

들어가기에 앞서

항상 그렇게 명확지는 않았지만, 러스트 프로그래밍 언어는 근본적으로 권한 분산에 관한 것입니다: 여러분이 어떠한 종류의 코드를 작성하는 중이던 간에, 러스트는 여러분에게 더 멀리 뻗어갈 권한을 주어, 다양한 분야에서 여러분이 전에 했던 것보다 자신감을 가지고 프로그래밍 할 수 있도록 해줍니다.

예를 들어, 메모리 관리, 데이터 표현, 그리고 동시성에 대한 저수준의 디테일을 다루는 “시스템 레벨”的 일을 해보세요. 전통적으로, 이 프로그래밍 영역은 신비로운 것으로 보이고, 이 영역의 악명높은 함정에 빠지지 않기 위해 필요한 수 년의 시간을 배우는데 혼신한 몇몇의 선택받은 자들만이 접근할 수 있는 것으로 여겨졌습니다. 그리고 이 분야를 연마해온 그들조차도 그들의 코드가 이용당하거나, 망가지거나, 붕괴되지 않도록 조심스럽게 작업을 합니다.

러스트는 여러분이 길을 잊지 않도록 하기 위해, 오래된 함정들을 제거하고 친근하면서도 세련된 도구 세트를 제공함으로써 이 장벽들을 부릅니다. 저수준의 제어에 “살짝만 발을 담글” 필요가 있는 프로그래머들은, 변덕스러운 툴체인의 미세한 지점들을 학습할 필요없이 러스트를 통해 그렇게 할 수 있습니다. 그 정도가 아니라, 이 언어는 속도와 메모리 사용 측면에서 효율적이면서도 안정적인 코드를 작성해 나갈 수 있도록 여러분들을 자연스럽게 안내하도록 설계되었습니다.

이미 저수준의 코드를 가지고 일하고 있는 프로그래머들은 러스트를 사용하여 그들의 야망을 더 키울 수 있습니다. 예를 들면, 러스트에서 소개하는 병렬성은 상대적으로 저위험성 연산입니다: 컴파일러가 여러분을 위해 고전적인 실수를 잡아줄 것입니다. 그리고 여러분은 뜻하지 않게 프로그램이 망가지거나 악용되지 않으리라는 자신감을 가지고 여러분의 코드에 대하여 더 공격적인 최적화에 몰두할 수 있습니다.

하지만 러스트는 저수준의 시스템 프로그래밍에 한정되지 않습니다. 이 언어는 CLI 앱, 웹 서버, 그리고 작성하기에 꽤나 즐거운 종류의 다른 코드들을 만들기에 충분할 정도로 표현력이 풍부하고 인간 공학적입니다 - 여러분은 이 책의 뒷 부분에서 이에 대한 단순한 예제들을 보게될 것입니다. 러스트로 일하는 것은 여러분이 어떤 영역에서 또다른 영역으로 옮기는 기술을 만들수 있게 해줍니다; 여러분은 웹 앱을 작성하는 것으로 러스트를 배울 수 있고, 그 다음 여러분의 라즈베리 파이를 대상으로 동일한 기술을 적용할 수 있습니다.

이 책은 러스트 사용자에게 권한을 주기 위해 러스트의 잠재력을 모두 담아내었습니다. 이 책은 러스트에 대한 여러분의 지식을 향상시키는 것 뿐만 아니라, 일반적인 프로그래머로서의 도약과 자신감을 향상시키는 것을 돋기 위한 의도로 친절하고 이해하기 쉬운 텍스트로 되어있습니다. 그러니 뛰어 들어서 배울 준비를 하세요-러스트 커뮤니티에 오신 것을 환영합니다!

- Nicholas Matsakis와 Aaron Turon

소개

러스트 프로그래밍 언어, 러스트 입문서에 오신 것을 환영합니다.

러스트 프로그래밍 언어는 여러분이 더 빠르고, 더 안정적인 소프트웨어를 작성하도록 해줍니다. 프로그래밍 언어 디자인에서 고수준의 인간공학과 저수준의 제어는 종종 조화롭지 못합니다; 러스트는 이러한 갈등에도 전합니다. 강력한 기술적 능력과 훌륭한 개발자 경험을 조화롭게 하는 것을 통해, 러스트는 (메모리 사용 같은) 저수준 디테일을 그러한 제어를 하는데 동반되는 전통적으로 귀찮은 것들 없이도 제어하는 옵션을 제공합니다.

러스트는 누구를 위한 것인가요?

러스트는 다양한 이유로 수많은 사람들에게 이상적입니다. 가장 중요한 그룹 중 일부를 살펴봅시다.

개발자 팀

러스트는 시스템 프로그래밍 지식에 대한 다양한 수준을 가진 큰 개발자 팀들 사이에서 협업을 하기 위한 생산적인 도구라는 것이 밝혀지고 있습니다. 저수준 코드는 다양한 감지하기 힘든 버그들에 노출되기 쉬운데, 이는 다른 대부분의 언어들에서는 경험 있는 개발자들에 의한 대규모의 테스트 및 세심한 코드 리뷰를 통해 잡을 수 있습니다. 러스트에서는, 컴파일러가 동시성 버그를 포함하여 이러한 찾기 어려운 버그를 가진 코드의 컴파일을 거부함으로써 문지기 역할을 수행합니다. 이 컴파일러와 나란히 작업을 함으로써, 팀은 버그를 추적하는 것보다는 프로그램의 로직에 집중하는데 더 많은 시간을 쓸 수 있습니다.

또한 러스트는 시스템 프로그램 세계로 현대적인 개발자 도구들을 가져옵니다:

- Cargo라고 불리는 기본 구성에 포함된 의존성(dependency) 관리자 및 빌드 도구는, 러스트 생태계 상에서 고통 없고 일관되게 의존성을 추가하고, 컴파일하고, 관리하도록 해줍니다.
- Rustfmt는 개발자들 사이에서 일관된 코딩 스타일을 반드시 따르도록 해줍니다.
- 러스트 언어 서버(Rust Language Server)는 코드 자동완성(code completion) 및 인라인 에러 메시지를 위한 통합 개발 환경(IDE)으로의 결합에 힘을 제공합니다.

이들 및 러스트 생태계의 다른 툴들을 이용함으로서, 개발자들은 시스템 수준의 코드를 작성하면서도 생산적일 수 있습니다.

학생

러스트는 학생들 및 시스템 개념에 대하여 공부하는데 관심이 있는 이들을 위한 것입니다. 러스트를 사용하여, 많은 사람들이 운영 체제 개발과 같은 주제에 대해 공부해왔습니다. 커뮤니티는 매우 따뜻하고 기쁘게 학생들을 질문에 대하여 대답해줍니다. 이 책과 같은 노력들을 통해서, 러스트 팀은 더 많은 사람들, 특히 프로그래밍에 새로 입문한 사람들이 시스템 개념에 더 접근하기 쉬워지길 원합니다.

회사

크고 작은 수백 개의 회사들이 다양한 작업들을 위해 프로덕션에 러스트를 사용합니다. 그 작업들에는 커맨드 라인 도구, 웹 서비스, 데브옵스(DevOps) 도구화, 임베디드 장치, 오디오 및 비디오 분석과 트랜스코딩, 암호화폐, 생물정보학, 검색 엔진, IOT(internet of things) 애플리케이션, 머신 러닝, 그리고 심지어는 파이어폭스 웹브라우저의 주요 부분들을 포함합니다.

오픈 소스 개발자

러스트는 러스트 프로그래밍 언어, 커뮤니티, 개발자 도구, 그리고 라이브러리를 만들기를 원하는 사람들을 위한 것입니다. 우리는 여러분이 러스트 언어에 기여하는 것을 정말 원합니다.

속도와 안정성을 소중하게 생각하는 사람

러스트는 언어에서 속도와 안정성을 간절히 기원하는 사람들을 위한 것입니다. 여기서 속도란, 여러분이 러스트를 가지고 만들 수 있는 프로그램의 속도와 러스트가 여러분들로 하여금 이를 작성하게 하는 속도를 의미하는 것입니다. 러스트 컴파일러의 검사들은, 이런 검사들을 가지고 있지 않은 언어라서 개발자들이 고치기를 꺼려하는 불안정한 레거시 코드들과는 반대로 기능 추가 및 리팩토링을 통해 안정성을 보장해줍니다. 비용 없는 추상화, 더 낮은 수준의 코드를 수동으로 작성한 코드만큼 빠르게 컴파일해주는 더 높은 수준의 기능을 위해 고군분투함으로서, 러스트는 안정적인 코드가 또한 빠른 코드가 되도록 노력합니다.

비록 모든 이들이 러스트 언어가 지원하기를 바라는 완벽한 리스트를 우리가 제공하지는 않을지라도, 우리가 언급해온 이들은 가장 큰 이해당사자들의 일부입니다. 종합적으로, 러스트의 가장 큰 야망은 프로그래머들이 수십 년간 받아들여 온 트레이드오프의 이분법을 제거하는 것입니다: 안정성과 생산성, 속도와 인간공학을 말이지요. 러스트에게 기회를 주고, 이 선택이 여러분에게도 작동하는지 알아보세요.

이 책은 누구를 위한 것인가요?

이 책은 여러분이 다른 프로그래밍 언어로 코드를 작성해 본적은 있다고 가정하지만, 그게 언어인지에 대해서는 어떠한 가정도 하지 않습니다. 우리는 이 교재가 다양한 종류의 프로그래밍 배경으로부터 온 이들에게 널

리 접근될 수 있도록 시도해 왔습니다. 우리는 무엇이 프로그래밍인지, 혹은 프로그래밍에 대해 어떻게 생각해야 하는지에 대하여 많은 시간을 쓰지 않습니다. 만일 여러분이 프로그래밍에 대해 완전히 초보라면, 특별히 프로그래밍에 대한 소개를 제공하는 책을 읽는 것이 더 좋을 것입니다.

이 책을 이용하는 방법

일반적으로, 이 책은 여러분이 앞에서부터 뒤로 순차적으로 읽고 있음을 가정합니다. 뒤편의 장들은 그 이전의 장들의 개념 위에서 만들어지고, 그 이전의 장들은 어떤 주제에 대해 더 깊이 탐구하지 않을 수도 있습니다; 우리는 보통 이후의 장에서 그 주제에 대해 다시 이야기 합니다.

여러분은 이 책에서 두 종류의 장들을 발견할 것입니다: 개념 장과 프로젝트 장입니다. 개념 장에서는 러스트의 관점에 대해 배울 것입니다. 프로젝트 장에서는 여러분이 여태껏 배운 것을 적용하여, 함께 작은 프로그램을 만들어볼 것입니다. 2, 12, 20장은 프로젝트 장입니다; 나머지는 개념 장입니다.

추가적으로, 2장은 러스트 언어에 대한 직접 해 보는 소개입니다. 우리는 개념들을 높은 수준에서 다루고, 이후 장들에서는 추가적인 디테일을 제공할 것입니다. 만일 여러분이 바로 손에 흙을 묻히고 싶다면, 2장은 그런 이들을 위한 장입니다. 여러분은 심지어 처음부터 다른 프로그래밍 언어 특성과 유사한 러스트 특성을 다루는 3장을 건너뛰고, 러스트의 소유권 시스템을 배우는 4장으로 진행하고 싶어 할지도 모릅니다. 하지만, 여러분이 만약 다음으로 넘어가기 전에 모든 디테일을 공부하기를 선호하는 특별히 꼼꼼한 학습자라면, 여러분은 2장을 건너뛰어 3장으로 곧바로 간 다음, 학습한 디테일들을 프로젝트에 적용해보기 위해 2장으로 돌아오는 것을 원할 수도 있습니다.

