보안의 예입니다.

네임스페이스 `std::placeholders` 사용 // 위와 같이

```
자동 사이B =
    std::bind(std::logical_and<>(),
              std::bind(std::less_equal<>(), lowVal, _1),
              std::bind(std::less_equal<>(), _1, highVal)); // C++14
```

C++11에서는 비교하려는 유형을 지정해야 하며 `std::bind` 호출은 다음과 같습니다.

```
자동 사이B =
    std::bind(std::logical_and<bool>(),
              std::bind(std::less_equal<int>(), lowVal, _1),
              std::bind(std::less_equal<int> (), _1, highVal)); // C++11 버전
```

물론 C++11에서 람다는 `auto` 매개변수를 사용할 수 없으므로 유형에도 커밋해야 합니다.

```
auto betweenL = // C++11 버전
    [lowVal, highVal] (int
        val) { return lowVal <=
            val && val <= highVal; };
```

어느 쪽이든 람다 버전이 더 짧을 뿐만 아니라 더 이해하기 쉽고 유지 관리하기 쉽다는 데 동의할 수 있기를 바랍니다.

앞서 나는 `std::bind` 경험이 거의 없는 사람들에게 그 자리 표시자(예: `_1`, `_2` 등)가 본질적으로 마법이라고 언급했습니다. 그러나 불투명한 것은 자리 표시자의 동작만이 아닙니다. 위젯의 압축된 복사본을 만드는 기능이 있다고 가정합니다.

열거형 클래스 `CompLevel` { 낮음, 보통, 높음 }; // 압축 // 레벨

```
위젯 압축(const Widget& w,
            CompLevel lev); // 압축된 // w의 복사본 만들
                           기
```

그리고 우리는 특정 위젯 `w`를 얼마나 압축해야 하는지 지정할 수 있는 함수 객체를 만들고 싶습니다. `std::bind`를 사용하면 다음과 같은 객체가 생성됩니다.

위젯 w;

네임스페이스 std::placeholders 사용

자동 compressRateB = std::bind(압축, w, _1);

이제 w를 std::bind에 전달할 때 압축을 위한 나중 호출을 위해 저장해야 합니다.

객체 compressRateB 내부에 저장되지만 값으로 또는 참조로 어떻게 저장됩니까? std::bind에 대한 호출과 compressRateB에 대한 호출 사이에 w가 수정된 경우 참조로 w를 저장하면 변경 사항이 반영되지만 값으로 저장하면 변경 사항이 반영되지 않기 때문에 차이가 있습니다.

대답은 값에 의해 저장된다는 것입니다. 그러나 이를 알 수 있는 유일한 방법은 std::bind가 어떻게 작동하는지 기억하는 것입니다. std::bind에 대한 호출에는 표시가 없습니다. w가 값으로 캡처되는지 아니면 참조로 캡처되는지가 명시적인 람다 접근 방식과 대조됩니다.

자동 compressRateL = [w]
 (CompLevel lev) { 반환 압
 축(w, lev); };

// w는 // 값으로 캡처됩니다.
 lev는 // 값으로 전달됩니다.

매개변수가 람다에 전달되는 방식도 마찬가지로 명시적입니다. 여기에서 매개변수 lev가 값으로 전달된다는 것이 분명합니다. 따라서:

compressRateL(CompLevel::High);

// 인수는 // 값으로 전달됩니다.

그러나 std::bind로 인한 객체 호출에서 인수는 어떻게 전달됩니까?

compressRateB(CompLevel::High);

// 인수는 // 어떻게 전달됩니까?

다시 말하지만, 알 수 있는 유일한 방법은 std::bind가 작동하는 방식을 암기하는 것입니다. (대답은 바인드 객체에 전달된 모든 인수가 참조로 전달된다는 것입니다. 그러한 객체에 대한 함수 호출 연산자는 완전 전달을 사용하기 때문입니다.)

람다와 비교할 때 std::bind를 사용하는 코드는 가독성이 떨어지고 표현력이 떨어지며 효율성이 떨어질 수 있습니다. C++14에는 std::bind에 대한 합리적인 사용 사례가 없습니다. 그러나 C++11에서 std::bind는 두 가지 제한된 상황에서 정당화될 수 있습니다.

사용:

1 std::bind는 항상 인수를 복사하지만 호출자는 std::ref를 적용하여 참조로 인수를 저장하는 효과를 얻을 수 있습니다. 결과

자동 compressRateB = std::bind(압축, std::ref(w), _1);

compressRateB는 복사본이 아닌 w에 대한 참조를 보유하는 것처럼 작동합니다.

- 이동 캡처. C++11 람다는 이동 캡처를 제공하지 않지만 람다와 std::bind의 조합을 통해 에뮬레이트할 수 있습니다. 자세한 내용은 C++14에서 람다의 초기화 캡처 지원으로 에뮬레이션이 필요하지 않다고 설명하는 [항목 32](#)를 참조하세요.
- 다형성 함수 개체. 바인드 객체의 함수 호출 연산자는 완전 전달을 사용하기 때문에 모든 유형의 인수를 허용할 수 있습니다([항목 30](#)에 설명된 완전 전달에 대한 모듈로 제한 사항). 이것은 템플릿화된 함수 호출 연산자를 사용하여 개체를 바인딩하려는 경우에 유용할 수 있습니다. 예를 들어, 이 클래스가 주어졌을 때,

```
클래스 PolyWidget { 공개:
    템플릿<유형 이름 T> 무효 연
        산자()(const T& 매개변수);

    ...
};
```

std::bind는 다음과 같이 PolyWidget을 바인딩할 수 있습니다.

```
폴리위젯 pw;
자동 boundPW = std::bind(pw, _1);
```

그런 다음 여러 유형의 인수를 사용하여 boundPW를 호출할 수 있습니다.

```
바운드PW(1930);           // int를 전달
                           // 폴리위젯::연산자()

boundPW(nullptr);          // nullptr을 전달
                           // 폴리위젯::연산자()

boundPW("로즈버드");      // 문자열 리터럴 전달
                           // 폴리위젯::연산자()
```

C++11 람다에서는 이를 수행할 방법이 없습니다. 그러나 C++14에서는 auto 매개변수가 있는 람다를 통해 쉽게 달성을 할 수 있습니다.

```
자동 boundPW = [pw](const auto& param)           // C++14
{ pw(param); };
```

물론 이것은 극단적인 경우이며 C++14 람다를 지원하는 컴파일러가 점점 더 일반적이기 때문에 일시적인 경우입니다.

bind가 2005년에 C++에 비공식적으로 추가되었을 때 1998년 이전에 비해 크게 개선되었습니다. 그러나 C++11에 람다 지원을 추가하면 std::bind가 더 이상 사용되지 않고 렌더링되고 C++14에서는 이에 대한 좋은 사용 사례가 없습니다.

기억해야 할 사항 •

Lambda는 std::bind를 사용하는 것보다 읽기 쉽고 표현력이 뛰어나며 더 효율적일 수 있습니다.

- C++11에서만 std::bind는 이동 캡처를 구현하거나 템플릿화된 함수 호출 연산자로 객체를 바인딩하는 데 유용할 수 있습니다.

동시성 API

C++11의 위대한 승리 중 하나는 언어와 라이브러리에 동시성을 통합한 것입니다. 다른 스레딩 API(예: pthreads 또는 Windows 스레드)에 익숙한 프로그래머는 때때로 C++가 제공하는 비교적 스파르타적인 기능 세트에 놀라기도 합니다. 그 결과 언어 보증은 C++ 역사상 처음으로 프로그래머가 모든 플랫폼에서 표준 동작으로 다중 스레드 프로그램을 작성할 수 있음을 의미합니다. 이것은 표현형 라이브러리를 구축할 수 있는 견고한 기반을 설정하고 표준 라이브러리의 동시성 요소(작업, 퓨처, 스레드, 뮤텍스, 조건 변수, 원자 객체 등)는 동시 C++ 소프트웨어 개발을 위한 점점 더 풍부한 도구 세트가 됩니다.

다음 항목에서 표준 라이브러리에는 `std::future` 및 `std::shared_future`의 두 가지 `future` 템플릿이 있습니다. 많은 경우 구별이 중요하지 않기 때문에 나는 종종 단순히 미래에 대해 이야기하는데, 두 가지를 모두 의미합니다.

항목 35: 스레드 기반보다 작업 기반 프로그래밍을 선호하십시오.

`doAsyncWork` 함수를 비동기적으로 실행하려는 경우 두 가지 기본 선택 사항이 있습니다. `std::thread`를 만들고 `doAsyncWork`를 실행하여 스레드 기반 접근 방식을 사용할 수 있습니다.

```
int doAsyncWork();
```

`표준:: 스레드 t(doAsyncWork);`

또는 작업 기반으로 알려진 전략인 `doAsyncWork`를 `std::async`에 전달할 수 있습니다.

```
자동 fut = std::async(doAsyncWork); // "미래"에 대한 "fut"
```

이러한 호출에서 std::async(예: doAsyncWork)에 전달된 함수 자체는 작업으로 간주됩니다.

작업 기반 접근 방식은 일반적으로 스레드 기반 접근 방식보다 우수하며 우리가 본 적은 양의 코드가 이미 몇 가지 이유를 보여줍니다. 여기에서 doAsyncWork는 반환 값을 생성합니다. 이 값은 doAsyncWork를 호출하는 코드가 관심을 갖고 있다고 합리적으로 가정할 수 있습니다. 스레드 기반 호출을 사용하면 이에 액세스할 수 있는 직접적인 방법이 없습니다. 작업 기반 접근 방식을 사용하면 std::async에서 반환된 future가 get 기능을 제공하기 때문에 쉽습니다. get 함수는 doAsyncWork가 예외를 내보낸다면 훨씬 더 중요합니다. get도 이에 대한 액세스를 제공하기 때문입니다. 스레드 기반 접근 방식을 사용하면 doAsyncWork가 throw되면 프로그램이 종료됩니다(std::terminate 호출을 통해).

스레드 기반 프로그래밍과 작업 기반 프로그래밍의 보다 근본적인 차이점은 작업 기반이 구현하는 더 높은 수준의 추상화입니다. 이것은 스레드 관리의 세부 사항에서 당신을 해방시킵니다. 동시성 C++ 소프트웨어에서 "스레드"의 세 가지 의미를 요약할 필요가 있음을 상기시키는 관찰입니다.

- 하드웨어 스레드는 실제로 계산을 수행하는 스레드입니다. 현대 기계 아키텍처는 CPU 코어당 하나 이상의 하드웨어 스레드를 제공합니다. • 소프트웨어 스레드(OS 스레드 또는 시스템 스레드라고도 함)는 운영 체제¹가 하드웨어 스레드에서 실행하기 위해 모든 프로세스와 일정에서 관리하는 스레드입니다. 일반적으로 하드웨어 스레드보다 더 많은 소프트웨어 스레드를 생성할 수 있습니다. 소프트웨어 스레드가 차단되면(예: I/O에서 또는 뮤텍스 또는 조건 변수를 기다리는 경우) 차단되지 않은 다른 스레드를 실행하여 처리량을 향상시킬 수 있기 때문입니다.
- std::threads 는 기본 소프트웨어 스레드에 대한 핸들 역할을 하는 C++ 프로세스의 자체입니다. 일부 std::thread 객체는 "null" 핸들을 나타냅니다. 즉, 기본 구성 상태(따라서 실행할 기능이 없음)에 있기 때문에 소프트웨어 스레드에 해당하지 않습니다. 그런 다음 std::thread는 기본 소프트웨어 스레드에 대한 핸들 역할을 하고, 결합되었거나(실행할 기능이 완료됨) 분리되었습니다 (그들과 기본 소프트웨어 스레드 사이의 연결이 끊어짐).

소프트웨어 스레드는 제한된 리소스입니다. 시스템이 제공할 수 있는 것보다 더 많이 생성하려고 하면 std::system_error 예외가 발생합니다. 실행하려는 함수가 throw할 수 없는 경우에도 마찬가지입니다. 예를 들어 doAsyncWork가 noexcept인 경우에도

¹ 가지고 있다고 가정합니다. 일부 임베디드 시스템은 그렇지 않습니다.

```
int doAsyncWork() noexcept;
```

// noexcept에 대한 항목 14 참조

이 문은 예외를 초래할 수 있습니다.

표준:: 스레드 t(doAsyncWork);

// 더 이상 사용 가능한 스레드
가 없으면 던집니다.

잘 작성된 소프트웨어는 이러한 가능성을 어떻게든 처리해야 합니다. 하지만 어떻게 해야 합니까? 한 가지 접근 방식은 현재 스레드에서 doAsyncWork를 실행하는 것이지만 이는 불균형 로드로 이어질 수 있고 현재 스레드가 GUI 스레드인 경우 응답 문제를 일으킬 수 있습니다. 또 다른 옵션은 일부 기존 소프트웨어 스레드가 완료될 때까지 기다렸다가 새 std::thread를 다시 생성하려고 시도하는 것(이지만 기존 스레드가 doAsyncWork가 수행해야 하는 작업(예: 결과 생성 또는 조건 변수를 알립니다)).

스레드가 부족하지 않더라도 초과 구독에 문제가 발생할 수 있습니다.