5장은 구조체와 메소드를, 6장은 열거형과 `match` 표현식, 그리고 `if let` 흐름 제어문을 다룹니다. 여러분들은 러스트 내에서 커스텀 타입을 만들기 위해 구조체와 열거형을 이용할 것입니다.

7장에서는 여러분의 코드와 공개적인 API(Application Programming Interface)를 조직화하기 위한 러스트의 모듈 시스템 및 접근 권한 규칙에 대해 배울 것입니다. 8장에서는 벡터, 스트링, 해쉬맵과 같은 표준 라이브러리에서 제공하는 일반적인 컬렉션 데이터 구조를 다룹니다. 9장에서는 러스트의 에러 처리 철학과 기술에 대해 탐구합니다.

10장에서는 제네릭, 트레이트, 그리고 라이프타임에 대해 깊이 파보는데, 이는 여러분에게 여러 개의 타입에 대하여 적용되는 코드를 정의하는 힘을 줍니다. 11장은 테스트에 관한 모든 것을 다루는데, 이는 러스트의 안정성 보장에도 불구하고 여러분의 프로그램 로직이 옳음을 확실히 하기 위해 여전히 필요합니다. 12장에서, 우리는 파일 내에서 텍스트를 검색하는 `grep` 커맨드 라인 도구가 제공하는 기능의 일부를 직접 구현해 볼 것입니다. 이를 위하여, 우리는 이전 장에서 다루었던 수많은 개념들을 이용할 것입니다.

13장에서는 클로저와 반복자에 대해 탐구합니다: 함수형 프로그래밍 언어에서부터 온 러스트의 특성입니다. 14장에서는 Cargo를 더 깊이 조사하고 여러분의 라이브러리를 다른 사람들과 공유하는 최고의 관례들에 대해 이야기하겠습니다. 15장에서는 표준 라이브러리가 제공하는 스마트 포인터와 이 기능을 가능케 하는 트레

잇에 대해 다룹니다.

16장에서는 동시성 프로그래밍의 서로 다른 모델들을 알아보고 러스트가 어떤 식으로 다수의 쓰레드를 겁 없이 프로그래밍할 수 있도록 해주는지 이야기 하겠습니다. 17장에서는 아마도 여러분이 친숙할 수 있는 객체 지향 프로그래밍 원칙과 러스트의 표현 양식이 어떤 차이가 있는지 보겠습니다.

18장은 패턴과 패턴 매칭에 대한 참고자료인데, 이 패턴 및 패턴 매칭은 러스트 프로그램 전체를 통틀어 아이디어를 표현하는 강력한 방식입니다. 19장은 다양한 고급 주제를 뷔페처럼 담고 있는데, 이를테면 unsafe 러스트와 라이프타임, 트레이트, 타입, 함수, 그리고 클로저에 대한 추가적인 주제를 포함하고 있습니다.

20장에서는 저수준 멀티쓰레드 웹서버를 구현하는 것으로 프로젝트를 완성할 것입니다!

마지막으로, 몇 개의 부록들은 언어에 대한 유용한 정보들을 참고자료 같은 형식으로 담고 있습니다. 부록 A는 러스트의 키워드를 다룹니다. 부록 B는 러스트의 연산자와 심볼을 다룹니다. 부록 C는 표준 라이브러리가 제공하는 추론 가능한 (derivable) 트레이트들을 다룹니다. 부록 D는 매크로를 다룹니다.

이 책을 읽는 잘못된 방식이란 없습니다: 만일 여러분이 건너뛰기를 원한다면, 그렇게 하세요! 만일 여러분이 어떠한 혼란이라도 경험한다면 다시 이전 장들로 돌아와야 할지도 모릅니다. 하지만 어떻게 하든 여러분 둎입니다.

러스트를 배우는 과정의 중요한 부분은 컴파일러가 표시해주는 에러 메시지를 어떻게 읽는지를 배우는 것입니다: 이는 여러분들을 작동하는 코드로 향해 안내해줄 것입니다. 그렇기 때문에, 우리는 컴파일이 되지 않은 다양한 예제 코드와 함께 그러한 상황에서 컴파일러가 여러분에게 보여줄 에러 메시지를 제공할 것입니다. 만일 여러분이 입문하여 임의의 예제를 실행한다면, 그게 컴파일 안 될 수도 있음을 알아두세요! 여러분이 실행하기를 시도하는 그 예제가 에러를 의도한 것인지를 알아보기 위해서 그 주변의 텍스트를 읽어주세요. 대부분의 경우, 우리는 컴파일 되지 않는 어떤 코드의 올바른 버전으로 여러분을 이끌어갈 것입니다.

소스 코드

이 책을 제작하도록 하는 소스코드는 [GitHub](#)에서 찾을 수 있습니다.

시작하기

여러분의 러스트 여정을 시작해봅시다! 이 장에서는 다음을 다룰 것입니다:

- Linux, macOS, Windows에 러스트 설치하기
- "Hello, world!"를 출력하는 프로그램 작성하기
- 러스트의 패키지 매니저이자 빌드 시스템인 `cargo` 사용하기

설치하기

첫 번째 단계는 러스트를 설치하는 것입니다. 우리는 `rustup`이라고 하는 러스트 버전 및 관련 도구들을 관리하기 위한 커맨드 라인 도구를 통하여 러스트를 다운로드할 것입니다. 다운로드를 위해서는 인터넷 연결이 필요할 것입니다.

다음 단계들이 러스트 컴파일러의 최신 안정 버전을 설치합니다. 이 책에 나오는 모든 예제들과 출력들은 안정화된 러스트 1.21.0을 사용했습니다. 러스트의 안정성에 대한 보장은 책에 나오는 모든 예제들이 새로운 러스트 버전에서도 계속해서 잘 컴파일 되도록 해줍니다. 버전마다 출력이 약간씩 다를 수도 있는데, 이는 러스트가 종종 에러 메시지와 경고들을 개선하기 때문입니다. 바꿔 말하면, 이 단계들을 이용하여 여러분이 설치한 러스트가 어떤 새로운 안정화 버전이라도 이 책의 내용에 기대하는 수준으로 동작해야 합니다.

커맨드 라인 표기법

이 장 및 책 곳곳에서, 우리는 터미널에서 사용되는 몇몇 커맨드를 보여줄 것입니다. 여러분이 터미널에 입력해야 하는 라인들은 모두 `$`로 시작합니다. 여러분은 `$` 문자를 입력할 필요가 없습니다; 이는 각 커맨드의 시작을 나타냅니다. 여러분이 일반 사용자로서 실행할 커맨드를 위해 `$`를 그리고 여러분이 관리자로서 실행할 커맨드를 위해 `#`를 쓰는 관례는 많은 튜토리얼들이 사용합니다. `$`로 시작하지 않는 라인들은 보통 이전 커맨드의 출력을 나타냅니다. 추가적으로, 파워쉘 한정 예제는 `$` 대신 `>`를 이용할 것입니다.

Linux와 macOS에서 Rustup 설치하기

만일 여러분들이 Linux 혹은 macOS를 사용중이라면, 터미널을 열고 다음 커맨드를 입력하세요:

```
$ curl https://sh.rustup.rs -sSf | sh
```

이 커맨드는 스크립트를 다운로드하고 `rustup` 도구의 설치를 시작하는데, 이 도구는 가장 최신의 러스트 안정화 버전을 설치해줍니다. 여러분의 패스워드를 입력하라는 프롬프트가 나올 수도 있습니다. 설치가 성공적이면, 다음과 같은 라인이 나타날 것입니다:

```
Rust is installed now. Great!
```

물론 여러분이 어떤 소프트웨어를 설치하기 위해 `curl URL | sh`를 사용하는 것을 신용하지 않는다면, 여러분이 원하는 어떤 방식으로든 이 스크립트를 다운로드하고, 검사하고, 실행할 수 있습니다.

설치 스크립트는 여러분의 다음 로그인 이후에 러스트를 자동적으로 여러분의 시스템 패스에 추가합니다. 만일 여러분이 터미널을 재시작하지 않고 러스트를 바로 사용하기를 원한다면, 다음과 같은 커멘트를 쉘에서 실행하여 수동적으로 러스트를 시스템 패스에 추가하세요:

```
$ source $HOME/.cargo/env
```

혹은 그 대신에, 여러분의 *~/.bash_profile*에 다음과 같은 라인을 추가할 수 있습니다:

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

추가적으로, 여러분은 어떤 종류의 링커가 필요할 것입니다. 이미 설치되어 있을 것 같지만, 여러분이 러스트 프로그램을 컴파일하다가 링커를 실행할 수 없음을 나타내는 에러를 보게 되면, 링커를 설치해야 합니다. 여러분은 C 컴파일러를 설치할 수 있는데, 이것이 보통 올바른 링커와 함께 설치되기 때문입니다. C 컴파일러를 인스톨하는 방법을 위해서는 여러분의 플랫폼 문서를 확인하세요. 몇몇의 일반적인 러스트 패키지는 C 코드에 의존적이고 C 컴파일러 또한 사용할 것이므로, 지금 상황에 상관없이 하나 설치하는것이 좋을 수도 있습니다.

Windows에서 Rustup 설치하기

Windows에서는 <https://www.rust-lang.org/en-US/install.html> 페이지로 가서 러스트 설치를 위한 지시를 따르세요. 설치의 몇몇 지점에서, 여러분이 Visual Studio 2013이나 이후 버전용 C++ 빌드 도구 또한 설치할 필요가 있음을 설명하는 메세지를 받을 것입니다. 이 빌드 도구를 얻는 가장 쉬운 방법은 [Visual Studio 2017용 빌드 도구](#)를 설치하는 것입니다. 이 도구들은 다른 도구 및 프레임워크 섹션 내에 있습니다.

이 책의 나머지 부분에서는 *cmd.exe* 및 파워쉘 모두에서 동작하는 커멘드를 사용합니다. 만일 특별히 다른 부분이 있다면, 어떤 것을 이용하는지 설명할 것입니다.

Rustup 없이 커스텀 설치하기

만일 여러분이 어떤 이유로 **rustup**을 쓰지 않기를 선호한다면, [the Rust installation page](#) 페이지에서 다른 옵션을 확인하세요.

업데이트 및 설치 제거하기

rustup을 통해 러스트를 설치한 뒤라면, 최신 버전을 업데이트하는 것은 쉽습니다. 여러분의 쉘에서 다음과 같은 업데이트 스크립트를 실행하세요:

```
$ rustup update
```

러스트와 `rustup`을 제거하려면 다음과 같은 설치 제거용 스크립트를 쉘에서 실행하세요:

```
$ rustup self uninstall
```

문제 해결하기

러스트가 올바르게 설치되었는지를 확인하기 위해서는, 쉘을 열고 다음 라인을 입력하세요:

```
$ rustc --version
```

버전 번호, 커밋 해쉬, 그리고 배포된 최신 안정 버전에 대한 커밋 일자가 다음과 같은 형식으로 보여야 합니다:

```
rustc x.y.z (abcabca... yyyy-mm-dd)
```

이 정보가 보인다면, 여러분은 러스트를 성공적으로 설치한 것입니다! 만일 이 정보가 보이지 않고 Windows를 이용중이라면, `%PATH%` 시스템 변수 내에 러스트가 있는지 확인해주세요. 만일 이 설정이 모두 정확하고 러스트가 여전히 동작하지 않는다면, 여러분이 도움을 구할 수 있는 몇 군데의 장소가 있습니다. 가장 쉬운 방법은 [#rust IRC 채널](irc.mozilla.org)인데, 이는 [Mibbit](#)을 통해 접속할 수 있습니다. 이 주소에서 여러분을 도와줄 수 있는 다른 러스티시안(Rustacean, 우리가 스스로를 부르는 우스운 별명입니다)들과 채팅을 할 수 있습니다. 다른 훌륭한 리소스들에는 [유저 포럼](#)과 [Stack Overflow](#)가 있습니다.

로컬 문서

인스톨러에는 또한 문서 복사본이 로컬에 포함되어 있으므로, 여러분은 이를 오프라인으로 읽을 수 있습니다. 여러분의 브라우저에서 로컬 문서를 열려면 `rustup doc`을 실행하세요.

표준 라이브러리가 제공하는 타입이나 함수가 무엇을 하는지 혹은 어떻게 사용하는지 확신이 들지 않는다면 언제라도 API (application programming interface) 문서를 이용하여 알아보세요!

Hello, World!

여러분이 러스트를 설치했으니, 이제 여러분의 첫번째 러스트 프로그램을 작성해봅시다. 새로운 언어를 배울 때면 “Hello, world!”라는 텍스트를 스크린에 출력하는 짧은 프로그램을 작성하는 것이 전통이니, 우리도 여기서 그렇게 할 것입니다!

노트: 이 책은 커맨드 라인에 대한 기본적인 친숙성을 가정하고 있습니다. 러스트는 여러분의 코드 수 정, 도구 사용, 혹은 어디에 여러분의 코드가 있는지에 대한 어떠한 특별 요구도 없으므로, 커맨드 라인 대신 IDE (Integrated Development Environment, 통합 개발 환경)를 이용하는 것은 선호한다면, 여러분이 좋아하는 IDE를 편히 이용하세요. 이제 많은 IDE들이 어느 정도 수준의 러스트 지원을 해줍니다; 자세한 사항은 해당 IDE의 문서를 확인하세요. 최근에는 러스트 팀이 훌륭한 IDE 지원을 활성화하는데 집중해왔으며, 매우 급격한 진전이 이루어지고 있습니다!

프로젝트 디렉토리 만들기

여러분의 러스트 코드를 저장하기 위한 디렉토리를 만드는 것으로 시작할 것입니다. 여러분의 코드가 어디에 있는지는 러스트에게 문제가 되지 않습니다만, 이 책의 예제 및 프로젝트들을 위해서, 우리는 여러분의 홈 디렉토리에 *projects* 디렉토리를 만들고 모든 프로젝트를 그곳에 유지하는 것을 제안합니다.

터미널을 열고 다음 커맨드를 입력하여 *projects* 디렉토리를 만들고 *projects* 디렉토리 내에 “Hello, world!” 프로젝트를 위한 디렉토리를 만드세요.

Linux와 macOS에서는 다음을 입력하세요:

```
$ mkdir ~/projects  
$ cd ~/projects  
$ mkdir hello_world  
$ cd hello_world
```

Windows CMD에서는 다음을 입력하세요:

```
> mkdir "%USERPROFILE%\projects"  
> cd /d "%USERPROFILE%\projects"  
> mkdir hello_world  
> cd hello_world
```

Windows 파워쉘에서는 다음을 입력하세요:

```
> mkdir $env:USERPROFILE\projects
> cd $env:USERPROFILE\projects
> mkdir hello_world
> cd hello_world
```

러스트 프로그램을 작성하고 실행하기

다음으로, *main.rs*이라 불리우는 새로운 소스 파일을 만드세요. 러스트 파일들은 언제나 *.rs* 확장자로 끝납니다. 만일 여러분이 한 단어 이상을 여러분의 파일에 사용하겠다면, 단어 구분을 위해서 언더스코어(_)를 사용하세요. 예를 들면, *helloworld.rs* 보다는 *hello_world.rs*를 사용하세요.

이제 여러분이 방금 만든 *main.rs*을 열고 Listing 1-1의 코드를 입력하세요.

Filename: *main.rs*

```
fn main() {
    println!("Hello, world!");
}
```

Listing 1-1: “Hello, world!”를 출력하는 프로그램

파일을 저장하고, 여러분의 터미널 윈도우로 돌아가세요. Linux나 macOS에서는 다음 커맨드를 입력하여 파일을 컴파일하고 실행하세요:

```
$ rustc main.rs
$ ./main
Hello, world!
```

Windows에서는 `./main` 대신 `.\main.exe` 커맨드를 입력하세요.

```
> rustc main.rs
> .\main.exe
Hello, world!
```

여러분의 운영체제와 상관없이, `Hello, world!` 문자열이 터미널에 출력되어야 합니다. 만일 여러분이 이 출력을 보지 못한다면, “문제 해결하기”절로 돌아가서 도움을 구할 방법을 참조하세요.

`Hello, world!`이 출력되었다면, 축하드립니다! 여러분은 공식적으로 러스트 프로그램을 작성하셨어요. 즉 러스트 프로그래머가 되셨다는 말이지요! 환영합니다!

러스트 프로그램 해부하기

여러분의 “Hello, world!” 프로그램에서 어떤 일이 벌어졌는지를 상세하게 짚어보겠습니다. 여기 첫번째 퍼즐 조각이 있습니다:

```
fn main() {  
}
```

이 라인들은 러스트의 *함수(function)*를 정의합니다. `main` 함수는 특별합니다: 이것은 모든 실행 가능한 러스트 프로그램 내에서 첫번째로 실행되는 코드입니다. 첫번째 라인은 파라미터가 없고 아무것도 반환하지 않는 `main`이라는 이름의 함수를 정의합니다. 만일 파라미터가 있었다면, 파라미터들이 괄호 `(`와 `)` 내에 위치했을 것입니다.

또한 함수의 본체가 중괄호 기호 `{`와 `}`로 감싸져 있음을 주목하세요. 러스트는 모든 함수 본체들 주위에 이것들을 요구합니다. 여는 중괄호 기호를 함수 정의부와 같은 줄에 한 칸 띄워서 위치시키는 것은 좋은 스타일입니다.

이 글을 쓰는 시점에서 `rustfmt` 라 불리우는 자동 포맷팅 도구가 개발중에 있습니다. 만일 여러분이 러스트 프로젝트를 가로지르는 표준 스타일을 고수하길 원한다면, `rustfmt`가 여러분의 코드를 특정한 스타일로 포매팅해줄 것입니다. 러스트 팀은 궁극적으로 이 도구가 `rustc`처럼 표준 러스트 배포에 포함되기를 계획하고 있습니다. 따라서 여러분이 이 책을 언제 읽는가에 따라써, 이 툴이 여러분의 컴퓨터에 이미 설치되어 있을지도 모릅니다! 더 자세한 사항에 대해서는 온라인 문서를 참고하세요.

`main` 함수 내부에는 다음과 같은 코드가 있습니다:

```
println!("Hello, world!");
```

이 라인이 이 짧은 프로그램 내의 모든 일을 합니다: 스크린에 텍스트를 출력합니다. 여기에 주목할만 한 네 가지의 중요한 디테일이 있습니다. 첫째로, 러스트 스타일은 탭이 아닌 네 개의 스페이스로 들여쓰기를 합니다.

둘째로, `println!`은 러스트 **매크로(macro)**라고 불립니다. 만일 대신에 함수라고 불리려면, (`!` 없이) `println`으로 입력되었어야 할 것입니다. 러스트 매크로에 대한 자세한 사항은 부록 D에서 다룰 것입니다. 지금은 `!`이 보통의 함수 대신 매크로를 호출하고 있음을 의미한다는 것만 알아두면 됩니다.

셋째로, 여러분은 `"Hello, world!"` *스트링(string)*을 볼 수 있습니다. 우리는 이 스트링을 `println!`의 인자로 넘기고, 이 스트링이 화면에 출력됩니다.

넷째로, 우리는 이 라인을 세미콜론 `;`으로 끝내는데, 이는 이 표현식이 끝났고 다음 것이 시작될 준비가 되었음을 나타냅니다. 대다수의 러스트 코드 라인들이 세미콜론으로 끝납니다.

컴파일과 실행은 개별적인 단계입니다

여러분이 이제 막 새로 만든 프로그램을 실행했으므로, 이 과정의 각 단계를 검토해 봅시다.

러스트 프로그램을 실행하기 전에, 여러분은 아래와 같이 `rustc` 커マン드를 입력하고 여기에 여러분의 소스 코드를 넘기는 식으로 러스트 컴파일러를 사용하여 이를 컴파일해야 합니다:

```
$ rustc main.rs
```

만일 여러분이 C 혹은 C++ 배경지식을 갖고 있다면, 이것이 `gcc` 혹은 `clang`과 유사하다는 것을 눈치챘을 것입니다. 컴파일을 성공적으로 한 뒤, 러스트는 실행가능한 바이너리를 출력합니다.

Linux, macOS, 그리고 Windows의 파워쉘 상에서는 여러분의 쉘에 다음과 같이 `ls` 커マン드를 입력하여 이 실행 파일을 볼 수 있습니다:

```
$ ls  
main main.rs
```

Windows의 CMD 환경에서는 다음과 같이 입력해야 합니다:

```
> dir /B %= the /B option says to only show the file names =%  
main.exe  
main.pdb  
main.rs
```

이 커マン드는 `.rs` 확장자를 가진 소스 코드 파일, 실행 파일 (Windows에서는 `main.exe`, 다른 모든 플랫폼에서는 `main`), 그리고 만일 CMD를 이용하는 중이라면, `.pdb` 확장자를 가지고 있는 디버깅 정보를 담고 있는 파일을 보여줍니다. 여기서 여러분은 아래와 같이 `main` 혹은 `main.exe` 파일을 실행합니다:

```
$ ./main # or .\main.exe on Windows
```

만일 `main.rs`가 여러분의 “Hello, world!” 프로그램이었다면, 위의 라인이 여러분의 터미널에 `Hello, world!`라고 출력해줄 것입니다.

여러분이 루비, 파이썬, 자바스크립트와 같은 동적 언어에 더 친숙하다면, 아마도 프로그램의 컴파일과 실행을 개별적인 단계로 이용하지 않았을지도 모릅니다. 러스트는 *ahead-of-time compiled* 언어인데, 이는 여러분이 프로그램을 컴파일하고, 그 실행파일을 다른 이들에게 주면, 그들은 러스트를 설치하지 않고도 이를 실행할 수 있다는 의미입니다. 만일 여러분이 누군가에게 `.rb`, `.py` 혹은 `.js` 파일을 준다면, 그는 (각각) 루비, 파이썬, 혹은 자바스크립트 구현체가 설치되어 있어야 합니다. 하지만 그러한 언어들에서는 하나의 커맨드로 여러분의 프로그램을 컴파일하고 실행할 수 있습니다. 언어 디자인에서는 모든 것이 트레이드 오프입니다.