하드웨어 스레드보다 실행 준비가 된(즉, 차단되지 않은) 소프트웨어 스레드가 더 많을 때입니다. 이 경우 스레드 스케줄러(일반적으로 OS의 일부)는 하드웨어의 소프트웨어 스레드를 시간 분할합니다. 한 스레드의 타임 슬라이스가 끝나고 다른 스레드의 타임 슬라이스가 시작되면 컨텍스트 전환이 수행됩니다. 이러한 컨텍스트 스위치는 시스템의 전체 스레드 관리 오버헤드를 증가시키며, 소프트웨어 스레드가 예약된 하드웨어 스레드가 마지막 시간 동안 소프트웨어 스레드의 경우와 다른 코어에 있을 때 특히 비용이 많이 들 수 있습니다. 일부분.

이 경우, (1) CPU 캐시는 일반적으로 해당 소프트웨어 스레드에 대해 차갑고(즉, 데이터와 유용한 명령이 거의 포함되지 않음) (2) 해당 코어에서 "새" 소프트웨어 스레드를 실행하면 "오염"됩니다. 해당 코어에서 실행 중이었고 다시 실행되도록 예약된 "이전" 스레드에 대한 CPU 캐시입니다.

소프트웨어 스레드와 하드웨어 스레드의 최적 비율은 소프트웨어 스레드가 실행 가능한 빈도에 따라 달라지며, 이는 예를 들어 프로그램이 I/O가 많은 영역에서 연산 영역으로 이동할 때 동적으로 변경될 수 있기 때문에 초과 구독을 피하는 것은 어렵습니다. 무거운 지역. 하드웨어 스레드에 대한 소프트웨어의 최상의 비율은 컨텍스트 전환 비용과 소프트웨어 스레드가 CPU 캐시를 얼마나 효과적으로 사용하는지에 따라 달라집니다. 또한 하드웨어 스레드의 수와 CPU 캐시의 세부 정보(예: 스레드의 크기 및 상대 속도)는 시스템 아키텍처에 따라 다르므로 초과 구독을 방지하기 위해 애플리케이션을 조정하더라도(하드웨어를 계속 사용 중인 상태로 유지하면서) 한 플랫폼에서는 솔루션이 다른 종류의 시스템에서 잘 작동한다는 보장이 없습니다.

이러한 문제를 다른 사람에게 맡기고 std::async를 사용하면 정확히 다음과 같은 작업을 수행할 수 있습니다.

자동 fut = std::async(doAsyncWork); // 스레드 mgmt의 책임은 // 표준 라이브러리의 구현자에 있습니다.

이 호출은 스레드 관리 책임을 C++ 표준 라이브러리의 구현자에게 이전합니다. 예를 들어, `out-of-threads` 예외를 수신할 가능성은 상당히 감소합니다. 왜냐하면 이 호출은 아마도 절대 예외를 생성하지 않을 것이기 때문입니다. "어떻게 그렇게 될수 있니?" 당신은 궁금해 할 수 있습니다. "시스템이 제 공할 수 있는 것보다 더 많은 소프트웨어 스레드를 요청하면 `std::threads`를 생성하거나 `std::async`를 호출하여 수행하는지 여부가 왜 중요합니까?" 중요한 이유는 `std::async`가 이 형식으로 호출될 때 (즉, 기본 시작 정책으로 - [항목 36 참조](#)) 새 소프트웨어 스레드를 생성할 것이라고 보장하지 않기 때문입니다. 오히려 스케줄러가 `doAsyncWork`의 결과를 요청하는 스레드에서(즉, `get` 또는 `wait on fut`를 호출하는 스레드에서) 지정된 함수(이 예에서는 `doAsyncWork`)가 실행되도록 정렬하고 합리적인 스케줄러는 이러한 자유를 활용합니다. 시스템이 초과 구독되었거나 스레드가 부족한 경우.

이 "결과가 필요한 스레드에서 실행" 트릭을 직접 가져오면 로드 밸런싱 문제로 이어질 수 있으며 이러한 문제는 `std::async` 및 런타임 스케줄러가 직면하기 때문에 사라지지 않습니다. 당신 대신 그들. 그러나 로드 밸런싱과 관련하여 런타임 스케줄러는 코드가 실행 중인 프로세스뿐만 아니라 모든 프로세스의 스레드를 관리하기 때문에 사용자보다 시스템에서 일어나는 일에 대해 더 포괄적인 그림을 가질 수 있습니다.

`std::async`를 사용하면 스케줄러가 응답 요구 사항이 엄격한 스레드를 알 수 있는 방법이 없기 때문에 GUI 스레드의 응답성은 여전히 문제가 될 수 있습니다. 이 경우 `std::launch::async` 시작 정책을 `std::async`에 전달하고 싶을 것입니다. 그러면 실행하려는 함수가 실제로 다른 스레드에서 실행됩니다([항목 36 참조](#)).

최첨단 스레드 스케줄러는 시스템 전체 스레드 풀을 사용하여 초과 구독을 방지하고 작업 도용 알고리즘을 통해 하드웨어 코어 간의 로드 밸런싱을 개선합니다. C++ 표준은 스레드 풀이나 작업 훔치기의 사용을 요구하지 않으며, 솔직히 말해서 C++11 동시성 사양의 몇 가지 기술적 측면에서 우리가 원하는 것보다 이를 사용하기 더 어렵게 만듭니다..

그럼에도 불구하고 일부 공급업체는 표준 라이브러리 구현에서 이 기술을 활용하며 이 분야에서 계속 발전할 것으로 예상하는 것이 합리적입니다. 동시 프로그래밍에 작업 기반 접근 방식을 취하면 이러한 기술이 널리 보급됨에 따라 자동으로 이점을 얻을 수 있습니다. 반면에 `std::threads`를 사용하여 직접 프로그래밍하는 경우 이러한 문제에 대한 솔루션이 `std::threads`에 구현된 솔루션과 어떻게 맞물리는지는 말할 것도 없이 스레드 고갈, 초과 구독 및 로드 밸런싱을 처리하는 부담을 스스로 떠맡게 됩니다. 동일한 시스템의 다른 프로세스에서 실행 중인 프로그램

스레드 기반 프로그래밍과 비교할 때 작업 기반 설계는 수동 스레드 관리의 수고를 덜어주고 비동기적으로 실행된 함수(즉, 반환 값 또는 예외)의 결과를 검사하는 자연스러운 방법을 제공합니다. 그럼에도 불구하고-

덜하지만 스레드를 직접 사용하는 것이 적절한 상황이 있습니다. 여기에는 다음이 포함됩니다.

- 기본 스레딩 구현의 API에 액세스해야 합니다. C++ 동시성 API는 일반적으로 낮은 수준의 플랫 폼별 API, 일반적으로 pthreads 또는 Windows의 Threads를 사용하여 구현됩니다. 이러한 API는 현재 C++에서 제공하는 것보다 풍부합니다. (예를 들어, C++에는 스레드 우선 순위나 선호도에 대한 개념이 없습니다.) 기본 스레딩 구현의 API에 대한 액세스를 제공하기 위해 std::thread 객체는 일반적으로 native_handle 멤버 함수를 제공합니다. std::futures에 대한 이 기능에 대응하는 것은 없습니다(즉, std::async가 반환하는 것).
- 애플리케이션에 대한 스레드 사용을 최적화해야 하고 최적화할 수 있습니다. 예를 들어 고정된 하드웨어 특성을 가진 시스템에서 유일한 중요한 프로세스로 배포될 알려진 실행 프로필을 사용하여 서버 소프트웨어를 개발하는 경우가 이에 해당합니다.
- C++ 동시성 API를 넘어서는 스레딩 기술을 구현해야 합니다(예: C++ 구현에서 제공하지 않는 플랫폼의 스레드 풀).

그러나 이러한 경우는 흔하지 않습니다. 대부분의 경우 스레드로 프로그래밍하는 대신 작업 기반 디자인을 선택해야 합니다.

기억할 사항 •

std::thread API는 비동기식으로 실행되는 함수에서 반환 값을 가져오는 직접적인 방법을 제공하지 않으며 해당 함수가 throw되면 프로그램이 종료됩니다.

- 스레드 기반 프로그래밍은 스레드 소진, 초과 구독, 로드 밸런싱 및 새로운 플랫폼에 대한 적응의 수동 관리를 요구합니다. • 기본 시작 정책을 사용하는 std::async를 통한 작업 기반 프로그래밍은 이러한 문제의 대부분을 자동으로 처리합니다.

항목 36: 비동기성이 필수적인 경우 std::launch::async 를 지정하십시오.

함수(또는 다른 호출 가능한 객체)를 실행하기 위해 std::async를 호출하면 일반적으로 함수를 비동기식으로 실행하려고 합니다. 그러나 그것이 반드시 std::async에 요청하는 것은 아닙니다. 실제로 std::async 시작 정책에 따라 함수를 실행하도록 요청하고 있습니다. 두 가지 표준 정책이 있으며 각각

std::launch 범위 열거형에서 열거자로 표시됩니다. (범위가 지정된 열거형에 대한 정보는 [항목 10](#) 을 참조하십시오 .) 함수 f가 실행을 위해 std::async에 전달된다고 가정하면,

- std::launch::async 시작 정책은 f가 비동기식으로 실행되어야 함을 의미합니다.
즉, 다른 스레드에서.
- std::launch::deferred 시작 정책은 f가 std::async에서 반환된 future에 대해 get 또는 wait가 호출될 때만 실행될 수 있음을 의미합니다. 2 즉, f의 실행은 그러 한 호출이 이루어질 때까지 연기됩니다. get 또는 wait가 호출되면 f는 동기적으로 실행됩니다. 즉, 호출자는 f가 실행을 마칠 때까지 차단됩니다. get이나 wait가 호출되지 않으면 f는 실행되 지 않습니다.

아마도 놀랍게도 std::async의 기본 시작 정책(명시적으로 지정하지 않은 경우 사용하는 정책)은 둘 다 아닙니다. 오히려, 이것들이 또는 함께 있습니다. 다음 두 호출은 정확히 같은 의미를 갖습니다.

```
자동 fut1 = std::async(f); // 기본 시작 정책을 사용하여 // f를 실행합니다.
```

```
auto fut2 = std::async(std::launch::async | // f를 실행 std::launch::deferred, // async 또는  
f); // 지연
```

따라서 기본 정책은 f가 비동기식 또는 동기식으로 실행되도록 허용합니다.

[항목 35](#) 에서 지적한 것처럼 이러한 유연성 덕분에 std::async 및 표준 라이브러리의 스레드 관리 구 성 요소가 스레드 생성 및 소멸, 초과 구독 방지, 로드 밸런싱에 대한 책임을 맡을 수 있습니다 . 이것이 std::async를 사용한 동시 프로그래밍을 매우 간단하게 만드는 것 중 하나입니다.

니앙.

그러나 기본 시작 정책과 함께 std::async를 사용하면 몇 가지 흥미로운 의미가 있습니다. 이 명령문 을 실행하는 스레드 t가 주어졌을 때,

```
자동 fut = std::async(f); // 기본 시작 정책을 사용하여 f 실행
```

2 이것은 단순화입니다. 중요한 것은 get 또는 wait가 호출되는 future가 아니라 future가 참조하는 공유 상태입니다. ([항목 38](#) 은 future와 공유 상태 사이의 관계에 대해 설명합니다.) std::futures는 이동을 지원하고 std::shared_futures를 구성하는 데 사 용할 수도 있고 std::shared_futures를 복사할 수 있기 때문에 future 객체는 공유 상태를 참조합니다. f가 전달된 std::async 에 대한 호출에서 발생하는 것은 std::async에서 반환된 것과 다를 수 있습니다. 그러나 그것은 한 입에 불과하므로 진실을 왜곡하 고 단순히 std::async에서 반환되는 미래에 대해 get 또는 wait를 호출하는 것에 대해 이야기하는 것이 일반적입니다.

- f 가 지역 실행되도록 예약될 수 있기 때문에 f가 t 와 동시에 실행되는지 여부를 예측할 수 없습니다 .
- f가 get 또는 wait on fut를 호출하는 스레드와 다른 스레드에서 실행되는지 여부를 예측할 수 없습니다. 해당 스레드가 t이면 f가 t와 다른 스레드에서 실행되는지 여부를 예측할 수 없다는 의미입니다.
- 프로그램의 모든 경로를 따라 fut에서 get 또는 wait가 호출되는 것을 보장할 수 없기 때문에 f가 실행되는지 여부를 전혀 예측 하지 못할 수도 있습니다.

기본 시작 정책의 스케줄링 유연성은 종종 thread_local 변수의 사용과 잘 섞이지 않습니다. 왜냐하면 f가 이러한 스레드 로컬 저장소(TLS)를 읽거나 쓰는 경우 액세스할 스레드 변수를 예측할 수 없기 때문입니다.

```
자동 fut = std::async(f); // f에 대한 TLS는 // 독립 스레드의 경
                           // 우 // 가능하지만 // 스레드 의 경우 // fut
                           //에서 get 또는 wait 호출
```

지연된 작업(항목 35 참조)에서 wait_for 또는 wait_until을 호출 하면 std::launch::deferred 값이 생성 되기 때문에 시간 초과를 사용하는 대기 기반 루프에도 영향을 줍니다 . 이것은 결국 종료되어야 하는 것처럼 보이는 다음 루프가 실제로는 영원히 실행될 수 있음을 의미합니다.