간단한 프로그램에 대해 그냥 `rustc`만으로 컴파일하는 것은 괜찮지만, 여러분의 프로젝트가 커지면서, 여러분은 모든 옵션을 관리하고 여러분의 코드를 공유하기 쉽도록 하길 원할 것입니다. 다음 절에서 우리는 여러분에게 Cargo 도구를 소개할 것인데, 이것이 여러분의 실생활 러스트 프로그램 작성을 도와줄 것입니다.

Hello, Cargo!

Cargo(카고)는 러스트의 빌드 시스템 및 패키지 매니저입니다. 대부분의 러스트인들이 이 도구를 이용하여 그들의 러스트 프로젝트를 관리하는데, 그 이유는 Cargo가 여러분의 코드를 빌드하고, 여러분의 코드가 의존하고 있는 라이브러리를 다운로드해주고, 그 라이브러리들을 빌드하는 등 여러분을 위한 많은 작업들을 다루기 때문입니다. (여러분의 코드가 필요로 하는 라이브러리를 **의존성 (dependency)** 이라고 부릅니다)

여러분이 이제껏 작성한 것과 같은 가장 단순한 러스트 프로그램은 어떠한 의존성도 없습니다. 따라서 만일 Cargo를 가지고 “Hello, world!” 프로젝트를 빌드했다면, 여러분의 코드를 빌드하는 것을 다루는 카고의 일부만 일 이용하게 되었을 것입니다. 여러분이 더 복잡한 러스트 프로그램을 작성할 때면, 여러분은 의존성을 추가할 것이고, 여러분이 Cargo를 이용하여 프로젝트를 시작한다면, 의존성 추가가 훨씬 더 하기 쉬워질 것입니다.

압도적인 숫자의 러스트 프로젝트가 Cargo를 이용하기 때문에, 이 책의 나머지 부분에서는 여러분 또한 Cargo를 이용하고 있음을 가정합니다. 만일 여러분이 “설치하기” 절에서 다른대로 공식 인스톨러를 이용했다면 Cargo는 러스트와 함께 설치되어 있습니다. 만일 여러분이 다른 수단을 통해 러스트를 설치했다면, Cargo가 설치되어 있는지 확인하기 위해서 여러분의 터미널에 다음을 입력해보세요:

```
$ cargo --version
```

버전 숫자가 보인다면, 가지고 있는 것입니다! **command not found** 같은 에러를 보게 된다면, 여러분이 설치한 방법에 대한 문서에서 Cargo를 개별적으로 어떻게 설치하는지 찾아보세요.

Cargo를 사용하여 프로젝트 생성하기

Cargo를 사용하여 새 프로젝트를 만들고 우리의 원래 “Hello, world!” 프로젝트와 얼마나 차이가 나는지 살펴봅시다. 여러분의 *projects* 디렉토리로 (혹은 여러분의 코드를 저장하기로 결정한 어느 곳이든) 이동하세요. 그 다음, 어떤 운영체제이든 상관없이 다음을 실행하세요:

```
$ cargo new hello_cargo --bin
$ cd hello_cargo
```

첫번째 커マン드는 *hello_cargo*라고 불리우는 새로운 실행 가능한 바이너리를 생성합니다. **cargo new**에게 넘겨지는 **--bin** 인자가 라이브러리가 아닌 실행 가능한 애플리케이션으로 만들어줍니다 (흔히들 그냥 **바이너리 (binary)**라고 부릅니다). 우리의 프로젝트는 *hello_cargo*라고 이름지었고, Cargo는 동일한 이름의 디렉토리에 이 프로젝트의 파일들을 생성합니다.

hello_cargo 디렉토리로 가서 파일 리스트를 보세요. 여러분은 Cargo가 우리를 위해 두 개의 파일과 하나의 디렉토리를 생성한 것을 볼 수 있을 것입니다: *Cargo.toml* 파일 및 안에 *main.rs* 파일을 담고 있는 *src* 디렉

토리가 그것입니다. 안에는 또한 `.gitignore`과 함께 새로운 Git 저장소도 초기화되어 있습니다.

노트: Git은 보편적인 버전 관리 시스템입니다. 여러분은 `--vcs` 플래그를 사용하여 `cargo new` 가 다른 버전 관리 시스템을 사용하거나 혹은 버전 관리 시스템을 사용하지 않도록 변경할 수 있습니다. 사용 가능한 옵션을 보려면 `cargo new --help`를 실행하세요.

`Cargo.toml`을 여러분이 원하는 텍스트 에디터로 여세요. 이 파일은 Listing 1-2의 코드와 유사하게 보여야 합니다.

Filename: `Cargo.toml`

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

Listing 1-2: `cargo new`가 생성한 `Cargo.toml` 내용

이 파일은 **TOML** (Tom's Obvious, Minimal Language) 포맷으로 작성되었는데, 이것이 Cargo의 환경 설정 포맷입니다.

첫번째 라인 `[package]`은 이후의 문장들이 패키지 환경설정이라는 것을 나타내는 섹션의 시작지점입니다. 우리가 이 파일에 더 많은 정보를 추가하기 위해, 다른 섹션들을 추가할 것입니다.

그 다음 세 라인들은 Cargo가 여러분의 프로그램을 컴파일하기 위해 필요로 하는 정보에 대한 설정을 합니다: 이름, 버전, 그리고 누가 작성했는가 입니다. Cargo는 여러분의 환경으로부터 여러분의 이름과 이메일 정보를 얻어내므로, 만일 그 정보가 정확하지 않다면, 지금 수정하고 파일을 저장하세요.

마지막 라인 `[dependencies]`은 여러분 프로젝트의 의존성들의 리스트를 적을 수 있는 섹션의 시작점입니다. 러스트에서는 코드의 패키지를 **크레이트 (crate)**라고 부릅니다. 이 프로젝트를 위해서는 어떤 다른 크레이트도 필요없지만, 2장의 첫 프로젝트에서는 필요할 것이므로, 그때 이 의존성 섹션을 사용하겠습니다.

이제 `src/main.rs`을 열어서 살펴봅시다:

Filename: `src/main.rs`

```
fn main() {
    println!("Hello, world!");
}
```

Cargo는 우리가 Listing 1-1에서 작성했던 것과 똑같이 여러분을 위해 “Hello, world!” 프로그램을 작성해 놨습니다! 여기까지, 우리의 이전 프로젝트와 Cargo가 만든 프로젝트 간의 차이점은 Cargo가 코드를 `src` 디렉토리 안에 위치시킨다는 점, 그리고 최상위 디렉토리에 `Cargo.toml` 환경 파일을 가지게 해준다는 점입니다.

Cargo는 여러분의 소스 파일들이 `src` 디렉토리 안에 있을 것으로 예상합니다. 최상위 프로젝트 디렉토리는 그저 README 파일들, 라이센스 정보, 환경 파일들, 그리고 여러분의 코드와는 관련이 없는 다른 것들 뿐입니다. Cargo를 이용하는 것은 여러분이 프로젝트를 조직화하는 데에 도움을 줍니다. 모든 것을 위한 공간이 있고, 모든 것은 자신의 공간 안에 있습니다.

만일 여러분이 Hello, world! 프로젝트에서 했던 것처럼 Cargo를 사용하지 않은 프로젝트를 시작했다면, Cargo를 사용한 프로젝트로 이를 바꿀 수 있습니다. 프로젝트 코드를 `src` 디렉토리로 옮기고 적합한 `Cargo.toml` 파일을 생성하세요.

Cargo 프로젝트를 빌드하고 실행하기

이제 Cargo로 만든 “Hello, world!” 프로젝트를 빌드하고 실행할 때의 차이점을 살펴봅시다! `hello_cargo` 디렉토리에서, 다음 커맨드를 입력하는 것으로 여러분의 프로젝트를 빌드하세요:

```
$ cargo build
Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

이 커맨드는 여러분의 현재 디렉토리 대신 `target/debug/hello_cargo`에 (혹은 Windows에서는 `target\debug\hello_cargo.exe`에) 실행 파일을 생성합니다. 여러분은 아래 커맨드를 통해 이 실행 파일을 실행할 수 있습니다:

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on Windows
Hello, world!
```

만일 모든 것이 잘 진행되었다면, 터미널에 `Hello, world!` 가 출력되어야 합니다. 처음으로 `cargo build`를 실행하는 것은 또한 Cargo가 최상위 디렉토리에 `Cargo.lock`이라는 새로운 파일을 생성하도록 합니다. 이 프로젝트는 어떠한 의존성도 가지고 있지 않으므로, 파일의 내용이 얼마 없습니다. 여러분이 이 파일을 손수 변경할 필요는 전혀 없습니다; Cargo가 여러분을 위해 이 파일의 내용을 관리합니다.

우리는 그저 `cargo build`로 프로젝트를 빌드하고 `./target/debug/hello_cargo`로 이를 실행했지만, 또한 `cargo run`을 사용하여 한번의 커맨드로 코드를 컴파일한 다음 결과 실행파일을 실행할 수 있습니다:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/hello_cargo`
Hello, world!
```

이번에는 Cargo가 `hello_cargo`를 컴파일하는 중이었다는 것을 나타내는 출력을 볼 수 없음을 주목하세요. Cargo는 파일들이 변경된 적이 없음을 알아내고, 따라서 해당 바이너리를 그저 실행했을 뿐입니다. 만일 여러분이 여러분의 코드를 수정한 적 있다면, Cargo는 그 프로젝트를 실행하기 전에 다시 빌드할 것이고, 여러분은 아래와 같은 출력을 보게될 것입니다:

```
$ cargo run
  Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
  Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
    Running `target/debug/hello_cargo`
Hello, world!
```

Cargo는 또한 `cargo check`라고 하는 커マン드를 제공합니다. 이 커マン드는 여러분의 코드가 컴파일되는지를 빠르게 확인해주지만 실행파일을 생성하지는 않습니다:

```
$ cargo check
  Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
  Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

왜 여러분이 실행파일을 원치 않게 될까요? 종종 `cargo check`가 `cargo build`에 비해 훨씬 빠른데, 그 이유는 이 커マン드가 실행파일을 생성하는 단계를 생략하기 때문입니다. 만일 여러분이 코드를 작성하는 동안 계속적으로 여러분의 작업물을 검사하는 중이라면, `cargo check`를 이용하는 것이 그 과정의 속도를 높여 줄 것입니다! 그런 이유로, 많은 러스트인들이 자신들의 프로그램을 작성하면서 이것이 컴파일 되는지 확인하기 위해 주기적으로 `cargo check`를 실행합니다. 그런 다음 실행파일을 사용할 준비가 되었을 때 `cargo build`를 실행합니다.

여태까지 Cargo에 대하여 우리가 배운 것들을 정리하자면:

- 우리는 `cargo build`나 `cargo check`를 사용하여 프로젝트를 빌드할 수 있습니다.
- 우리는 `cargo run`을 사용하여 단숨에 프로젝트를 빌드하고 실행할 수 있습니다.
- 우리 코드가 있는 동일한 디렉토리에 빌드의 결과물이 저장되는 대신, Cargo는 이를 `target/debug` 디렉토리에 저장합니다.

Cargo를 사용하면 생기는 추가적인 장점은 여러분이 어떠한 운영체제로 작업을 하든 상관없이 커맨드들이 동일하다는 점입니다. 따라서 이러한 점 때문에 우리는 더 이상 Linux와 macOS 및 Windows를 위한 특정 명령을 제공하지 않을 것입니다.

릴리즈 빌드

여러분의 프로젝트가 마침내 배포(릴리즈)를 위한 준비가 되었다면, `cargo build --release`를 사용하여 최적화와 함께 이를 컴파일할 수 있습니다. 이 커맨드는 `target/debug` 대신 `target/release`에 실행파일을 생성할 것입니다. 최적화는 여러분의 러스트 코드를 더 빠르게 만들어주지만, 최적화를 켜는 것은 여러분의 프로그램을 컴파일하는데 드는 시간을 길게 할 것입니다: 이것이 바로 두 개의 서로 다른 프로파일이 있는 이유입니다: 하나는 여러분이 빠르게 그리고 자주 다시 빌드하기를 원하는 개발용, 그리고 다른 하나는 반복적으로 다시 빌드를 할 필요 없고 가능한 빠르게 실행되어 여러분이 사용자들에게 제공할 최종 프로그램을 빌드하기 위한 용도입니다. 만일 여러분이 코드의 실행 시간을 벤치마킹 중이라면, `cargo build --release`를 실행하고 `target/release`의 실행파일을 가지고 벤치마킹하고 있음을 확인하세요.

관례로서의 Cargo

단순한 프로젝트와 함께 Cargo를 사용하는 것은 그냥 `rustc`을 이용하는 것에 비해 큰 가치를 제공해주지는 못합니다만, 여러분의 프로그램이 점점 더 복잡해질수록 Cargo는 자신의 가치를 증명할 것입니다. 여러 개의 크레이트들로 구성된 복잡한 프로젝트와 함께라면 Cargo가 빌드를 조직화하도록 하는 것이 훨씬 쉽습니다.

비록 `hello_cargo` 프로젝트가 단순했을지라도, 이 프로젝트는 이제 여러분의 남은 러스트 경력 생활 내에 사용하게 될 진짜배기 도구를 사용하였습니다. 사실, 어떤 기존 프로젝트들 상에서 작업을 하기 위해서, 여러분은 Git을 사용하여 코드를 체크 아웃하고 그 프로젝트 디렉토리로 가서 빌드하기 위해 다음 커맨드를 사용할 수 있습니다:

```
$ git clone someurl.com/someproject  
$ cd someproject  
$ cargo build
```

Cargo에 대해 더 많은 정보를 보려면 [문서](#)를 참고하세요.

정리

여러분은 이미 여러분의 러스트 여정에서 아주 좋은 출발을 하고 있습니다! 이 장에서는 아래 항목들을 어떻게 하는지에 대해 배웠습니다:

- `rustup`을 사용하여 최신의 안정화된 러스트 버전 설치하기
- 더 최근에 나온 러스트 버전으로 업데이트하기
- 로컬에 설치된 문서 열기

- `rustc`를 직접 사용하여 “Hello, world!” 프로그램을 작성하고 실행하기
- Cargo의 관례를 사용하여 새로운 프로젝트를 만들고 실행하기

이제 러스트 코드를 읽고 쓰는데 익숙해지기 위해서 좀더 상당한 프로그램을 빌드하기 좋은 시간입니다. 따라서 다음 장에서는 추리 게임 프로그램을 빌드해 볼 것입니다. 만약 그보다 러스트에서 어떻게 보편적인 프로그래밍 개념이 동작하는지를 배우는 것으로 시작하길 원한다면, 3장을 먼저 보시고 2장으로 돌아오세요.

추리 게임

실습 프로젝트를 통해 러스트를 사용해 봅시다. 이번 장은 실제 프로젝트에서 몇몇 일반적인 Rust 개념이 어떻게 활용되는지를 소개하려 합니다. 이 과정에서 `let`, `match`, 메소드, 연관함수(associated functions), 외부 크레이트(external crates) 등의 활용 방법을 배울 수 있습니다. 이런 개념들은 다음 장들에서 더 자세히 다뤄질 것입니다. 이번 장에서는 여러분이 직접 기초적인 내용을 실습합니다.

우리는 고전적인 입문자용 프로그래밍 문제인 추리 게임을 구현해 보려 합니다. 이 프로그램은 1~100 사이의 임의의 정수를 생성합니다. 다음으로 플레이어가 프로그램에 추리한 정수를 입력합니다. 프로그램은 입력 받은 추리값이 정답보다 높거나 낮은지를 알려줍니다. 추리값이 정답이라면 축하 메세지를 보여주고 종료됩니다.

새로운 프로젝트를 준비하기

새로운 프로젝트를 준비하기 위해 1장에서 생성했던 디렉토리인 *projects*로 이동하고 아래 예제처럼 Cargo를 이용하여 새로운 프로젝트를 생성합니다.

```
$ cargo new guessing_game --bin
$ cd guessing_game
```

첫 명령문인 `cargo new`는 프로젝트의 이름 (`guessing_game`)을 첫번째 인자로 받습니다. `--bin` 플래그는 Cargo가 1장과 비슷하게 바이너리용 프로젝트를 생성하도록 합니다. 두번째 명령문은 작업 디렉토리를 새로운 프로젝트의 디렉토리로 변경합니다.

생성된 *Cargo.toml* 파일을 살펴봅시다.

Filename: *Cargo.toml*

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

만약 Cargo가 환경변수에서 가져온 author 정보가 잘못되었다면 파일을 수정하고 저장하면 됩니다.

1장에서 보았듯이 `cargo new`는 여러분을 위해 "Hello, world!" 프로그램을 생성합니다. *src/main.rs* 파일을 살펴보면 다음과 같습니다.

Filename: src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

이제 이 "Hello, world!" 프로그램을 `cargo run` 명령문을 이용하여 컴파일하고 실행해 봅시다.

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
Running `target/debug/guessing_game`
Hello, world!
```

`run` 명령어는 이번 실습 프로젝트처럼 빠르게 반복(iteration)을 하고 싶을 때 유용합니다. 우리는 다음 iteration으로 넘어가기 전 빠르게 각 iteration을 테스트하고 싶습니다.

src/main.rs 를 다시 열어 두세요. 이 파일에 모든 코드를 작성할 것입니다.

추리값을 처리하기

프로그램의 첫 부분은 사용자 입력 요청, 입력값의 처리 후 입력값이 기대하던 형식인지 검증합니다. 첫 시작으로 플레이어가 추리한 값을 입력받을 수 있게 할 것입니다. Listing 2-1의 코드를 *src/main.rs* 에 작성하세요.

Filename: src/main.rs

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");
    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Listing 2-1: 사용자가 추리한 값을 입력 받아 그대로 출력하는 코드

이 코드에 담긴 다양한 정보를 하나씩 살펴 보겠습니다. 사용자 입력을 받고 결과값을 표시하기 위해서는 `io` (input/output) 라이브러리를 스코프로 가져와야 합니다. `io` 라이브러리는 `std`라고 불리는 표준 라이브러리에 있습니다.

```
use std::io;
```

러스트는 모든 프로그램의 스코프에 *prelude* 내의 타입들을 가져옵니다. 만약 여러분이 원하는 타입이 *prelude*에 없다면 `use` 문을 활용하여 명시적으로 그 타입을 가져와야 합니다. `std::io`는 사용자의 입력을 받는 것을 포함하여 `io`와 관련된 기능들을 제공합니다.

1장에서 보았듯이 `main` 함수는 프로그램의 진입점입니다.

```
fn main() {
```

`fn` 문법은 새로운 함수를 선언하며 `()`는 인자가 없음을 나타내고 `{`는 함수 본문의 시작을 나타냅니다.

1장에서 배웠듯이 `println!` 은 `string`을 화면에 표시하는 매크로입니다.

```
println!("Guess the number!");  
println!("Please input your guess.");
```

이 코드는 게임에 대한 설명과 사용자의 입력을 요청하는 글자를 표시합니다.

값을 변수에 저장하기

다음으로 우리는 다음 코드처럼 사용자의 입력값을 저장할 공간을 생성할 수 있습니다.

```
let mut guess = String::new();
```

이제 프로그램이 점점 흥미로워지고 있습니다! 이 짧은 라인에서 여러 일들이 벌어집니다. 이 라인이 변수를 생성하는 `let` 문장을 주목하세요. 다음 코드도 변수를 선언하는 예시입니다.

```
let foo = bar;
```

이 라인은 `foo`라는 변수를 선언하고 `bar`라는 값과 묶습니다. 러스트에서 변수는 기본적으로 불변입니다. 다음 예시는 변수 앞에 `mut`을 이용하여 가변변수를 만드는 법을 보여줍니다.

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

Note: // 문법은 현재 위치부터 라인의 끝까지 주석임을 나타냅니다. 러스트는 주석의 모든 내용을 무시합니다.

이제 `let mut guess`가 `guess`라는 이름의 가변변수임을 알 수 있습니다. `=`의 반대편의 값은 `guess`와 묶이게 되는데 이번 예시에서는 함수 `String::new`의 결과값인 새로운 `String` 인스턴스가 묶이는 대상이 됩니다. `String`은 표준 라이브러리에서 제공하는 확장 가능한(growable) UTF-8 인코딩의 문자열 타입입니다.

`::new`에 있는 `::`는 `new`가 `String` 타입의 연관함수임을 나타냅니다. 연관함수는 하나의 타입을 위한 함수이며, 이 경우에는 하나의 `String` 인스턴스가 아니라 `String` 타입을 위한 함수입니다. 몇몇 언어에서는 이것을 정적 메소드라고 부릅니다.

`new` 함수는 새로운 빈 `String`을 생성합니다. `new` 함수는 새로운 값을 생성하기 위한 일반적인 이름이므로 많은 타입에서 찾아볼 수 있습니다.

요약하자면 `let mut guess = String::new();` 라인은 새로운 빈 `String` 인스턴스와 연결된 가변 변수를 생성합니다.

프로그램에 첫번째 라인에 `use std::io;` 를 이용하여 표준 라이브러리의 input/output 기능을 포함한 것을 떠올려 보세요. 이제 우리는 `io`의 연관함수인 `stdin`을 호출합니다.

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

만약 프로그램 시작점에 `use std::io`가 없다면 함수 호출 시 `std::io::stdin`처럼 작성해야 합니다. `stdin` 함수는 터미널의 표준 입력의 핸들(handle)의 타입인 `std::io::Stdin`의 인스턴스를 돌려줍니다.

코드의 다음 부분인 `.read_line(&mut guess)`는 사용자로부터 입력을 받기 위해 표준 입력 핸들에서 `.read_line(&mut guess)` 메소드를 호출합니다. 또한 `read_line`에 `&mut guess`를 인자로 하나 넘깁니다.

`read_line`은 사용자가 표준 입력에 입력할 때마다 입력된 문자들을 하나의 문자열에 저장하므로 인자로 값을 저장할 문자열이 필요합니다. 그 문자열 인자는 사용자 입력을 추가하면서 변경되므로 가변이어야 합니다.

`&`는 코드의 여러 부분에서 데이터를 여러 번 메모리로 복사하지 않고 접근하기 위한 방법을 제공하는 참조자임을 나타냅니다. 참조자는 복잡한 특성으로서 러스트의 큰 이점 중 하나가 참조자를 사용함으로써 얻는 안전성과 용이성입니다. 이 프로그램을 작성하기 위해 참조자의 자세한 내용을 알 필요는 없습니다. 4장에서 참조자에 대해 전체적으로 설명할 것입니다. 지금 당장은 참조자가 변수처럼 기본적으로 불변임을 알기만 하면 됩니다. 따라서 가변으로 바꾸기 위해 `&guess`가 아니라 `&mut guess`로 작성해야 합니다.

아직 이 라인에 대해 다 설명하지 않았습니다. 한 라인처럼 보이지만 사실은 이 라인과 논리적으로 연결된 라인이 더 있습니다. 두번째 라인은 다음 메소드입니다.