네임스페이스 std::literals 사용	// C++14 기간의 경우 // 접미사;
	항목 34 참조

무효 f()	// f는 1초 동안 휴면하고, // 그 후 반환
{ std::this_thread::sleep_for(1s); }	

자동 fut = std::async(f);	// f를 비동기적으로 실행 // (개념적으 로)
동안 (fut.wait_for(100ms) != std::future_status::ready) { ... }	// f가 // 실행을 완료할 때까지 루프를 돌립니다... // 절대 일어나 지 않을 것입니다!

f가 std::async를 호출하는 스레드와 동시에 실행되는 경우(즉, 시작 정책이 f에 대해 선택된 것은 std::launch::async임), 여기에는 문제가 없습니다(f 가정 결국 완료), 그러나 f가 지연되면 fut.wait_for는 항상 std::l을 반환합니다. future_status::지연. 그것은 결코 std::future_status::와 같지 않을 것입니다. 루프가 종료되지 않습니다.

이러한 종류의 버그는 개발 및 단위 테스트 중에 간과하기 쉽습니다. 무거운 하중에서만 나타날 수 있습니다. 밀어붙이는 조건들이다. 초과 구독 또는 스레드 고갈을 향한 기계, 그리고 그 때 작업은 연기될 가능성이 높습니다. 결국 하드웨어가 과잉 공급에 의해 위협받지 않는다면 스크립션 또는 스레드 고갈, 런타임 시스템이 예약되지 않을 이유가 없습니다. 동시 실행 작업을 ule.

수정은 간단합니다. std::async 호출에 해당하는 미래를 확인하기만 하면 됩니다. 작업이 지연되는지 여부, 그렇다면 시간 초과 기반 루프에 들어가는 것을 피하십시오. 불행히도, 미래의 작업이 연기되었는지 여부를 묻는 직접적인 방법은 없습니다. 대신 wait_for와 같은 시간 제한 기반 함수를 호출해야 합니다. ~ 안에 이 경우에는 아무 것도 기다리지 않고 반환 여부를 확인하기만 하면 됩니다. 값은 std::future_status::deferred이므로 필요에 따라 약간의 불신을 얹으십시오-sary circumlocution 및 0 시간 초과로 wait_for 호출:

```

자동 fut = std::async(f);                                // 위와 같이

if (fut.wait_for(0s) ==                                // 태스크가 있는 경우
    std::future_status::deferred)                      // 연기...
{
    ...                                                 // ...기다림을 사용하거나 fut
    ...                                                 // f를 동기적으로 호출하기 위해

} else { // 작업이 지연되지 않음
    동안 (fut.wait_for(100ms) !=                      // 무한 루프 아님
          std::future_status::ready) {                  // 가능(가정
        ...                                                 // f완료)

        ...                                                 // 작업이 지연되거나 준비되지 않았습니다.
        ...                                                 // 준비가 될 때까지 동시 작업을 수행합니다.

    }
    ...
} // 끝이 준비되었습니다

```

이러한 다양한 고려 사항의 결과는 다음 조건이 충족되는 한 작업에 대한 기본 시작 정책과 함께 std::async를 사용하는 것이 좋다는 것입니다.

- 작업은 get 또는 wait를 호출하는 스레드와 동시에 실행할 필요가 없습니다. • 어떤 스레드의 thread_local 변수를 읽거나 쓰는지는 중요하지 않습니다.
- 반환된 미래에 대해 get 또는 wait가 호출된다라는 보장이 있습니다. std::async 또는 작업이 실행되지 않는 것이 허용됩니다.
- wait_for 또는 wait_until을 사용하는 코드는 지연된 상태의 가능성을 계정.

이러한 조건 중 하나라도 유지되지 않으면 std::async가 진정한 비동기 실행을 위해 작업을 예약하도록 보장하고 싶을 것입니다. 그렇게 하는 방법은 호출할 때 std::launch::async를 첫 번째 인수로 전달하는 것입니다.

```
자동 fut = std::async(std::launch::async, f); // f 실행
// 비동기적으로
```

사실 std::async처럼 작동하지만 자동으로 std::launch::async를 시작 정책으로 사용하는 기능을 갖는 것은 주변에 편리한 도구이므로 작성하기 쉽다는 것이 좋습니다. C++11 버전은 다음과 같습니다.

```
template<typename F, typename... Ts> inline
std::future<typename std::result_of<F(Ts...)>::type>
realAsync(F&& f, Ts&&... params) {
    // 미래를 반환 // 비동기
    // 경우
    return std::async(std::launch::async, // f(params...) 호출
                     std::forward<F>(f),
                     std::forward<Ts>(params)...);
}
```

이 함수는 호출 가능한 객체 f와 0개 이상의 매개변수 params를 수신하고 시작 정책으로 std::launch::async를 전달하여 std::async로 완벽하게 전달합니다([항목 25 참조](#)). std::async와 마찬가지로 params에서 f를 호출한 결과에 대해 std::future를 반환합니다. 그 결과의 유형을 결정하는 것은 쉽습니다. 유형 특성 std::result_of가 제공하기 때문입니다. (유형 특성에 대한 일반 정보는 [항목 9](#) 를 참조하십시오.)

realAsync는 std::async처럼 사용됩니다.

```
자동 fut = realAsync(f);
// f를 비동기적으로 실행합니다. //
std::async // 던지면 던짐
```

C++14에서 `realAsync`의 반환 유형을 추론하는 기능은 함수 선언을 간소화합니다.

`template<typename F, typename... Ts> inline`

```
자동 // C++14
realAsync(F&& f, Ts&&... 매개변수) {

    return std::async(std::launch::async, std::forward<F>(f),
                      std::forward<Ts>(params)...);

}
```

이 버전은 `realAsync`가 `std::launch::async` 시작 정책으로 `std::async`를 호출하는 것 외에는 아무 것도 하지 않는다는 것을 분명히 합니다.

기억할 사항 • `std::async`

의 기본 시작 정책은 비동기 및
동기 작업 실행.

- 이러한 유연성은 `thread_locals`에 액세스할 때 불확실성을 초래하고 작업이 실행되지 않을 수 있음을 의미하며 시간 초과 기반 대기 호출에 대한 프로그램 논리에 영향을 줍니다.
- 비동기 작업 실행이 필수적인 경우 `std::launch::async`를 지정합니다.

항목 37: 모든 경로에서 `std::threads` 를 결합할 수 없도록 만드세요.

모든 `std::thread` 객체는 결합 가능 또는 결합 불가능의 두 가지 상태 중 하나입니다. 결합 가능한 `std::thread`는 실행 중이거나 실행될 수 있는 기본 비동기 실행 스레드에 해당합니다. 예를 들어 차단되었거나 예약 대기 중인 기본 스레드에 해당하는 `std::thread`는 조인 가능합니다. `std::thread` 객체가 완료될 때까지 실행된 기본 스레드에 해당하는 객체도 결합 가능한 것으로 간주됩니다.

결합할 수 없는 `std::thread`는 예상한 대로 결합할 수 없는 `std::thread`입니다.

조인할 수 없는 `std::thread` 개체에는 다음이 포함됩니다.

- 기본 구성 `std::threads`. 이러한 `std::threads`에는 다음과 같은 기능이 없습니다.
실행하므로 기본 실행 스레드에 해당하지 않습니다.

- std::thread 객체가 이동되었습니다. 이동의 결과는 (있는 경우) 해당하는 데 사용되는 std::thread 실행의 기본 스레드가 이제 다른 std::thread에 해당한다는 것입니다.
- std::결합된 스레드. 조인 후 std::thread 객체는 더 이상 실행이 완료된 기본 실행 스레드에 해당하지 않습니다.
- std::분리된 스레드. 분리는 std::thread 객체와 이에 해당하는 실행 스레드 간의 연결을 끊습니다.

std::thread의 결합 가능성이 중요한 한 가지 이유는 결합 가능한 스레드의 소멸자가 호출되면 프로그램 실행이 종료되기 때문입니다. 예를 들어 필터링 함수 filter와 최대값 maxVal을 매개변수로 취하는 함수 doWork가 있다고 가정합니다. doWork는 계산에 필요한 모든 조건이 충족되었는지 확인한 다음 필터를 통과하는 0과 maxVal 사이의 모든 값으로 계산을 수행합니다. 필터링을 하는 데 시간이 많이 걸리고 doWork의 조건이 만족되는지 여부를 확인하는 데도 시간이 걸린다면 이 두 가지를 동시에 수행하는 것이 합리적입니다.

이를 위해 작업 기반 설계를 선호하지만(항목 35 참조) 필터링을 수행하는 스레드의 우선 순위를 설정한다고 가정해 보겠습니다. 항목 35 는 스레드의 기본 핸들을 사용해야 하며 std::thread API를 통해서만 액세스할 수 있다고 설명합니다. 작업 기반 API(예: futures)는 이를 제공하지 않습니다.

따라서 우리의 접근 방식은 작업이 아닌 스레드를 기반으로 합니다.

다음과 같은 코드를 만들 수 있습니다.

```
constexpr 자동 1000만 = 10000000; // constexpr의 경
                                    // 우 // 항목 15 참조

bool doWork(std::function<bool(int)> filter, // 계산이 수행되었는지 여부를 // 반환합니다.
            int maxVal = 1000만) // 항목 2 for //
{
    std::vector<int> goodVals; // 필터를 만족하는 //
                                // 값

    std::thread t([&filter, maxVal, &goodVals] // 채우기 { // goodVals for (auto i
        = 0; i <= maxVal; ++i) { if (filter(i)) goodVals.push_back(i) ; }
}
```

```

}); // t의 기본 // 핸들을 사용
자동 nh = t.native_handle(); 하여 // t의 우선 순위를
... 설정합니다.

if (conditionsAreSatisfied()) { t.join(); 계산
    수행(goodVals); true를 반환합니다. // 끝내자

    // 계산이 // 수행됨
}

거짓을 반환합니다. // 계산이 // 수행되지 않음
}

```

이 코드가 문제가 되는 이유를 설명하기 전에 아포스트로피를 숫자 구분 기호로 사용하는 C++14의 기능을 활용하여 TenMillion의 초기화 값을 C++14에서 더 읽기 쉽게 만들 수 있다는 점을 언급하겠습니다.

```
constexpr 자동 1000만 = 10'000'000; // C++14
```

나는 또한 그것이 실행을 시작한 후 t의 우선 순위를 설정하는 것은 똑같이 속담이 볼트로 몰린 후에 속담의 헛간 문을 닫는 것과 약간 비슷하다는 점을 언급할 것입니다. 더 나은 설계는 일시 중단된 상태에서 t를 시작하는 것이지만(따라서 계산을 수행하기 전에 우선 순위를 조정할 수 있게 함) 해당 코드로 주의를 산만하게 하고 싶지 않습니다. 코드가 없어서 더 산만하다면 [항목 39](#)로 이동 하십시오. 중단된 스레드를 시작하는 방법을 보여주기 때문입니다.

그러나 할 일로 돌아가십시오. conditionAreSatisfied() 가 true를 반환 하면 모든 것이 정상 이지만 false를 반환하거나 예외를 throw하면 doWork의 끝에서 소멸자가 호출될 때 std::thread 객체 t가 조인 가능합니다. 그러면 프로그램 실행이 종료됩니다.

std::thread 소멸자가 왜 이런 식으로 동작하는지 궁금할 것입니다. 다른 두 가지 명백한 옵션이 틀림없이 더 나쁘기 때문입니다. 그들은:

- **암시적 조인.** 이 경우 std::thread의 소멸자는 기본 비동기 실행 스레드가 완료될 때까지 기다립니다. 이는 합리적으로 들리지만 추적하기 어려운 성능 이상으로 이어질 수 있습니다. 예를 들어, conditionAreSatisfied() 가 이미 false를 반환한 경우 doWork가 필터가 모든 값에 적용될 때까지 기다리는 것은 직관적이지 않습니다 .
- **암시적 분리.** 이 경우 std::thread의 소멸자는 std::thread 객체와 기본 실행 스레드 간의 연결을 끊습니다. 기본 스레드는 계속 실행됩니다. 이것은 덜 이유가 될 것입니다-

조인 접근 방식보다 가능하지만 디버깅 문제가 더 악화될 수 있습니다. 예를 들어 doWork에서 goodVals는 참조로 캡처되는 지역 변수입니다. 또한 람다 내부에서 수정됩니다(push_back 호출을 통해). 그러면 람다가 비동기식으로 실행되는 동안 conditionsAreSatisfied() 가 false를 반환한다고 가정합니다. 이 경우 doWork가 반환되고 해당 지역 변수(goodVals 포함)가 소멸됩니다. 스택 프레임이 팝되고 해당 스레드의 실행은 doWork의 호출 사이트에서 계속됩니다.

해당 호출 사이트 다음의 명령문은 어느 시점에서 추가 함수 호출을 수행할 것이며, 이러한 호출 중 적어도 하나는 아마도 doWork 스택 프레임이 한 번 점유했던 메모리의 일부 또는 전체를 사용하게 될 것입니다. 그런 함수를 f라고 하자. f가 실행되는 동안 doWork가 시작한 람다는 여전히 비동기식으로 실행됩니다. 그 람다는 goodVals에 사용되던 스택 메모리에서 push_back을 호출할 수 있지만 이제는 f의 스택 프레임 내부 어딘가에 있습니다. 이러한 호출은 goodVals에 사용된 메모리를 수정하며 이는 f의 관점에서 스택 프레임의 메모리 내용이 자발적으로 변경될 수 있음을 의미합니다! 디버깅할 때의 재미를 상상해 보십시오.