```
.expect("Failed to read line");
```

`.foo()` 형태의 문법으로 메소드를 호출할 경우 긴 라인을 나누기 위해 다음 줄과 여백을 넣는 것이 바람직합니다. 위 코드를 아래처럼 쓸 수도 있습니다.

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

하지만 하나의 긴 라인은 가독성이 떨어지므로 두 개의 메소드 호출을 위한 라인으로 나누는 것이 좋습니다. 이제 이 라인이 무엇인지에 대해 이야기해 봅시다.

Result 타입으로 잠재된 실패 다루기

이전에 언급한 것처럼 `read_line`은 우리가 인자로 넘긴 문자열에 사용자가 입력을 저장할 뿐 아니라 하나의 값을 돌려 줍니다. 여기서 돌려준 값은 `io::Result`입니다. 러스트는 표준 라이브러리에 여러 종류의 `Result` 타입을 가지고 있습니다. 제네릭 `Result`이나 `io::Result`가 그 예시입니다.

`Result` 타입은 [열거형\(enumerations\)](#)로써 `enums`라고 부르기도 합니다. 열거형은 정해진 값들만을 가질 수 있으며 이러한 값들은 열거형의 `variants`라고 부릅니다. 6장에서 열거형에 대해 더 자세히 다룹니다.

`Result`의 `variants`는 `Ok`와 `Err`입니다. `Ok`는 처리가 성공했음을 나타내며 내부적으로 성공적으로 생성된 결과를 가지고 있습니다. `Err`는 처리가 실패했음을 나타내고 그 이유에 대한 정보를 가지고 있습니다.

이러한 `Result`는 에러 처리를 위한 정보를 표현하기 위해 사용됩니다. `Result` 타입의 값들은 다른 타입들처럼 메소드들을 가지고 있습니다. `io::Result` 인스턴스는 `expect` 메소드를 가지고 있습니다. 만약 `io::Result` 인스턴스가 `Err`일 경우 `expect` 메소드는 프로그램이 작동을 멈추게 하고 `expect`에 인자로 넘겼던 메세지를 출력하도록 합니다. 만약 `read_line` 메소드가 `Err`를 돌려줬다면 그 에러는 운영체제로부터 생긴 에러일 경우가 많습니다. 만약 `io::Result`가 `Ok` 값이라면 `expect`는 `Ok`가 가지고 있는 결과값을 돌려주어 사용할 수 있도록 합니다. 이 경우 결과값은 사용자가 표준 입력으로 입력했던 바이트의 개수입니다.

만약 `expect`를 호출하지 않는다면 컴파일은 되지만 경고가 나타납니다.

```
$ cargo build
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
--> src/main.rs:10:5
  |
10 |     io::stdin().read_line(&mut guess);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
= note: #[warn(unused_must_use)] on by default
```

러스트는 `read_line`가 돌려주는 `Result` 값을 사용하지 않았음을 경고하며 일어날 수 있는 에러를 처리하지 않았음을 알려줍니다. 이 경고를 없애는 옳은 방법은 에러를 처리하는 코드를 작성하는 것이지만 만약 문제가 발생했을 때 프로그램이 멈추길 바란다면 `expect`를 사용할 수 있습니다. 9장에서 에러가 발생했을 때 이를 처리하는 방법에 대해 배웁니다.

println! 변경자(placeholder)를 이용한 값 출력

지금까지 작성한 코드에서 닫는 중괄호 말고도 살펴봐야 하는 코드가 하나 더 있습니다. 내용은 아래와 같습니다.

```
println!("You guessed: {}", guess);
```

이 라인은 사용자가 입력한 값을 저장한 문자열을 출력합니다. `{}`는 변경자로써 값이 표시되는 위치를 나타냅니다. `{}`를 이용하여 하나 이상의 값을 표시할 수도 있습니다. 첫번째 `{}`는 형식 문자열(format string) 이후의 첫번째 값을 표시하며, 두번째 `{}`는 두번째 값을 나타내며 이후에도 비슷하게 작동합니다. 다음 코드는 `println!`를 이용하여 여러 값을 표시하는 방법을 보여줍니다.

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

이 코드는 `x = 5 and y = 10`을 출력합니다.

첫번째 부분을 테스트하기

추리 게임의 처음 부분을 테스트 해 봅시다. `cargo run`을 통해 실행할 수 있습니다.

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

지금까지 게임의 첫번째 부분을 작성했습니다. 우리는 입력값을 받고 그 값을 출력했습니다.

비밀번호를 생성하기

다음으로 사용자가 추리하기 위한 비밀번호를 생성해야 합니다. 게임을 다시 하더라도 재미있도록 비밀번호는 매번 달라야 합니다. 게임이 너무 어렵지 않도록 1에서 100 사이의 임의의 수를 사용합시다. 러스트는 아직 표준 라이브러리에 임의의 값을 생성하는 기능이 없습니다. 하지만 러스트 팀에서는 **rand** 크레이트를 제공합니다.

크레이트(Crate)를 사용하여 더 많은 기능 가져오기

크레이트는 러스트 코드의 묶음(package)임을 기억하세요. 우리가 만들고 있는 프로젝트는 실행이 가능한 *binary crate*입니다. **rand** crate는 다른 프로그램에서 사용되기 위한 용도인 *library crate*입니다.

Cargo에서 외부 크레이트의 활용이 정말 멋진 부분입니다. **rand**를 사용하는 코드를 작성하기 전에 *Cargo.toml*을 수정하여 **rand** 크레이트를 의존 리스트에 추가해야 합니다. 파일을 열고 Cargo가 여러분을 위해 생성한 **[dependencies]** 절의 시작 바로 아래에 다음 내용을 추가하세요.

Filename: *Cargo.toml*

```
[dependencies]
```

```
rand = "0.3.14"
```

Cargo.toml 파일에서 하나의 절의 시작 이후의 모든 내용은 그 절에 포함되며 이는 다음 절이 나타날 때까지 동일합니다. **[dependencies]** 절은 여러분의 프로젝트가 의존하고 있는 외부 크레이트와 각각의 요구 버전을 Cargo에 명시하는 곳입니다. 지금의 경우 우리는 **rand** 크레이트의 유의적 버전인 **0.3.14**를 명시했습니다. Cargo는 버전을 명시하는 표준에 해당하는 *Semantic Versioning(semver)*을 이용합니다.

0.3.14는 **^0.3.14**의 축약형이 되며 이는 버전 **0.3.14**와 호환되는 API를 제공하는 모든 버전임을 의미합니다.

미합니다.

이제 Listing 2-2처럼 코드 수정 없이 프로젝트를 빌드 해 봅시다.

```
$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading rand v0.3.14
  Downloading libc v0.2.14
    Compiling libc v0.2.14
    Compiling rand v0.3.14
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Listing 2-2: rand 크레이트를 의존성으로 추가한 후 `cargo build` 를 실행한 결과

여러분은 다른 버전명이나 라인의 순서가 다르게 보일 수 있습니다. 버전명이 다르더라도 SemVer 덕분에 현재 코드와 호환될 것입니다.

이제 우리는 외부 의존성을 가지게 되었고, Cargo는 [Crates.io](#) 데이터의 복사본인 레지스트리(registry)에서 모든 것들을 가져옵니다. Crates.io는 러스트의 생태계의 개발자들이 다른 사람들도 이용할 수 있도록 러스트 오픈소스를 공개하는 곳입니다.

레지스트리를 업데이트하면 Cargo는 `[dependencies]` 절을 확인하고 아직 여러분이 가지고 있지 않은 것들을 다운 받습니다. 이 경우 우리는 `rand`만 의존한다고 명시했지만 `rand`는 `libc`에 의존하기 때문에 `libc`도 다운 받습니다. 러스트는 이것들을 다운받은 후 컴파일 하여 의존성이 해결된 프로젝트를 컴파일합니다.

만약 아무것도 변경하지 않고 `cargo build`를 실행한다면 어떠한 결과도 얻지 못합니다. Cargo는 이미 의존 패키지들을 다운받고 컴파일했음을 알고 있고 여러분이 `Cargo.toml` 를 변경하지 않은 것을 알고 있습니다. 또한 Cargo는 코드가 변경되지 않은 것도 알고 있기에 코드도 다시 컴파일하지 않습니다. 아무것도 할 일이 없기에 그냥 종료될 뿐입니다. 만약 여러분이 `src/main.rs` 파일을 열어 사소한 변경을 하고 저장한 후 다시 빌드를 한다면 한 라인이 출력됨을 확인할 수 있습니다.

```
$ cargo build
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
```

이 라인은 Cargo가 `src/main.rs` 의 사소한 변경을 반영하여 빌드를 업데이트 했음을 보여줍니다. 의존 패키지가 변경되지 않았으므로 Cargo는 이미 다운받고 컴파일된 것들을 재사용할 수 있음을 알고 있습니다. 따라서 Cargo는 여러분의 코드에 해당하는 부분만을 다시 빌드합니다.

재현 가능한 빌드를 보장하는 `Cargo.lock`

Cargo는 여러분뿐만이 아니라 다른 누구라도 여러분의 코드를 빌드할 경우 같은 산출물이 나오도록 보장하는 방법을 가지고 있습니다. Cargo는 여러분이 다른 의존성을 추가하지 전까지는 여러분이 명시한 의존 패키지만을 사용합니다. 예로 `rand` 크레이트의 다음 버전인 `v0.3.15`에서 중요한 결함이 고쳐졌지만 당신의 코드를 망치는 변경점(regression)이 있다면 어떻게 될까요?

이 문제의 해결책은 여러분이 처음 `cargo build`를 수행할 때 생성되어 이제 `guessing_game` 디렉토리 내에 존재하는 `Cargo.lock`입니다. 여러분이 처음 프로젝트를 빌드할 때 Cargo는 기준을 만족하는 모든 의존 패키지의 버전을 확인하고 `Cargo.lock`에 이를 기록합니다. 만약 여러분이 미래에 프로젝트를 빌드할 경우 Cargo는 모든 버전들을 다시 확인하지 않고 `Cargo.lock` 파일이 존재하는지 확인하여 그 안에 명시된 버전들을 사용합니다. 이는 여러분이 재현 가능한 빌드를 자동으로 가능하게 합니다. 즉 여러분의 프로젝트는 `Cargo.lock` 덕분에 당신이 명시적으로 업그레이드하지 않는 이상 `0.3.14`를 이용합니다.