표준화 위원회는 결합 가능한 스레드를 파괴하는 결과가 충분히 끔찍하여 본질적으로 금지한다고 결정했습니다(결합 가능한 스레드의 파괴가 프로그램 종료를 유발한다고 명시함으로써).

이렇게 하면 std::thread 개체를 사용하는 경우 개체가 정의된 범위를 벗어난 모든 경로에서 결합할 수 없도록 해야 하는 책임이 있습니다. 그러나 모든 경로를 다루는 것은 복잡할 수 있습니다. 여기에는 범위 끝에서 벗어나는 것과 return, continue, break, goto 또는 예외를 통해 점프하는 것이 포함됩니다. 많은 길이 될 수 있습니다.

블록의 모든 경로를 따라 어떤 작업을 수행하려는 경우 일반적인 접근 방식은 해당 작업을 로컬 개체의 소멸자에 넣는 것입니다. 이러한 개체를 RAII 개체라고 하고 개체가 가져온 클래스를 RAII 클래스라고 합니다.

(RAII 자체는 "Resource Acquisition Is Initialization"을 의미하지만 기술의 핵심은 초기화가 아니라 파괴입니다.) RAII 클래스는 표준 라이브러리에서 일반적입니다. 예를 들면 STL 컨테이너(각 컨테이너의 소멸자는 컨테이너의 내용을 파괴하고 메모리를 해제함), 표준 스마트 포인터(항목 18-20은 std::unique_ptr의 소멸자가 가리키는 객체에서 삭제자를 호출한다고 설명합니다. std::shared_ptr 및 std::weak_ptr 감소 참조 카운트), std::fstream 객체(해당 소멸자가 해당 파일을 닫음) 등. 그러나 std::thread 객체에 대한 표준 RAII 클래스는 없습니다. 아마도 Standardization Committee가 기본 옵션으로 join과 detach를 모두 거부했기 때문에 그러한 클래스가 무엇을 해야 하는지 몰랐기 때문일 것입니다.

다행히 직접 작성하는 것은 어렵지 않습니다. 예를 들어 다음 클래스를 사용하면 호출자가 ThreadRAII 개체(std::thread에 대한 RAII 개체)가 파괴될 때 결합 또는 분리를 호출해야 하는지 여부를 지정할 수 있습니다.

```
클래스 ThreadRAII { 공개:
    열거형 클래스 DtorAction
    { 조인, 분리 };
                                            // 열거형 클래스 정보는 // 항
                                            목 10 참조
```

```
ThreadRAII(std::thread&& t, DtorAction a) // dtor에서 take : action(a),
t(std::move(t)) {} // t에 대한 액션 a
```

```
~ThreadRAII() { if
    (t.joinable()) {
                                            // 아래에서 // 결합 가능
                                            성 테스트를 참조하십시오.
```

```
    if (액션 == DtorAction::join) { t.join(); } else { t.
        분리(); }
```

```
}
```

```
std::thread& get() { 리턴 t; } // 아래 참조
```

```
사적인:
DtorAction 액션; 표준::스
레드 t; };
```

이 코드가 설명이 잘 되었으면 좋겠지만 다음 사항이 도움이 될 수 있습니다.

- 전달된 std::thread를 ThreadRAII 개체로 이동하려고 하기 때문에 생성자는 std::thread rvalue만 허용합니다. (std::thread 객체는 복사할 수 없음을 기억하십시오.) • 생성자의 매개변수 순서는 호출자에게 직관적으로 설계되었습니다(std::thread를 먼저 지정하고 소멸자 작업을 두 번째로 지정하는 것이 그 반대의 경우보다 더 합리적임). 멤버 초기화 목록은 데이터 멤버 선언의 순서와 일치하도록 설계되었습니다. 그 순서는 std::thread 객체를 마지막에 둡니다. 이 클래스에서 순서는 차이가 없지만 일반적으로 한 데이터 멤버의 초기화가 다른 멤버에 종속될 수 있으며 std::thread 개체가 실행된 직후에 함수 실행을 시작할 수 있기 때문에

초기화되면 클래스에서 마지막으로 선언하는 것이 좋습니다. 이는 생성된 시점에 선행하는 모든 데이터 멤버가 이미 초기화되어 있으므로 std::thread 데이터 멤버에 해당하는 비동기적으로 실행 중인 스레드에서 안전하게 액세스할 수 있음을 보장합니다. • ThreadRAII는 기본 std::thread 객체에 대한 액세스를 제공하는 get 함수를 제공합니다. 이것은 기본 원시 포인터에 대한 액세스를 제공하는 표준 스마트 포인터 클래스에서 제공하는 get 함수와 유사합니다. get을 제공하면 ThreadRAII가 전체 std::thread 인터페이스를 복제할 필요가 없으며 ThreadRAII 개체가 std::thread 개체가 필요한 컨텍스트에서 사용될 수 있음을 의미합니다.

- ThreadRAII 소멸자가 멤버 함수를 호출하기 전에

std::thread 객체 t, t가 조인 가능한지 확인합니다. 조인할 수 없는 스레드에서 조인 또는 분리를 호출하면 정의되지 않은 동작이 생성되기 때문에 이것이 필요합니다. 클라이언트가 std::thread를 구성하고, 이 개체에서 ThreadRAII 개체를 만들고, get을 사용하여 t에 대한 액세스 권한을 얻은 다음, t에서 이동하거나 연결 또는 분리를 호출했을 수 있습니다. 이러한 각 작업은 t를 결합할 수 없게 만듭니다.

이 코드에서

```
if (t.joinable()) {
    if (액션 == DtorAction::join) { t.join(); } else { t.
    분리(); } }
```

t.joinable() 실행과 조인 또는 분리 호출 사이에 다른 스레드가 t를 조인할 수 없도록 만들 수 있기 때문에 경쟁이 존재합니다. 직관은 칭찬할 만하지만 두려움은 근거가 없습니다. std::thread 객체는 멤버 함수 호출(예: 결합, 분리 또는 이동 작업)을 통해서만 결합 가능에서 결합 불가능으로 상태를 변경할 수 있습니다. ThreadRAII 개체의 소멸자가 호출될 때 다른 스레드가 해당 개체에 대해 멤버 함수를 호출해서는 안 됩니다.

동시 호출이 있는 경우 확실히 경쟁이 있지만 소멸자 내부가 아니라 한 개체에서 두 개의 멤버 함수(소멸자와 다른 것)를 동시에 호출하려고 하는 클라이언트 코드에 있습니다. 일반적으로 단일 개체에 대한 동시 멤버 함수 호출은 모두가 const 멤버 함수인 경우에만 안전합니다(항목 16 참조).

doWork 예제에서 ThreadRAII를 사용하면 다음과 같습니다.

```

bool doWork(std::function<bool(int)> filter, // 이전과 같이 int maxVal = tenMillion)

{
    표준::벡터<int> goodVals; // 이전과

    ThreadRAII // RAIi 객체 사용
    t( std::thread([& 필터, maxVal, &goodVals]

        { for (자동 i = 0; i <= maxVal; ++i) { if (filter(i))
            goodVals.push_back(i); }
        },
        ThreadRAII::DtorAction::join // RAIi 작업
    );
}

자동 nh = t.get().native_handle();
...

if (조건 만족()) { t.get().join(); 계산 수행
    (goodVals); true를 반환합니다.

}

거짓을 반환합니다.
}

```

이 경우 ThreadRAII 소멸자에서 비동기적으로 실행 중인 스레드에서 조인을 선택했습니다. 앞서 보았듯이 분리를 수행하면 정말 악몽 같은 디버깅이 발생할 수 있기 때문입니다. 우리는 또한 조인을 하면 성능 이상(솔직히 말하면 디버그하기에도 불쾌할 수 있음)이 발생할 수 있지만 정의되지 않은 동작(분리하면 얻을 수 있음), 프로그램 종료(사용 raw std::thread는 다음을 낳을 것입니다.) 또는 성능 이상, 성능 이상은 최악의 경우 중 가장 좋은 것 같습니다.

아아, 항목 39 는 ThreadRAII를 사용하여 std::thread 파괴에 대한 조인을 수행하는 것이 때때로 성능 이상뿐만 아니라 정지된 프로그램으로 이어질 수 있음을 보여줍니다. 이러한 종류의 문제에 대한 "적절한" 솔루션은 비동기적으로 실행되는 람다와 더 이상 작업이 필요하지 않고 조기 에 반환되어야 하지만 C++11에서는 인터럽트 가능에 대한 지원이 없음을 통신하는 것입니다.

스레드. 손으로 구현할 수도 있지만 이는 이 책의 범위를 벗어난 주제입니다.³

항목 17은 ThreadRAII가 소멸자를 선언하기 때문에 컴파일러에서 생성한 이동 작업이 없지만 ThreadRAII 객체가 이동되지 않아야 할 이유가 없다고 설명합니다. 컴파일러가 이러한 함수를 생성하는 경우 함수는 올바른 작업을 수행하므로 명시적으로 생성을 요청하는 것이 적절합니다.

```
클래스 ThreadRAII { 공개:
    열거형 클래스 DtorAction
    { 조인, 분리 };                                // 이전과

    ThreadRAII(std::thread&& t, DtorAction a) : action(a),
    t(std::move(t)) {}                            // 이전과

    ~ThreadRAII() {

        ...                                         // 이전과
    }

    ThreadRAII(ThreadRAII&&) = 기본값;           // 지원 // 이동
    ThreadRAII& 연산자=(ThreadRAII&&) = 기본값;

    std::thread& get() { 리턴 t; }                  // 이전과

    사적인:
        DtorAction 액션; 표준::스
        레드 t; };                                // 이전과
```

기억할 사항 •

- std::threads를 모든 경로에서 결합할 수 없도록 만듭니다.
- Join-On-Destruction은 디버그하기 어려운 성능 이상으로 이어질 수 있습니다. • detach-on-destruction은 디버그하기 어려운 정의되지 않은 동작으로 이어질 수 있습니다.
- std::thread 객체를 데이터 멤버 목록에서 마지막으로 선언합니다.

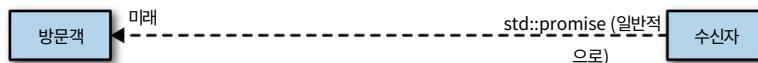
³ Anthony Williams의 C++ Concurrency in Action(Manning Publications, 2012)에서 좋은 치료법을 찾을 수 있습니다.
섹션 9.2.

항목 38: 다양한 스레드 핸들 소멸자 동작에 주의하십시오.

항목 37은 결합 가능한 std::thread가 실행의 기본 시스템 스레드에 해당한다고 설명합니다. 자연되지 않은 작업의 future(항목 36 참조)는 시스템 스레드와 유사한 관계를 갖습니다. 따라서 std::thread 객체와 미래 객체는 모두 시스템 스레드에 대한 핸들로 생각할 수 있습니다.

이러한 관점에서 보면 std::threads와 future가 소멸자에서 이처럼 다른 동작을 한다는 것이 흥미롭습니다. 항목 37에서 언급했듯이 조인 가능한 std::thread를 파괴하면 프로그램이 종료됩니다. 두 가지 명백한 대안(암시적 조인 및 암시적 분리)이 더 나쁜 선택으로 간주되었기 때문입니다. 그러나 미래에 대한 소멸자는 때때로 암시적 조인을 수행하는 것처럼, 때로는 암시적 분리를 수행한 것처럼, 때로는 둘 다 수행하지 않는 것처럼 작동합니다. 프로그램 종료를 일으키지 않습니다. 이 스레드 핸들 동작 부야베스는 더 자세히 살펴볼 가치가 있습니다.

우리는 미래가 호출 수신자가 호출자에게 결과를 전송하는 통신 채널의 한쪽 끝이라는 관찰로 시작할 것입니다.⁴ 호출 수신자(보통 비동기식으로 실행)는 계산 결과를 통신 채널에 씁니다. (일반적으로 std::promise 객체를 통해) 호출자는 future를 사용하여 해당 결과를 읽습니다. 점선 화살표는 수신자에서 호출자로의 정보 흐름을 나타내는 다음과 같이 생각할 수 있습니다.



그러나 수신자의 결과는 어디에 저장됩니까? 호출자가 호출하기 전에 호출자가 완료될 수 있음

해당하는 미래에 도착하므로 결과를 수신자의 std::promise에 저장할 수 없습니다. 호출 수신자에게 로컬인 해당 객체는 호출 수신자가 완료되면 소멸됩니다.

결과는 호출자의 미래에 저장할 수 없습니다. 왜냐하면 (다른 이유로) std::future가 std::shared_future를 생성하는 데 사용될 수 있기 때문입니다(따라서 호출 수신자의 결과 소유권을 std::future에서 std::shared_future), 원본 std::future가 파괴된 후 여러 번 복사될 수 있습니다.

모든 결과 유형을 복사할 수 있는 것은 아니며(즉, 이동 전용 유형) 결과가

4 항목 39는 미래와 관련된 통신 채널의 종류가 다른 용도로 사용될 수 있음을 설명합니다. 그러나 이 항목의 경우 호출 수신자가 호출자에게 결과를 전달하는 메커니즘으로만 사용할 것입니다.

적어도 그것을 참조하는 마지막 미래만큼은 살아 있어야 합니다. 호출 수신자에 해당하는 잠재적으로 많은 미래 중 어느 것이 결과를 포함해야 합니까?

호출 수신자와 연결된 개체나 호출자와 연결된 개체 모두 호출 수신자의 결과를 저장하기에 적합한 위치가 아니므로 둘 다 외부의 위치에 저장됩니다. 이 위치를 공유 상태라고 합니다. 공유 상태는 일반적으로 힙 기반 개체로 표시되지만 해당 유형, 인터페이스 및 구현은 표준에서 지정되지 않습니다. 표준 라이브러리 작성자는 원하는 방식으로 공유 상태를 자유롭게 구현할 수 있습니다.

호출 수신자, 호출자 및 공유 상태 간의 관계를 다음과 같이 상상할 수 있습니다. 여기서 점선 화살표는 다시 한 번 정보의 흐름을 나타냅니다.



이 항목의 주제인 future 소멸자의 동작은 future와 관련된 공유 상태에 의해 결정되기 때문에 공유 상태의 존재가 중요합니다. 특히,

- std::async를 통해 시작된 지연되지 않은 작업의 공유 상태를 참조하는 마지막 미래의 소멸자는 작업이 완료될 때까지 차단됩니다. 본질적으로 그러한 미래를 위한 소멸자는 비동기적으로 실행 중인 작업이 실행 중인 스레드에서 암시적 조인을 수행합니다. • 다른 모든 future에 대한 소멸자는 단순히 future 객체를 소멸시킵니다. 비동기적으로 실행되는 작업의 경우 이는 기본 스레드에 대한 암시적 분리와 유사합니다. 이것이 최종 미래인 지연된 작업의 경우 지연된 작업이 실행되지 않음을 의미합니다.

이 규칙은 생각보다 복잡하게 들립니다. 우리가 실제로 다루고 있는 것은 단순한 "정상적인" 행동과 이에 대한 단 하나의 예외입니다. 일반적인 동작은 future의 소멸자가 future 객체를 파괴하는 것입니다. 그게 다야 어떤 것과도 결합하지 않고, 어떤 것과도 분리하지 않으며, 아무 것도 실행하지 않습니다. 미래의 데이터 멤버를 파괴할 뿐입니다. (사실, 그것은 한 가지 더 많은 일을 합니다. 그것은 그것을 참조하는 future와 수신자의 std::promise에 의해 조작되는 공유 상태 내부의 참조 횟수를 감소시킵니다. 이 참조 횟수는 라이브러리가 알 수 있도록 합니다. 공유 상태가 파괴될 수 있는 경우 참조 카운팅에 대한 일반적인 정보는 [항목 19](#)를 참조하십시오.)

이 정상적인 행동에 대한 예외는 다음 사항이 모두 적용되는 미래에 대해서만 발생합니다.

- std::async 호출로 인해 생성된 공유 상태를 나타냅니다. • 작업의 시작 정책은 std::launch::async입니다 (항목 36 참조). 런타임 시스템에서 선택했거나 std::async에 대한 호출에서 지정했기 때문입니다.
- future는 공유 상태를 나타내는 마지막 future입니다. std::futures의 경우 항상 그렇습니다. std::shared_futures의 경우 다른 std::shared_futures가 소멸되는 미래와 동일한 공유 상태를 참조하는 경우 소멸되는 future는 정상적인 동작을 따릅니다(즉, 단순히 데이터 멤버를 소멸).

이러한 모든 조건이 충족될 때만 future의 소멸자는 특별한 동작을 나타내며 그 동작은 비동기적으로 실행 중인 작업이 완료될 때까지 차단됩니다. 실제로 이것은 std::async 생성 작업을 실행하는 스레드와의 암시적 조인에 해당합니다.

"Futures from std::async block in their destructors"로 요약된 정상적인 미래 소멸자 동작에 대한 이 예외를 듣는 것은 일반적입니다. 첫 번째 근사값은 정확하지만 때로는 첫 번째 근사값보다 더 많은 것이 필요합니다. 이제 당신은 그 모든 영광과 경이로움 속에 있는 진리를 압니다.

당신의 경이는 다른 형태를 취할 수 있습니다. "std::async에 의해 시작되는 지연되지 않은 작업에 대한 공유 상태에 대한 특별한 규칙이 있는 이유가 궁금합니다." 합리적인 질문입니다. 내가 말할 수 있는 바에 따르면, 표준화 위원회는 암시적 분리(항목 37 참조)와 관련된 문제를 피하기를 원했지만 필수 프로그램 종료와 같은 급진적인 정책을 채택하기를 원하지 않았습니다(가입 가능한 std에서 그랬던 것처럼). ::threads—다시, 항목 37 참조), 암시적 조인에서 타협했습니다. 이 결정은 논란의 여지가 없었고 C++14에서 이 동작을 포기하는 것에 대해 진지한 논의가 있었습니다. 결국 변경 사항이 없으므로 future에 대한 소멸자의 동작은 C++11 및 C++14에서 일관됩니다.

API for futures는 future가 std::async에 대한 호출로 인해 발생하는 공유 상태를 참조하는지 여부를 결정할 방법을 제공하지 않으므로 임의의 future 객체가 주어지면 소멸자를 비동기식으로 기다리는 소멸자에서 차단할지 여부를 알 수 없습니다. 작업을 완료하기 위해 실행 중입니다. 여기에는 몇 가지 흥미로운 의미가 있습니다.

```
// 하나 이상의 포함된 future가 std::async를 통해 시작된 지연되지 않은 작업에 대한 공유 상태
를 // 참조할 수 있기 때문에 이 컨테이너 는 dtor에서 차단 될 수 있습니다 .
std::vector<std::future<void> > 뜯내기; // std::future<void>에 대한 정보는 // 항목 39 를
참조하세요 .
```

클래스 위젯 { 공개:

// 위젯 객체 는 // 해당 dtor에서 차
단 될 수 있습니다.

...

개인:

```
std::shared_future<더블> fut; };
```

물론 주어진 future가 특별한 소멸자 동작을 유발하는 조건을 충족하지 않는다는 것을 알 수 있는 방법이 있다면(예: 프로그램 논리로 인해) future가 소멸자에서 차단되지 않을 것임을 확신할 수 있습니다.. 예를 들어 std::async에 대한 호출에서 발생하는 공유 상태만 특수 동작에 적합하지만 공유 상태가 생성되는 다른 방법이 있습니다. 하나는 std::packaged_task를 사용하는 것입니다.

std::packaged_task 객체는 결과가 공유 상태가 되도록 래핑하여 비동기 실행을 위해 함수(또는 다른 호출 가능한 객체)를 준비합니다. 공유 상태를 참조하는 future는 std::packaged_task의 get_future 함수를 통해 얻을 수 있습니다.

```
정수 계산값(); // 실행할 함수

std::packaged_task<int()> // 비동기적으로 실행할 수 있도록 //
    pt(calcValue); calcValue를 래핑합니다.

자동 fut = pt.get_future(); // pt의 미래를 얻습니다.
```

이 시점에서 우리는 future fut가 std::async 호출에 의해 생성된 공유 상태를 참조하지 않는다는 것을 알고 있으므로 소멸자가 정상적으로 동작할 것입니다.

일단 생성되면 std::packaged_task pt를 스레드에서 실행할 수 있습니다. (std::async에 대한 호출을 통해 실행할 수도 있지만 std::async를 사용하여 작업을 실행하려는 경우 std::async가 모든 std를 수행하기 때문에 std::packaged_task를 만들 이유가 거의 없습니다. :packaged_task는 실행할 작업을 예약하기 전에 수행합니다.)

std::packaged_tasks는 복사할 수 없으므로 pt가 std::thread 생성자에 전달되면 rvalue로 캐스트되어야 합니다(std::move를 통해— [항목 23](#) 참조).

```
표준::스레드 t(표준::이동(pt)); // t에서 pt를 실행
```

이 예제는 미래 소멸자의 정상적인 동작에 대한 통찰력을 제공하지만 문이 블록 내부에 함께 있는지 확인하는 것이 더 쉽습니다.

```
{ // 시작 블록
```

```
std::packaged_task<int()>
    pt(calcValue);
```

```
자동 fut = pt.get_future();
```

```

표준::스레드 t(표준::이동(pt));
...
}
// 아래 참조
// 종료 블록

```

여기에서 가장 흥미로운 코드는 std::thread 객체 t 생성 후 블록의 끝 앞에 오는 "...입니다. 흥미로운 점은 "..." 영역 내부에서 t에 무슨 일이 일어날 수 있다는 것입니다. 세 가지 기본 가능성이 있습니다.

- t에는 아무 일도 일어나지 않습니다. 이 경우 t는 범위 끝에서 결합할 수 있습니다.
그러면 프로그램이 종료됩니다([항목 37](#) 참조).
- 조인은 t에서 수행됩니다. 이 경우 fut가 블록에서 차단할 필요가 없습니다.
조인이 이미 호출 코드에 존재하기 때문에 소멸자입니다.
- t에서 분리가 수행됩니다. 이 경우 fut가 분리될 필요가 없습니다.
호출 코드가 이미 그렇게 하기 때문에 소멸자입니다.

즉, std::packaged_task로 인해 발생한 공유 상태에 해당하는 future가 있는 경우 일반적으로 특별한 소멸 정책을 채택할 필요가 없습니다. std::packaged_task가 일반적으로 실행되는 std::thread를 조작하는 코드입니다.

기억할 사항 • Future

소멸자는 일반적으로 future의 데이터 멤버를 파괴합니다. • 실행된 지연되지 않은 작업에 대한 공유 상태를 참조하는 최종 미래
std::async를 통해 작업이 완료될 때까지 차단됩니다.

항목 39: 일회성 이벤트 커뮤니케이션을 위해 무효 선물을 고려하십시오.

두 번째 작업은 이벤트가 발생할 때까지 진행할 수 없기 때문에 특정 이벤트가 발생했음을 비동기식으로 실행 중인 두 번째 작업에 작업에 알리는 것이 유용한 경우가 있습니다. 데이터 구조가 초기화되었거나 계산 단계가 완료되었거나 중요한 센서 값이 감지되었을 수 있습니다. 이 경우 이러한 종류의 스레드 간 통신을 수행하는 가장 좋은 방법은 무엇입니까?

명백한 접근 방식은 조건 변수(condvar)를 사용하는 것입니다. 조건을 감지하는 태스크를 감지 태스크와 조건에 반응하는 태스크를 호출하면

반응 작업, 전략은 간단합니다. 반응 작업은 조건 변수를 기다립니다.
이벤트가 발생하면 감지 스레드가 해당 condvar에 알립니다. 주어진

```
std::condition_variable 이력서; // 이벤트에 대한 condvar
```

```
표준::뮤텍스 m; // cv와 함께 사용할 뮤텍스
```

감지 작업의 코드는 다음과 같이 간단합니다.

```
... // 이벤트 감지
```

```
cv.notify_one(); // 반응하는 작업을 알려줍니다.
```

통지할 대응 작업이 여러 개 있는 경우 대체하는 것이 적절할 것입니다.

notify_one은 notify_all을 사용하지만 지금은 반응하는 것이 하나만 있다고 가정합니다.
직무.

반응하는 작업에 대한 코드는 대기를 호출하기 전에 조금 더 복잡합니다.

condvar에서는 std::unique_lock 객체를 통해 뮤텍스를 잠기야 합니다. (잠금
조건 변수를 기다리기 전의 뮤텍스는 스레딩 라이브러리에 일반적입니다. 그만큼
std::unique_lock 객체를 통해 뮤텍스를 잠글 필요는 단순히
C++11 API.) 다음은 개념적 접근 방식입니다.

```
... // 반응 준비
```

```
{ // 크리티컬 섹션 열기
```

```
std::unique_lock<std::mutex> lk(m); // 뮤텍스 잠금
```

```
cv.wait(lk); // 알림을 기다립니다.  
// 이것은 정확하지 않습니다!
```

```
... // 이벤트에 반응  
// (m은 잠겨 있음)
```

```
} // 닫기 크리티컬. 부분;  
// lk의 dtor를 통해 m 잠금 해제
```

```
... // 계속 반응  
// (m 이제 잠금 해제됨)
```

이 접근 방식의 첫 번째 문제는 때때로 코드 냄새라고 하는 것입니다.

코드가 작동하지만 뭔가 잘못된 것 같습니다. 이 경우 냄새가 발생합니다.
뮤텍스를 사용해야 합니다. 뮤텍스는 공유 데이터에 대한 액세스를 제어하는 데 사용되지만
탐지 및 대응 작업에는 그러한 매체가 필요하지 않을 가능성이 있습니다.
용. 예를 들어, 탐지 작업은 전역 초기화를 담당할 수 있습니다.
데이터 구조를 만든 다음 사용을 위해 반응하는 작업에 넘겨줍니다. 탐지 작업인 경우

초기화 후 데이터 구조에 액세스하지 않으며 감지 작업이 준비되었음을 나타내기 전에 반응하는 작업이 데이터 구조에 액세스하지 않으면 두 작업은 프로그램 논리를 통해 서로 방해가 되지 않습니다. 뮤텍스가 필요하지 않습니다. condvar 접근 방식이 의심스러운 디자인의 불안한 향기 뒤에 하나를 남겨야 한다는 사실.