크레이트를 새로운 버전으로 업그레이드하기

만약 당신이 정말 크레이트를 업데이트하고 싶은 경우를 위해 Cargo는 `update` 명령어를 제공합니다. 이것은 `Cargo.lock` 파일을 무시하고 `Cargo.toml`에 여러분이 명시한 요구사항에 맞는 최신 버전을 확인합니다. 만약 이 버전들로 문제가 없다면 Cargo는 해당 버전을 `Cargo.lock`에 기록합니다.

하지만 Cargo는 기본적으로 `0.3.0`보다 크고 `0.4.0`보다 작은 버전을 찾을 것입니다. 만약 `rand` 크레이트가 새로운 두 개의 버전인 `0.3.15`와 `0.4.0`을 릴리즈했다면 여러분이 `cargo update`를 실행했을 때 다음의 메세지를 볼 것입니다.

```
$ cargo update
Updating registry `https://github.com/rust-lang/crates.io-index`
Updating rand v0.3.14 -> v0.3.15
```

이 시점에 여러분은 `Cargo.lock` 파일에서 변경이 일어난 것과 앞으로 사용될 `rand` 크레이트의 버전이 `0.3.15`임을 확인할 수 있습니다.

만약 여러분이 `0.4.0`이나 `0.4.x`에 해당하는 모든 버전을 받고 싶다면 `Cargo.toml`을 다음과 같이 업데이트해야 합니다.

```
[dependencies]
rand = "0.4.0"
```

다음번에 여러분이 `cargo build`를 실행하면 Cargo는 가용 가능한 크레이트들의 레지스트리를 업데이트 할 것이고 여러분의 `rand` 요구사항을 새롭게 명시한 버전에 따라 재계산할 것입니다.

[Cargo와 그의 생태계](#)에 대해 더 많은 것들은 14장에서 다뤄지지만 지금 당장은 이 정도만 알면 됩니다. Cargo는 라이브러리의 재사용을 쉽게 하여 러스트 사용자들이 많은 패키지들과 결합된 더 작은 프로젝트들

을 작성할 수 있도록 도와줍니다.

임의의 숫자를 생성하기

이제 `rand`를 사용해 봅시다. 다음 단계는 `src/main.rs`를 Listing 2-3처럼 업데이트하면 됩니다.

Filename: `src/main.rs`

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");
    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Listing 2-3: 임의의 숫자를 생성하기 위해 필요한 코드

우리는 `extern crate rand;`을 추가하여 러스트에게 우리가 외부에 의존하는 크레이트가 있음을 알립니다. 이 라인은 `use rand`으로도 표기할 수 있으며 이제 우리는 `rand::`를 앞에 붙여 `rand`내의 모든 것을 호출할 수 있습니다.

다음으로 우리는 또 다른 `use` 라인인 `use rand::Rng`를 추가합니다. `Rng`는 정수 생성기가 구현한 메소드들을 정의한 trait이며 해당 메소드들을 이용하기 위해서는 반드시 스코프 내에 있어야 합니다. 10장에서 trait에 대해 더 자세히 다룰 것입니다.

또한 우리는 중간에 두 개의 라인을 추가합니다. `rand::thread_rng` 함수는 OS가 시드(seed)를 정하고 현재 스레드에서만 사용되는 특별한 정수생성기를 돌려 줍니다. 다음으로 우리는 `get_range` 메소드를 호출합니다. 이 메소드는 `Rng` trait에 정의되어 있으므로 `use rand::Rng` 문을 통해 스코프로 가져올 수 있습니다. `gen_range` 메소드는 두 개의 숫자를 인자로 받고 두 숫자 사이에 있는 임의의 숫자를 생성합니

다. 하한선은 포함되지만 상한선은 제외되므로 1부터 100 사이의 숫자를 생성하려면 1과 101을 넘겨야 합니다.

크레이트에서 어떤 trait를 사용하고 어떤 함수나 메소드들을 호출하는 것을 아는 것은 단순히 아는 것이 아닙니다. 각각의 크레이트의 문서에서 사용 방법을 제공합니다. Cargo의 또 다른 멋진 특성은 cargo doc --open 명령어로써 로컬에서 여러분의 모든 의존 패키지들이 제공하는 문서들을 빌드해서 브라우저에 표시해 줍니다. 만약 rand 크레이트의 다른 기능들에 흥미가 있다면 cargo doc --open을 실행하고 왼쪽의 사이드바에 rand를 클릭하세요.

코드에 추가한 두 번째 라인은 비밀번호를 표시합니다. 이 라인은 우리가 프로그램을 개발 중일 때 테스트를 할 수 있도록 하지만 최종 버전에서는 삭제할 것입니다. 게임을 시작하자마자 정답을 출력하는 게임은 그다지 많지 않으니까요!

이제 프로그램을 몇 번 실행해 봅시다.

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

매 실행마다 다른 숫자면서 1부터 100 사이의 숫자가 나타나야 합니다. 잘 하셨습니다!

비밀번호와 추리값을 비교하기

이제 우리는 입력값과 임의의 정수를 가지고 있음으로 비교가 가능합니다. Listing 2-4는 그 단계를 보여주고 있습니다.

Filename: src/main.rs

```

extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less     => println!("Too small!"),
        Ordering::Greater  => println!("Too big!"),
        Ordering::Equal    => println!("You win!"),
    }
}

```

Listing 2-4: 두 숫자를 비교한 결과 처리하기

처음으로 나타난 새로운 요소는 표준 라이브러리로부터 `std::cmp::Ordering`을 스코프로 가져오는 또 다른 `use`입니다. `Ordering`은 `Result`와 같은 열거형이지만 `Ordering`의 값은 `Less`, `Greater`, `Equal`입니다. 이것들은 여러분이 두 개의 값을 비교할 때 나올 수 있는 결과들입니다.

그리고 나서 우리는 `Ordering` 타입을 이용하는 다섯 줄을 마지막에 추가 했습니다.

```

match guess.cmp(&secret_number) {
    Ordering::Less     => println!("Too small!"),
    Ordering::Greater  => println!("Too big!"),
    Ordering::Equal    => println!("You win!"),
}

```

`cmp` 메소드는 두 값을 비교하며 비교 가능한 모든 것들에 대해 호출할 수 있습니다. 이 메소드는 비교하고 싶은 것들의 참조자를 받습니다. 여기서는 `guess`와 `secret_number`를 비교하고 있습니다. `cmp`는 `Ordering` 열거형을 돌려줍니다. 우리는 `match` 표현문을 이용하여 `cmp`가 `guess`와

`secret_number`를 비교한 결과인 `Ordering`의 값에 따라 무엇을 할 것인지 결정할 수 있습니다.

`match` 표현식은 `arm`으로 이루어져 있습니다. 하나의 `arm`은 하나의 패턴과 `match` 표현식에서 주어진 값이 패턴과 맞는다면 실행할 코드로 이루어져 있습니다. 러스트는 `match`에게 주어진 값을 `arm`의 패턴에 맞는지 순서대로 확인합니다. `match` 생성자와 패턴들은 여러분의 코드가 마주칠 다양한 상황을 표현할 수 있도록 하고 모든 경우의 수를 처리했음을 확신할 수 있도록 도와주는 강력한 특성들입니다. 이 기능들은 6장과 18장에서 각각 더 자세히 다뤄집니다.

예제로 사용된 `match` 표현식에 무엇이 일어날지 한번 따라가 봅시다. 사용자가 50을 예측했다고 하고 비밀 번호가 38이라 합시다. 50과 38을 비교하면 `cmp` 메소드의 결과는 `Ordering::Greater`입니다.

`match` 표현식은 `Ordering::Greater`를 값으로 받을 것입니다. 처음으로 마주하는 `arm`의 패턴인 `Ordering::Less`는 `Ordering::Greater`와 매칭되지 않으므로 첫번째 `arm`은 무시하고 다음으로 넘어갑니다. 다음 `arm`의 패턴인 `Ordering::Greater`는 확실히 `Ordering::Greater`와 매칭합니다! `arm`과 연관된 코드가 실행될 것이고 `Too big`가 출력될 것입니다. 이 경우 마지막 `arm`은 확인할 필요가 없으므로 `match` 표현식은 끝납니다.

하지만 Listing 2-4의 코드는 컴파일되지 않습니다. 한번 시도해 봅시다.

```
$ cargo build
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
--> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |           ^^^^^^^^^^^^^^ expected struct
`std::string::String`, found integral variable
   |
   = note: expected type `&std::string::String`
   = note:     found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

에러의 핵심은 일치하지 않는 타입이 있다고 알려 주는 것입니다. 러스트는 강한 정적 타입 시스템을 가지고 있습니다. 하지만 타입 추론도 수행합니다. 만약 `let guess = String::new()`를 작성한다면 러스트는 `guess`가 `String` 타입이어야 함을 추론할 수 있으므로 타입을 적으라고 하지 않습니다. 반대로 `secret_number`는 정수형입니다. 몇몇 숫자 타입들이 1과 100 사이의 값을 가질 수 있습니다. `i32`는 32비트 정수, `u32`는 32비트의 부호없는 정수, `i64`는 64비트의 정수이며 그 외에도 비슷합니다. 러스트는 기본적으로 우리가 다른 정수형임을 추론할 수 있는 다른 타입 정보를 제공하지 않는다면 숫자들을 `i32`으로 생각합니다. 이 에러의 원인은 러스트가 문자열과 정수형을 비교하지 않기 때문입니다.

최종적으로 우리는 추리값을 정수형으로 비교하기 위해 입력으로 받은 `String`을 정수로 바꾸고 싶을 것입니다. 이것은 `main` 함수 내에 다음 두 라인을 넣어서 할 수 있습니다.

Filename: src/main.rs

```

extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}

```

두 라인은 다음과 같습니다.