그걸 지나쳐도 반드시 치러야 할 두 가지 문제가 더 있다.

주의:

- 감지 작업이 반응 작업이 대기하기 전에 condvar에 통지하면 반응 작업이 중단됩니다. condvar의 알림이 다른 작업을 깨우려면 다른 작업이 해당 condvar에서 대기해야 합니다. 감지 작업이 반응 작업이 대기를 실행하기 전에 알림을 실행하는 경우 반응 작업은 알림을 놓치고 영원히 기다립니다.
- wait 문은 가짜 웨이크업을 설명하지 못합니다. 스레딩 API(C++뿐 아니라 많은 언어)에서 실제로 존재하는 사실은 condvar에 통지되지 않은 경우에도 조건 변수를 기다리는 코드가 깨어날 수 있다는 것입니다. 이러한 깨우기를 가짜 깨우기라고 합니다. 적절한 코드는 기다리고 있는 조건이 실제로 발생했는지 확인하여 처리하며 깨어난 후 첫 번째 작업으로 이를 수행합니다. C++ condvar API는 대기 조건을 테스트하는 람다(또는 다른 함수 개체)가 대기로 전달되도록 허용하기 때문에 이 작업을 매우 쉽게 만들습니다. 즉, 반응 작업의 대기 호출은 다음과 같이 작성할 수 있습니다.

```
cv.wait(lk, []
    { 이벤트 발생 여부를 반환 합니다. });
```

이 기능을 활용하려면 반응하는 작업이 대기 중인 조건이 참인지 여부를 결정할 수 있어야 합니다. 그러나 우리가 고려한 시나리오에서 대기하는 조건은 감지 스레드가 인식해야 하는 이벤트의 발생입니다. 반응하는 스레드는 대기 중인 이벤트가 발생했는지 여부를 결정할 방법이 없을 수 있습니다. 이것이 조건 변수를 기다리는 이유입니다!

작업이 condvar를 사용하여 통신하도록 하는 것이 당면한 문제에 적합한 상황이 많이 있지만 이것은 그 중 하나가 아닌 것 같습니다.

많은 개발자에게 다음 트릭은 공유 부울 플래그입니다. 플래그는 처음에 false입니다. 감지 스레드가 찾고 있는 이벤트를 인식하면 플래그를 설정합니다.

```
std::atomic<bool> 플래그(거짓);           // 공유 플래그; 보다
                                                // std::atomic의 항목 40
...
                                                // 이벤트 감지
```

```
플래그 = 참; // 반응하는 작업을 알려줍니다.
```

반응하는 스레드는 단순히 플래그를 풀링합니다. 깃발이 꽂혀 있는 것을 보면,
기다리던 이벤트가 발생했음을 알고 있습니다.

```
... // 반응 준비
```

```
동안 (! 플래그); // 이벤트 대기
```

```
... // 이벤트에 반응
```

이 접근 방식은 condvar 기반 설계의 단점이 전혀 없습니다.

뮤텍스가 필요하지 않으며 감지 작업이 플래그 앞에 플래그를 설정하면 문제가 없습니다.
반응하는 작업이 풀링을 시작하고 가짜 깨우기와 유사한 것은 없습니다. 좋다 (좋아요,
좋은.

덜 좋은 것은 반응 작업에서 풀링 비용입니다. 작업이 대기하는 동안-
플래그를 설정하려면 작업이 기본적으로 차단되지만 여전히 실행 중입니다. 이와 같이,
다른 작업이 사용할 수 있는 하드웨어 스레드를 차지합니다.
컨텍스트 스위치가 타임 슬라이스를 시작하거나 완료할 때마다 발생하는 비용은 다음과 같습니다.
그렇지 않으면 전원을 절약하기 위해 종료될 수 있는 코어를 계속 실행하십시오. 진정으로
차단된 작업은 이러한 작업을 수행하지 않습니다. 대기 호출의 작업이 실제로 차단되기 때문에 이것
이 condvar 기반 접근 방식의 장점입니다.

condvar와 플래그 기반 디자인을 결합하는 것이 일반적입니다. 플래그는 여부를 나타냅니다.
관심 이벤트가 발생했지만 플래그에 대한 액세스는 뮤텍스에 의해 동기화됩니다.

뮤텍스는 플래그에 대한 동시 액세스를 방지하기 때문에 항목 40 과 같이

플래그가 std::atomic일 필요가 없다고 설명합니다. 간단한 bool이 할 것입니다. 감지-
ing 작업은 다음과 같습니다.

```
std::condition_variable 이력서; 표준// 이전과  
뮤텍스 m;
```

```
부울 플래그(거짓); // std::atomic이 아님
```

```
... // 이벤트 감지
```

```
{
```

```
std::lock_guard<std::mutex> g(m); // g의 ctor를 통해 m을 잡습니다.
```

```
플래그 = 참; // 반응하는 작업을 알려줍니다.  
// (1 부)
```

```
}
```

```
// g의 dtor를 통해 m 잠금 해제
```

```
cv.notify_one(); // 반응하는 작업을 알려줍니다 //
(파트 2)
```

다음은 반응하는 작업입니다.

```
... // 반응 준비
```

```
{ // std::unique_lock<std::mutex> lk(m); // 이전과
```

```
cv.wait(lk, [] { 반환 플래그; }); // 람다를 사용하여 // 가짜 깨우기
를 방지합니다.
```

```
... // 이벤트에 반응 // (m은
잠겨 있음)
```

```
}
```

```
... // 계속 반응 // (m 이제 잠금 해
제됨)
```

이 접근 방식은 우리가 논의한 문제를 방지합니다. 감지 작업이 알리기 전에 반응하는 작업이 대기하는지 여부에 관계없이 작동하고, 가짜 웨이크업이 있는 경우 작동하며, 폴링이 필요하지 않습니다. 그러나 탐지 작업이 매우 흥미로운 방식으로 반응 작업과 통신하기 때문에 냄새가 남아 있습니다. 조건 변수를 알리는 것은 기다리고 있던 이벤트가 아마도 발생했음을 반응 작업에 알리지만 반응 작업은 플래그를 확인하여 확실하게 해야 합니다. 플래그를 설정하면 반응 작업에 이벤트가 확실히 발생했음을 알릴 수 있지만 감지 작업은 여전히 반응 작업이 깨어 플래그를 확인하도록 조건 변수를 알려야 합니다. 접근 방식은 효과가 있지만 그다지 깨끗하지 않은 것 같습니다.

대안은 반응 작업이 감지 작업에 의해 설정된 미래를 기다리게 하여 조건 변수, 뮤텍스 및 플래그를 피하는 것입니다. 이것은 이상한 생각처럼 보일 수 있습니다. 결국 항목 38은 미래가 호출 수신자에서 (일반적으로 비동기식) 호출자에게 전달되는 통신 채널의 수신 측을 나타내며 여기에서는 감지 작업과 응답 작업 사이에 호출 수신자-호출자 관계가 없다고 설명합니다. 그러나 항목 38은 송신측이 std::promise이고 수신측이 미래인 통신 채널이 단순한 호출자-호출자 통신 이상의 용도로 사용될 수 있음을 언급합니다. 이러한 통신 채널은 프로그램의 한 위치에서 다른 위치로 정보를 전송해야 하는 모든 상황에서 사용할 수 있습니다. 이 경우 탐지 작업에서 반응 작업으로 정보를 전송하는 데 사용하고 관심 있는 이벤트가 발생했다는 정보를 전달할 것입니다.

디자인은 간단합니다. 감지 작업에는 std::promise 개체(즉, 통신 채널의 쓰기 끝)가 있고 반응 작업에는 해당 개체가 있습니다.

미래. 감지 작업이 찾고 있는 이벤트가 발생했음을 확인하면 `std::promise`를 설정합니다(즉, 통신 채널에 씁니다). 한편, 반응하는 작업은 미래를 기다립니다. 그 대기는 `std::promise`가 설정될 때까지 반응 작업을 차단합니다.

이제 `std::promise`와 `futures`(예: `std::future` 및 `std::shared_future`)는 유형 매개변수가 필요한 템플릿입니다. 해당 매개변수는 통신 채널을 통해 전송될 데이터 유형을 나타냅니다. 그러나 우리의 경우에는 전달할 데이터가 없습니다. 반응 작업에 대한 유일한 관심은 미래가 설정되었다는 것입니다. `std::promise` 및 `future` 템플릿에 필요한 것은 통신 채널을 통해 전달되는 데이터가 없음을 나타내는 유형입니다. 그 유형은 무효입니다. 따라서 탐지 작업은 `std::promise<void>`를 사용하고 반응 작업은 `std::future<void>` 또는 `std::shared_future<void>`를 사용합니다. 감지 작업은 관심 있는 이벤트가 발생할 때 `std::promise<void>`를 설정하고 반응하는 작업은 미래를 기다립니다. 반응하는 작업이 탐지 작업에서 데이터를 수신하지 않더라도 통신 채널은 탐지 작업이 `std::promise`에서 `set_value`를 호출하여 무효 데이터를 "작성"했을 때 반응 작업이 알 수 있도록 허용합니다.

그래서 주어진

```
std::약속<무효> p; // 통신 채널에 대한 //
약속
```

탐지 작업의 코드는 사소하고,

```
... // 이벤트 감지
p.set_value(); // 반응하는 작업을 알려줍니다.
```

반응하는 작업의 코드는 똑같이 간단합니다.

```
... // 반응 준비
p.get_future().wait(); // 미래를 기다립니다 // p
에 해당
...
// 이벤트에 반응
```

플래그를 사용하는 접근 방식과 마찬가지로 이 디자인은 뮤텍스가 필요하지 않고 감지 작업이 반응 작업이 대기하기 전에 `std::promise`를 설정하는지 여부에 관계없이 작동하며 가짜 깨우기에 면역입니다. (조건 변수만 해당 문제에 취약합니다.) `condvar` 기반 접근 방식과 마찬가지로 반응 작업은 대기 호출을 수행한 후 실제로 차단되므로 대기하는 동안 시스템 리소스를 소비하지 않습니다. 완벽하죠?

정확히. 물론, 미래 기반 접근 방식은 이러한 떼를 피하지만 걱정해야 할 다른 위험 요소가 있습니다. 예를 들어, 항목 38 은 std::promise와 future 사이에 공유 상태가 있고 공유 상태는 일반적으로 동적으로 할당된다고 설명합니다.

따라서 이 디자인에서 힘 기반 할당 및 할당 해제 비용이 발생한다고 가정해야 합니다.

아마도 더 중요한 것은 std::promise가 한 번만 설정될 수 있다는 것입니다. std::promise와 future 사이의 통신 채널은 일회성 메커니즘이입니다. 반복적으로 사용할 수 없습니다. 이것은 condvar 및 flag 기반 설계와 눈에 띄는 차이점이며 둘 다 여러 번 통신하는 데 사용할 수 있습니다. (condvar 는 반복적으로 알림을 받을 수 있으며 플래그는 항상 지우고 다시 설정할 수 있습니다.)

원샷 제한은 생각만큼 제한적이지 않습니다. 일시 중단된 상태의 시스템 스레드를 생성한다고 가정합니다. 즉, 스레드에서 무언가를 실행할 준비가 되었을 때 일반적인 스레드 생성 대기 시간을 피할 수 있도록 스레드 생성과 관련된 모든 오버헤드를 제거하고 싶습니다. 또는 실행을 허용하기 전에 구성할 수 있도록 일시 중단된 스레드를 만들 수 있습니다. 이러한 구성에는 우선 순위 또는 핵심 친화도 설정과 같은 항목이 포함될 수 있습니다. C++ 동시성 API는 이러한 작업을 수행할 방법을 제공하지 않지만 std::thread 자체는 native_handle 멤버 함수를 제공하며, 그 결과 플랫폼의 기본 스레딩 API(일반적으로 POSIX 스레드 또는 Windows 스레드)에 대한 액세스를 제공하기 위한 것입니다. 하위 수준 API를 사용하면 우선 순위 및 선호도와 같은 스레드 특성을 구성할 수 있습니다.

스레드를 한 번만 일시 중단하고 싶다고 가정하면(생성 후, 스레드 기능을 실행하기 전에) void future 를 사용하는 디자인이 합리적인 선택입니다. 기술의 본질은 다음과 같습니다.

```
std::약속<무효> p;
무효 반응(); // 태스크 반응을 위한 함수
무효 감지() { 표준::
스레드 t();
{ p.get_future().wait();
반
    응(); });
// 쓰레드 생성
// future가 설정될 때까지 // t를 일시 중단합니다.
...
// 여기에서 t는 // 반응을 호출하기 전에 일시 중단됩니다.
p.set_value(); // t 일시 중단 해제 (따라서 // 반응 호출)
```

```

...
// 추가 작업 수행

t.join(); } // t를 조인할 수 없도록 만듭니다.
// ( 항목 37 참조)

```

t가 감지되지 않은 모든 경로에서 결합할 수 없게 되는 것이 중요하기 때문에 Item 37의 ThreadRAII와 같은 RAI이 클래스를 사용하는 것이 바람직할 것 같습니다. 다음과 같은 코드가 떠오릅니다.

```

무효 감지() {

스레드RAII tr( // RAIi 객체 사용
    std::thread([]

        { p.get_future().wait(); react(); },
        ThreadRAII::DtorAction::join );

        // 위험하다! (아래 참조)

...
// tr 내부의 스레드 // 여기에
서 일시 중단됨

        p.set_value(); // 스레드 일시 중단 해제 // tr
                        내부
...
}

}

```

이것은 그것보다 더 안전해 보입니다. 문제는 첫 번째 "..." 영역("tr 내부의 스레드가 여기에서 일시 중단됨" 주석이 있는 영역)에서 예외가 발생하고 p에서 set_value가 호출되지 않는다는 것입니다. 즉, 람다 내부에서 wait 호출이 반환되지 않습니다. 이는 차례로 람다를 실행하는 스레드가 절대 완료되지 않는다는 것을 의미하며 RAIi 객체 tr이 tr의 소멸자에서 해당 스레드에 대한 조인을 수행하도록 구성되어 있기 때문에 문제입니다. 즉, 코드의 첫 번째 "..." 영역에서 예외가 발생하면 이 함수는 중단됩니다. tr의 소멸자는 절대 완료되지 않기 때문입니다.

이 문제를 해결할 수 있는 방법이 있지만 독자를 위해 허용된 연습의 형태로 남겨두겠습니다.5 여기에서 원본 코드(예: ThreadRAII를 사용하지 않음)를 확장할 수 있는 방법을 보여주고 싶습니다. 일시 중지했다가 하나가 아닌 일시 중지를 해제하려면

5 이 문제에 대한 조사를 시작하기에 적절한 장소는 [View From Aristeia의 2013년 12월 24일 블로그 게시물](#)입니다. "[ThreadRAII + 스레드 서스펜션 = 문제?](#)"

반응 작업, 그러나 많은 핵심은 사용하는 것이기 때문에 간단한 일반화입니다.

반응 코드에서 std::future 대신 std::shared_futures를 사용합니다. 일단 알면

std::future의 공유 멤버 함수가 공유의 소유권을 이전합니다.

공유에 의해 생성된 std::shared_future 객체에 대한 상태, 코드는 거의 다음과 같이 작성합니다.

그 자체. 유일한 미묘함은 각 반응하는 스레드가 자체 사본을 필요로 한다는 것입니다.

std::shared_future 공유 상태를 참조하므로 std::shared_future

공유에서 얻은 값은 반응하는 람다에서 실행되는 값으로 캡처됩니다.

스레드:

```
std::약속<무효> p; // 이전과

무효 감지() { // 이제 다중
    // 반응하는 작업

자동 sf = p.get_future().share(); // sf의 유형은
// std::shared_future<void>

std::vector<std::thread> vt; // 컨테이너
// 반응하는 스레드

for (int i = 0; i < threadToRun; ++i) {
    vt.emplace_back([sf]{ sf.wait(); 반응(); });
    // 로컬에서 대기
    // sf의 사본; 보다
}
// 정보를 위한 항목 42
// emplace_back에서

...
// 다음 경우 중단 감지
// 이 "..." 코드는 던졌습니다!

p.set_value(); // 모든 스레드의 일시 중단 해제

...
// 모든 쓰레드 생성
// 가입 불가; 항목 2 참조
}

(자동& t : vt) { t.join(); }
// "auto&"에 대한 정보
```

선물을 사용하는 디자인이 이러한 효과를 얻을 수 있다는 사실은 주목할 만합니다.

원샷 이벤트 커뮤니케이션을 위해 고려해야 하는 이유.

기억해야 할 사항

- 간단한 이벤트 통신의 경우 condvar 기반 설계는 불필요한 뮤텍스를 필요로 하고 작업 감지 및 반응의 상대적 진행에 제약을 가하고 이벤트가 발생 했는지 확인하기 위한 반응 작업을 요구합니다. • 플래그를 사용하는 디자인은 이러한 문제를 방지하지만 풀링을 기반으로 합니다.
- 블로킹.
- condvar와 플래그를 함께 사용할 수 있지만 결과 통신은 메커니즘이 다소 부적절합니다.
- std::promises 및 futures를 사용하면 이러한 문제를 피할 수 있지만 이 접근 방식은 공유 상태에 대해 힘 메모리를 사용하며 일회성 통신으로 제한됩니다.

항목 40: 동시성을 위해서는 std::atomic 을 사용하고 특수 메모리에는 volatile 을 사용하세요.

가난한 휘발성. 그래서 오해. 동시 프로그래밍과 아무 관련이 없기 때문에 이 장에 있어서도 안 됩니다. 그러나 다른 프로그래밍 언어(예: Java 및 C#)에서는 이러한 프로그래밍에 유용하며 C++에서도 일부 컴파일러는 동시 소프트웨어에 적용할 수 있도록 하는 의미로 volatile을 주입했습니다(그러나 해당 컴파일러로 컴파일된 경우에만).. 따라서 동시성에 대한 장에서 volatile에 대해 논의하는 것은 주변의 혼란을 없애는 것 외에 다른 이유가 없다면 가치가 있습니다.

프로그래머가 때때로 volatile과 혼동하는 C++ 기능(이 장에 확실히 속하는 기능)은 std::atomic 템플릿입니다. 이 템플릿의 인스턴스화(예: std::atomic<int>, std::atomic<bool>, std::atomic<Widget*> 등)는 다른 스레드에서 원자성으로 간주되는 작업을 제공합니다.. 일단 std::atomic 객체가 생성되면, 그 객체에 대한 연산은 마치 뮤텍스로 보호되는 임계 구역 내부에 있는 것처럼 작동하지만 연산은 일반적으로 뮤텍스를 사용했습니다.

std::atomic을 사용하는 다음 코드를 고려하십시오.

```
std::atomic<int> ai(0);           // ai를 0으로 초기화
ai = 10;                         // 원자적으로 ai를 10으로 설정
std::cout << ai;                 // ai의 값을 원자적으로 읽습니다.
++ai;                            // ai를 11로 원자적으로 증가
```

-일체 포함;

// ai를 10으로 원자적으로 감소

이러한 명령문을 실행하는 동안 ai를 읽는 다른 스레드는 0, 10 또는 11의 값만 볼 수 있습니다. 다른 값은 가능하지 않습니다(물론 이것이 ai를 수정하는 유일한 스레드라고 가정).

이 예의 두 가지 측면은 주목할 가치가 있습니다. 먼저 "std::cout << ai;" 진술에서 ai가 std::atomic이라는 사실은 ai의 읽기가 원자적이라는 것만 보장합니다. 전체 명령문이 원자적으로 진행된다는 보장은 없습니다. ai의 값을 읽고 표준 출력에 쓰기 위해 operator<<가 호출되는 시간 사이에 다른 스레드가 ai의 값을 수정했을 수 있습니다. operator<< for ints는 출력할 int에 대해 by-value 매개변수를 사용하기 때문에 명령문의 동작에는 영향을 미치지 않지만(따라서 출력된 값은 ai에서 읽은 값이 됨) 다음을 이해하는 것이 중요합니다. 그 문장에서 원자적인 것은 ai를 읽는 것에 지나지 않습니다.

예제에서 두 번째로 주목할만한 측면은 마지막 두 명령문인 ai의 증가와 감소의 동작입니다. 이들은 각각 RMW(읽기-수정-쓰기) 작업이지만 원자적으로 실행됩니다. 이것은 std::atomic 유형의 가장 좋은 특성 중 하나입니다. 일단 std::atomic 객체가 생성되면 RMW 작업을 포함하는 모든 멤버 함수가 다른 스레드에서 볼 수 있도록 보장됩니다. 원자로.

대조적으로 volatile을 사용하는 해당 코드는 다중 스레드 컨텍스트에서 거의 아무 것도 보장하지 않습니다.

휘발성 int vi(0); // vi를 0으로 초기화

vi = 10; // vi를 10으로 설정

std::cout << vi; // vi의 값 읽기

++vi; // vi를 11로 증가

--vi; // vi를 10으로 감소

이 코드를 실행하는 동안 다른 스레드가 vi의 값을 읽고 있다면 -12, 68, 4090727과 같이 무엇이든 볼 수 있습니다. 이러한 코드는 정의되지 않은 동작을 가집니다. 이러한 명령문은 vi를 수정하기 때문에 다른 스레드가 동시에 vi를 읽는 경우 std::atomic도 아니고 mutex에 의해 보호되지도 않는 메모리의 동시 판독기 및 기록기가 있습니다. 데이터 레이스.

다중 스레드 프로그램에서 `std::atomics`와 `volatiles`의 동작이 어떻게 다를 수 있는지에 대한 구체적인 예로서 여러 스레드에 의해 증가되는 각 유형의 간단한 카운터를 고려하십시오. 각각을 0으로 초기화합니다.

```
std::atomic<int> ac(0); // "원자 카운터"
```

```
휘발성 int vc(0); // "휘발성 카운터"
```

그런 다음 동시에 실행되는 두 스레드에서 각 카운터를 한 번씩 증가시킵니다.

```
/*----- 스레드 1 -----*/
```

```
++악;  
++VC;
```

```
/*----- 스레드 2 -----*/
```

```
++악;  
++VC;
```

두 스레드가 모두 완료되면 `ac`의 값(즉, `std::atomic`의 값)은 2가 되어야 합니다. 각 증가는 나눌 수 없는 작업으로 발생하기 때문입니다. 반면에 `vc`의 값은 2가 될 필요가 없습니다. 증분이 원자적으로 발생하지 않을 수 있기 때문입니다. 각 증가는 `vc`의 값 읽기, 읽은 값 증가, 결과를 다시 `vc`에 쓰는 것으로 구성됩니다. 그러나 이 세 가지 작업은 휘발성 객체에 대해 원자적으로 진행되는 것이 보장되지 않으므로 `vc`의 두 증분의 구성 요소 부분이 다음과 같이 인터리브될 수 있습니다.

1. 스레드 1은 `vc`의 값인 0을 읽습니다.
2. 스레드 2는 여전히 0인 `vc`의 값을 읽습니다.
3. 스레드 1은 읽은 0을 1로 증가시킨 다음 해당 값을 `vc`에 씁니다.
4. 스레드 2는 읽은 0을 1로 증가시킨 다음 해당 값을 `vc`에 씁니다.

따라서 `vc`의 최종 값은 두 번 증가하더라도 1입니다.

가능한 결과는 이것만이 아닙니다. `vc`의 최종 값은 일반적으로 예측할 수 없습니다. 왜냐하면 `vc`는 데이터 경쟁에 관여하고 데이터 경쟁이 정의되지 않은 동작을 유발한다는 표준 법령은 컴파일러가 말 그대로 무엇이든 할 수 있는 코드를 생성할 수 있다는 것을 의미하기 때문입니다.

물론 컴파일러는 이 여유를 악의적으로 사용하지 않습니다. 오히려 그들은 데이터 경합이 없는 프로그램에서 유효한 최적화를 수행하며 이러한 최적화는 경합이 있는 프로그램에서 예상치 못한 예측할 수 없는 동작을 생성합니다.

`RMW` 작업의 사용은 `std::atomics`가 동시성 성공 사례를 구성하고 휘발성이 실패하는 유일한 상황이 아닙니다. 한 작업이 두 번째 작업에 필요한 중요한 값을 계산한다고 가정합니다. 첫 번째 작업이 값을 계산하면 이를 두 번째 작업에 전달해야 합니다. [항목 39](#)는 첫 번째 작업이 원하는 값의 가용성을 두 번째 작업에 전달하는 한 가지 방법을 설명합니다.

`std::atomic<bool>`을 사용하여. 값을 계산하는 작업의 코드는 다음과 같습니다.

```
std::atomic<bool> valAvailable(거짓);
```

자동 `imptValue = computeImportantValue();` // 값 계산

```
valAvailable = 참; // 다른 작업에 알립니다 //
사용 가능합니다.
```

이 코드를 읽는 인간으로서 우리는 `valAvailable`에 대한 할당 전에 `imptValue`에 대한 할당이 발생하는 것이 중요하다는 것을 알고 있지만 모든 컴파일러는 독립 변수에 대한 한 쌍의 할당을 봅니다. 일반적으로 컴파일러는 관련 없는 할당을 재정렬할 수 있습니다. 즉, 이 할당 시퀀스(여기서 `a`, `b`, `x` 및 `y`는 독립 변수에 해당)가 주어지면

```
a = b;
x = y;
```

컴파일러는 일반적으로 다음과 같이 재정렬할 수 있습니다.

```
x = y;
a = b;
```

컴파일러가 재정렬하지 않더라도 기본 하드웨어가 재정렬을 수행할 수 있습니다(또는 다른 코어에 있는 것처럼 보이게 할 수 있음). 코드를 더 빠르게 실행할 수 있기 때문입니다.

그러나 `std::atomics`의 사용은 코드를 재정렬할 수 있는 방법에 대한 제한을 부과하며 그러한 제한 중 하나는 소스 코드에서 `std::atomic` 변수의 쓰기 이전에 코드가 발생할 수 없다는 것입니다(또는 이후에 다른 코어에 나타남).⁶ 이는 우리 코드에서

자동 `imptValue = computeImportantValue();` // 값 계산

```
valAvailable = 참; // 다른 작업에 알립니다 //
사용 가능합니다.
```

컴파일러는 `imptValue` 및 `valAvailable`에 대한 할당 순서를 유지해야 할 뿐만 아니라 기본 하드웨어가

6 이것은 순차 일관성을 사용하는 `std::atomics`에만 해당되며, 이는 이 책에 나와 있는 구문을 사용하는 `std::atomic` 객체에 대한 기본 일관성 모델이자 유일한 일관성 모델입니다. C++11은 또한 보다 유연한 코드 재정렬 규칙으로 일관성 모델을 지원합니다. 이러한 약한(일명 느슨한) 모델을 사용하면 일부 하드웨어 아키텍처에서 더 빠르게 실행되는 소프트웨어를 만들 수 있지만 이러한 모델을 사용하면 소프트웨어를 올바르게 만들고 이해하고 유지 관리하기가 훨씬 더 어렵습니다. 완화된 원자를 사용하는 코드의 미묘한 오류는 전문가에게도 드문 일이 아니므로 가능한 한 순차적 일관성을 유지해야 합니다.

너무합니다. 결과적으로 valAvailable을 std::atomic으로 선언하면 중요한 순서 지정 요구 사항 (모든 스레드에서 valAvailable이 변경되기 전에 imptValue를 볼 수 있어야 함)이 유지됩니다.

valAvailable을 volatile로 선언해도 동일한 코드 재정렬 제한이 적용되지 않습니다.

휘발성 부울 valAvailable(거짓);

자동 imptValue = computeImportantValue();

valAvailable = 참; // 다른 스레드가 이 할당을 볼 수 있음
// imptValue 앞에!

여기서 컴파일러는 할당 순서를 imptValue 및 valAvailable로 바꿀 수 있으며, 그렇지 않더라도 기본 하드웨어가 다른 코어의 코드에서 valAvailable 변경을 볼 수 없도록 하는 기계어 코드를 생성하지 못할 수 있습니다. imptValue 전에.

작동 원자성을 보장하지 않고 코드 재정렬에 대한 제한이 불충분한 이 두 가지 문제는 volatile이 동시에 프로그래밍에 유용하지 않은 이유를 설명하지만 유용한 이유는 설명하지 않습니다. 간단히 말해서 컴파일러에게 정상적으로 작동하지 않는 메모리를 처리하고 있음을 알리기 위한 것입니다.

"일반" 메모리는 메모리 위치에 값을 쓰면 무언가를 덮어쓸 때까지 값이 그대로 유지되는 특성이 있습니다. 그래서 내가 정상적인 int를 가지고 있다면,

정수 x;

컴파일러는 다음과 같은 일련의 작업을 봅니다.

자동 y = x;	// x 읽기 // x 다
y = x;	시 읽기

컴파일러는 y의 초기화와 종복되기 때문에 y에 대한 할당을 제거하여 생성된 코드를 최적화할 수 있습니다.

일반 메모리는 또한 메모리 위치에 값을 쓰고 절대 읽지 않은 다음 해당 메모리 위치에 다시 쓰면 첫 번째 쓰기가 사용되지 않았기 때문에 제거될 수 있다는 특성도 있습니다. 따라서 이 두 개의 인접한 진술이 주어졌을 때,

x = 10; x = 20;	// x 쓰기 // x 다 시 쓰기
--------------------	------------------------

컴파일러는 첫 번째 것을 제거할 수 있습니다. 즉, 소스 코드에 이것이 있으면

자동 y = x; y = x;	// x 읽기 // x 다 시 읽기
------------------	------------------------

```
x = 10; x =           // x 쓰기 // x 다
20;                  시 쓰기
```

컴파일러는 다음과 같이 작성된 것처럼 처리할 수 있습니다.

```
자동 y = x;           // x 읽기
x = 20;              // x 쓰기
```

누가 이런 종류의 중복 읽기 및 불필요한 쓰기(기술적으로 중복 로드 및 데드 스토어로 알려짐)를 수행하는 코드를 작성하는지 궁금하지 않게 하십시오. 대답은 인간이 직접 작성하지 않는다는 것입니다. 적어도 우리는 그렇게 하기를 바랍니다. 그러나 컴파일러가 합리적으로 보이는 소스 코드를 취하고 템플릿 인스턴스화, 인라인 및 다양한 일반적인 재정렬 최적화를 수행한 후에는 컴파일러가 제거 할 수 있는 중복 로드와 데드 스토어(dead store)가 결과에 나타나는 것이 드문 일이 아닙니다.

이러한 최적화는 메모리가 정상적으로 작동하는 경우에만 유효합니다. "특별한" 메모리는 그렇지 않습니다. 아마도 가장 일반적인 종류의 특수 메모리는 메모리 매핑된 I/O에 사용되는 메모리일 것입니다. 이러한 메모리의 위치는 일반 메모리(예: RAM)를 읽거나 쓰는 것이 아니라 실제로 주변 장치(예: 외부 센서 또는 디스플레이, 프린터, 네트워크 포트 등)와 통신합니다. 이러한 맥락에서 겉보기에 중복된 읽기가 있는 코드를 다시 고려하십시오.

```
자동 y = x;           // x 읽기 // x 다
y = x;                시 읽기
```

x가 온도 센서에 의해 보고된 값에 해당하는 경우 x의 두 번째 판독은 중복되지 않습니다. 왜냐하면 온도가 첫 번째 판독과 두 번째 판독 사이에 변경되었을 수 있기 때문입니다.

겉보기에 불필요한 쓰기에 대한 비슷한 상황입니다. 예를 들어 이 코드에서

```
x = 10; x =           // x 쓰기 // x 다
20;                  시 쓰기
```

x가 무선 송신기의 제어 포트에 해당하면 코드가 무선에 명령을 내리고 값 10이 값 20과 다른 명령에 해당할 수 있습니다. 첫 번째 할당을 최적화하면 명령 시퀀스가 변경됩니다. 라디오로 보냅니다.

`volatile`은 컴파일러에게 우리가 특별한 메모리를 다루고 있다는 것을 알려주는 방식입니다. 컴파일러에 대한 의미는 "이 메모리에서 작업에 대한 최적화를 수행하지 마십시오."입니다. 따라서 x가 특수 메모리에 해당하면 휘발성으로 선언됩니다.

휘발성 int x;

원래 코드 시퀀스에 미치는 영향을 고려하십시오.

자동 `y = x;` // x 읽기
`y = x;` // x를 다시 읽습니다 (최적화할 수 없음).

`x = 10; x =` // x 쓰기 (최적화할 수 없음)
`20;` // x를 다시 씁니다.

x가 메모리 매핑된 경우(또는
프로세스 간에 공유되는 메모리 위치 등).

퀴즈! 마지막 코드에서 y의 유형은 무엇입니까: int 또는 volatile int? 7

겉보기에 중복된 로드와 데드 스토어가 다음과 같은 경우에 보존되어야 한다는 사실
그런데 특수 메모리를 다루는 것은 std::atomics가 부적합한 이유를 설명합니다.
이런 종류의 작업을 위해. 컴파일러는 이러한 중복 연산을 제거할 수 있습니다.
std::atomics에 대한 설명입니다. 코드는 Volatile 와 완전히 같은 방식으로 작성되지 않습니다.
타일이지만 잠시 간과하고 컴파일러가 무엇인지에 집중하면
허용된 경우, 개념적으로 컴파일러가 이것을 취할 수 있다고 말할 수 있습니다.

`std::atomic<int> x;`

자동 `y = x; y = x;` // 개념적 으로 x를 읽습니다(아래 참조).
// 개념적 으로 x를 다시 읽습니다(아래 참조).

`x = 10; x =` // x 쓰기
`20;` // x를 다시 씁니다.

다음과 같이 최적화하십시오.

자동 `y = x; x = 20;` // 개념적 으로 x를 읽습니다(아래 참조).
// x 쓰기

특수 메모리의 경우 이것은 분명히 용납할 수 없는 동작입니다.

이제 x가 다음과 같을 때 이 두 명령문 중 어느 것도 컴파일되지 않습니다.
표준::원자:

자동 `y = x;` // 오류!
`y = x;` // 오류!

std::atomic에 대한 복사 작업이 삭제되기 때문입니다(항목 11 참조). 그리고
좋은 이유가 있습니다. x com으로 y를 초기화하면 어떤 일이 일어날지 생각해 보십시오.

7 y의 유형은 자동 추론되므로 항목 2에 설명된 규칙을 사용합니다. 이러한 규칙은
비참조 비포인터 유형(y의 경우), const 및 volatile 한정자는 삭제됩니다. 애
따라서 type은 단순히 int입니다. 이는 y에 대한 중복 읽기 및 쓰기를 제거할 수 있음을 의미합니다. 에서
예를 들어 컴파일러는 x가 휘발성이기 때문에 초기화와 y에 대한 할당을 모두 수행해야 합니다.
따라서 x의 두 번째 읽기는 첫 번째 값과 다른 값을 산출할 수 있습니다.

쌓여있다. x는 std::atomic이므로 y의 유형도 std::atomic으로 추론됩니다(항목 2 참조). 나는 앞서 std::atomics의 가장 좋은 점 중 하나가 모든 연산이 원자적이라는 점이라고 언급했지만 x에서 y의 복사 생성이 원자적이 되기 위해서는 컴파일러가 x를 읽고 y를 쓰는 코드를 생성해야 합니다. 단일 원자 연산. 하드웨어는 일반적으로 그렇게 할 수 없으므로 std::atomic 유형에 대해서는 복사 생성이 지원되지 않습니다. 복사 할당은 같은 이유로 삭제되며, 이것이 x에서 y로의 할당이 컴파일되지 않는 이유입니다. (이동 작업은 std::atomic에서 명시적으로 선언되지 않으므로 항목 17에 설명된 컴파일러 생성 특수 함수에 대한 규칙에 따라 std::atomic은 이동 구성을 이동 할당을 제공하지 않습니다.)

x의 값을 y로 가져오는 것이 가능하지만 std::atomic의 멤버 함수 로드 및 저장을 사용해야 합니다. load 멤버 함수는 std::atomic의 값을 읽습니다.

원자적으로, store 멤버 함수는 그것을 원자적으로 씁니다. y를 x로 초기화한 다음 x의 값을 y에 넣으려면 코드를 다음과 같이 작성해야 합니다.

```
표준::원자<int> y(x.load());           // x 읽기
y.store(x.load());                      // x를 다시 읽습니다.
```

이것은 컴파일되지만 x를 읽는 것이 (x.load()를 통해) y를 초기화하거나 저장하는 것과는 별개의 함수 호출이라는 사실은 두 명령문이 전체적으로 단일 원자 연산으로 실행될 것으로 예상할 이유가 없음을 분명히 합니다.

해당 코드가 주어지면 컴파일러는 x 값을 두 번 읽는 대신 레지스터에 저장하여 이를 "최적화"할 수 있습니다.

```
레지스터 = x.load();                  // x를 레지스터로 읽어들임
std::atomic<int> y(레지스터);        // 레지스터 값으로 y 초기화
y.store(등록);                       // 레지스터 값을 y에 저장
```

결과는 보시다시피 x에서 한 번만 읽습니다. 특수 메모리를 다룰 때 피해야 하는 최적화 유형입니다. (휘발성 변수에 대해서는 최적화가 하용되지 않습니다.)

따라서 상황은 다음과 같이 명확해야 합니다.

- std::atomic은 동시 프로그래밍에 유용하지만 특정 항목에 액세스하는 데에는 유용하지 않습니다. 사회적 기억.
- volatile은 특수 메모리에 액세스하는 데 유용하지만 동시 처리에는 유용하지 않습니다. 문법.

`std::atomic`과 `volatile`은 다른 용도로 사용되기 때문에 함께 사용할 수도 있습니다.

```
휘발성 std::atomic<int> vai; // vai에 대한 작업은 // 원자적이며 //
                                // 최적화할 수 없습니다.
```

이는 `vai`가 여러 스레드에서 동시에 액세스한 메모리 매핑된 I/O 위치에 해당하는 경우 유용할 수 있습니다.

마지막으로 일부 개발자는 필요하지 않은 경우에도 `std::atomic`의 멤버 로드 및 저장 함수를 사용하는 것을 선호합니다. 소스 코드에서 관련된 변수가 "정상"이 아님을 명시하기 때문입니다. 그 사실을 강조하는 것은 무리가 아닙니다. `std::atomic`에 액세스하는 것은 일반적으로 비 `std::atomic`에 액세스하는 것보다 훨씬 느립니다. 우리는 이미 `std::atomics`를 사용하면 컴파일러가 그렇지 않으면 허용되는 특정 종류의 코드 재정렬을 수행하는 것을 방지하는 것을 보았습니다. 따라서 `std::atomics`의 로드 및 저장을 호출하면 잠재적인 확장성 문제를 식별하는 데 도움이 될 수 있습니다. 정확성 관점에서 다른 스레드와 정보를 전달하기 위한 변수에 대한 저장 호출(예: 데이터 가용성을 나타내는 플래그)을 보지 못한다는 것은 변수가 `std::atomic`으로 선언되어야 할 때 선언되지 않았음을 의미할 수 있습니다. .

그러나 이것은 대체로 스타일 문제이며 `std::atomic`과 `volatile` 사이에서 선택하는 것과는 상당히 다릅니다.

기억해야 할 사항

- `std::atomic`은 뮤텍스를 사용하지 않고 여러 스레드에서 액세스하는 데이터용입니다. 동시에 소프트웨어를 작성하기 위한 도구입니다.
- `volatile`은 읽기 및 쓰기가 최적화되어서는 안 되는 메모리용입니다. 특수기억으로 작업하기 위한 도구입니다.