```

let guess: u32 = guess.trim().parse()
    .expect("Please type a number!");

```

우리는 `guess` 변수를 생성했습니다. 잠깐, 이미 프로그램에서 `guess`라는 이름의 변수가 생성되지 않았나요? 그렇긴 하지만 러스트는 이전에 있던 `guess`의 값을 가리는(shadow) 것을 허락합니다. 이 특징은 종종 하나의 값을 현재 타입에서 다른 타입으로 변환하고 싶을 경우에 사용합니다. Shadowing은 우리들이 `guess_str`과 `guess`처럼 고유의 변수명을 만들도록 강요하는 대신 `guess`를 재사용 가능하도록 합니다. (3장에서 더 자세한 이야기를 다룹니다)

우리는 `guess`를 `guess.trim().parse()` 표현식과 묶습니다. 표현식 내의 `guess`는 입력값을 가지고

있던 `String`을 참조합니다. `String` 인스턴스의 `trim` 메소드는 처음과 끝 부분의 빈칸을 제거합니다. `u32`는 정수형 글자만을 가져야 하지만 사용자들은 `read_line`을 끝내기 위해 enter키를 반드시 눌러야 합니다. enter키가 눌리는 순간 개행문자가 문자열에 추가됩니다. 만약 사용자가 5를 누르고 enter키를 누르면 `guess`는 `5\n`처럼 됩니다. `\n`은 enter키, 즉 개행문자를 의미합니다. `trim` 메소드는 `\n`을 제거하고 `5`만 남도록 처리합니다.

문자열의 `parse` 메소드는 문자열을 숫자형으로 파싱합니다. 이 메소드는 다양한 종류의 정수형을 변환하므로 우리는 `let guess: u32`처럼 정확한 타입을 명시해야 합니다. `guess` 뒤의 콜론(:)은 변수의 타입을 명시했음을 의미합니다. 러스트는 몇몇 내장된 정수형을 가지고 있습니다. `u32`은 부호가 없는 32비트의 정수입니다. 이 타입은 작은 양수를 표현하기에는 좋은 선택입니다. 3장에서 다른 숫자형에 대해 배울 것입니다. 추가로 이 예시에서 명시했던 `u32`과 `secret_number`와의 비교는 러스트가 `secret_number`의 타입을 `u32`로 유추해야 함을 의미합니다. 이제 이 비교는 같은 타입의 두 값의 비교가 됩니다.

`parse` 메소드의 호출은 에러가 발생하기 쉽습니다. 만약 `A👍%`과 같은 문자열이 포함되어 있다면 정수로 바꿀 방법이 없습니다. "Result 타입으로 잠재된 실패 다루기"에서 `read_line`과 비슷하게 `parse` 메소드는 실패할 경우를 위해 `Result` 타입을 결과로 돌려 줍니다. 만약 `parse` 메소드가 문자열에서 정수로 파싱을 실패하여 `Err` `Result` variant를 돌려준다면 `expect` 호출은 게임을 멈추고 우리가 명시한 메세지를 출력합니다. 만약 `parse` 메소드가 성공적으로 문자열을 정수로 바꾸었다면 `Result`의 `Ok` variant를 돌려 받으므로 `expect`에서 `Ok`에서 얻고 싶었던 값을 결과로 받게 됩니다.

이제 프로그램을 실행해 봅시다!

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
76
You guessed: 76
Too big!
```

좋습니다! 추리값 앞에 빈칸을 넣더라도 프로그램은 추리값이 76임을 파악 했습니다. 추리값이 맞을 때나 너무 클 경우, 혹은 너무 작은 경우 등 여러 종류의 입력값으로 여러 시나리오를 검증해 봅시다.

우리는 게임의 대부분이 동작하도록 처리 했지만 사용자는 한 번의 추리만 가능합니다. 반복문을 추가하여 변경해 봅시다!

반복문을 이용하여 여러 번의 추리 허용

loop 키워드는 무한루프를 제공합니다. 이것을 이용하여 사용자들에게 숫자를 추리할 기회를 더 줍니다.

Filename: src/main.rs

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less     => println!("Too small!"),
            Ordering::Greater  => println!("Too big!"),
            Ordering::Equal    => println!("You win!"),
        }
    }
}
```

우리는 추리값을 입력 받는 코드부터 모든 코드들을 반복문 내로 옮겼습니다. 각각의 라인이 4간격 더 들여쓰기 되어 있음을 확실히 하고 프로그램을 다시 실행 해 보세요. 프로그램이 우리가 지시에 정확히 따르다보니 새로운 문제가 생긴 것을 확인하세요. 이제 프로그램이 영원히 다른 추리값을 요청합니다! 사용자가 이 프로그램을 종료할 수 없어요!

사용자는 **ctrl-C** 단축키를 이용하여 프로그램을 멈출 수 있습니다. 하지만 "비밀번호를 추리값과 비교하기"에서 **parse** 메소드에 대해 논의할 때 언급한 방법으로 이 만족할 줄 모르는 괴물에게서 빠져나올 수 있습니다. 만약 사용자가 숫자가 아닌 답을 적는다면 프로그램이 멈춥니다. 사용자는 프로그램 종료를 위해 다음처럼 이 장점을 활용할 수 있습니다.

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind: InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exit code: 101)
```

quit를 입력하면 게임은 확실히 끝나지만 다른 입력값들 또한 마찬가지입니다. 하지만 이것은 최소한의 차선책입니다. 우리는 정답을 입력할 경우 자동으로 게임이 끝나도록 하고 싶습니다.

정답 이후에 종료하기

사용자가 정답을 맞췄을 때 게임이 종료되도록 **break**문을 추가합니다.

Filename: src/main.rs

```

extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less     => println!("Too small!"),
            Ordering::Greater  => println!("Too big!"),
            Ordering::Equal    => {
                println!("You win!");
                break;
            }
        }
    }
}

```

`break` 문을 `You win!` 이후에 추가하여 사용자가 비밀번호를 맞췄을 때 프로그램이 반복문을 끝내도록 합니다. 반복문이 `main`의 마지막 부분이므로 반복문의 종료는 프로그램의 종료를 의미합니다.

잘못된 입력값 처리하기

사용자가 숫자가 아닌 값을 입력했을 때 프로그램이 종료되는 동작을 더 다듬어 숫자가 아닌 입력은 무시하여 사용자가 계속 입력할 수 있도록 해 봅시다. `guess` 가 `String`에서 `u32`로 변환되는 라인을 수정하면 됩니다.

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

`expect` 메소드 호출을 `match` 표현식으로 바꾸는 것은 에러 발생 시 종료에서 처리로 바꾸는 일반적인 방법입니다. `parse` 메소드가 `Result` 타입을 돌려주는 것과 `Result`는 `Ok`나 `Err` variants를 가진 열거형임을 떠올리세요. `cmp` 메소드의 `Ordering` 결과를 처리했을 때처럼 여기서 `match` 표현식을 사용하고 있습니다.

만약 `parse`가 성공적으로 문자열에서 정수로 변환했다면 결과값을 가진 `Ok`를 돌려줍니다. `Ok`는 첫번째 arm의 패턴과 매칭하게 되고 `match` 표현식은 `parse`가 생성한 `num`값을 돌려줍니다. 그 값은 우리가 생성하고 있던 새로운 `guess` 과 묶이게 됩니다.

만약 `parse`가 문자열을 정수로 바꾸지 못했다면 에러 정보를 가진 `Err`를 돌려줍니다. `Err`는 첫번째 arm의 패턴인 `Ok(num)`과 매칭하지 않지만 두 번째 arm의 `Err(_)`와 매칭합니다. `_`은 모든 값과 매칭될 수 있습니다. 이 예시에서는 `Err`내에 무슨 값이 있던지에 관계없이 모든 `Err`를 매칭하도록 했습니다. 따라서 프로그램은 두 번째 arm의 코드인 `continue`를 실행하며, 이는 `loop`의 다음 반복으로 가서 또 다른 추리값을 요청하도록 합니다. 효율적으로 프로그램은 `parse`에서 가능한 모든 에러를 무시합니다.

이제 우리가 원하는대로 프로그램이 작동해야 합니다. `cargo run`을 실행해 봅시다.

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

멋집니다! 마지막에 조금 값을 조정하여 우리는 추리 게임을 끝냈습니다. 프로그램이 여전히 비밀번호를 출력하고 있다는 것을 떠올리세요. 테스트 때는 괜찮지만 게임을 망치게 됩니다. 비밀번호를 출력하는

`println!`을 삭제합니다. Listing 2-5는 최종 코드를 보여줍니다.

Filename: src/main.rs

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Listing 2-5: 추리 게임의 완성된 코드

요약

이 시점에서 여러분은 성공적으로 추리 게임을 만들었습니다! 축하합니다!

이 프로젝트는 `let`, `match`, 메소드, 연관함수, 외부 크레이트 사용과 같은 많은 새로운 러스트 개념들을 소개하기 위한 실습이었습니다. 다음 장들에서는 이 개념들의 세부적인 내용을 배울 것입니다. 3장은 대부분의 프로그래밍 언어들이 가지고 있는 변수, 데이터 타입, 함수를 소개하고 러스트에서의 사용법을 다룹니다. 4장에서는 다른 프로그래밍 언어와 차별화된 러스트의 특성인 소유권을 다룹니다. 5장에서는 구조체와 메소드 문법을 다루며 6장에서는 열거형에 대해 다룹니다.

보편적인 프로그래밍 개념

이번 챕터에서는 모든 프로그래밍 언어가 대부분 가진 개념이 Rust에서는 어떻게 다루어지는지 알아보고자 합니다. 많은 프로그래밍 언어가 보편적인 핵심요소를 갖습니다. 이번 챕터에서 Rust 고유의 개념은 다루지 않을테지만, 보편적인 프로그래밍 개념을 Rust의 문법을 설명하는 과정에서 토의하고자 합니다.

특히 변수, 기본 타입들, 함수, 주석, 그리고 제어문에 대해서 배울 수 있을 것입니다. 이 기본 사항들은 모든 Rust 프로그램에서 사용되며 이들을 조기에 숙지하는 것은 Rust를 시작하는데 큰 바탕이 되줄 겁니다.

Keywords

다른 언어들과 마찬가지로 Rust에도 고정된 의미를 갖는 *Keywords*가 있습니다. 이들은 변수나 함수명으로 사용될 수 없다는 점을 명심하세요. 대부분의 keywords가 특별한 의미를 갖고, 이를 통해 다양한 작업을 Rust를 통해 수행할 수 있습니다; 소수의 keywords는 현재는 아무 기능도 없지만 향후 추가될 기능을 위해 예약되어 있습니다. 이들은 목록은 Appendix A에서 찾아볼 수 있습니다.

변수와 가변성

2장에서 언급했듯이, 기본 변수는 불변성입니다. 이것은 Rust가 제공하는 안전성과 손쉬운 동시성이라는 장점을 취할 수 있도록 코드를 작성하게끔 강제하는 요소 중 하나입니다. 하지만 여전히 당신은 가변 변수를 사용하고 싶을 테죠. 어떻게 그리고 왜 Rust에서 불변성을 애호해주길 권장하는지 알아보면 그런 생각을 포기할 수 있을지도 모르겠습니다.

변수가 불변성인 경우, 일단 값이 이름에 bound되면 해당 값을 변경할 수 없습니다. 시험삼아 `cargo new --bin variables`을 실행해서 `projects` 디렉토리에 `variables`라는 새 프로젝트를 생성해 봅시다. 그런 다음 새 `variables` 디렉토리에서 `src/main.rs`를 열고 코드를 다음과 같이 바꿉니다.

Filename: `src/main.rs`

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

저장하고 `cargo run` 명령을 통해 실행시켜 봅시다. 당신은 다음과 같이 출력되는 에러를 확인하게 될 겁니다.

```
error[E0384]: re-assignment of immutable variable `x`
--> src/main.rs:4:5
   |
2 |     let x = 5;
3 |     - first assignment to `x`
4 |     println!("The value of x is: {}", x);
   |     ^^^^^^ re-assignment of immutable variable
```

위의 예제는 컴파일러가 당신이 만든 프로그램에서 당신을 도와 에러를 찾아주는 방법에 대해 보여주고 있습니다. 컴파일러 에러가 힘빠지게 만들 수도 있지만, 단지 당신의 프로그램이 아직 안전하게 수행되길 미흡하다는 뜻이지, 당신의 소양이 부족함을 의미하는 건 아닙니다. 숙련된 Rustacean들도 여전히 에러를 발생시키니까요. 에러가 나타내는 것은 `불변성 변수에 재할당`이고, 원인은 우리가 불변성 변수 `x`에 두 번째로 값을 할당했기 때문입니다.

우리가 이전에 불변성으로 선언한 것의 값을 변경하고자 하는 시도를 하면 컴파일 타임의 에러를 얻게 되고 이로 인해 버그가 발생할 수 있기 때문에 중요합니다. 만약 우리 코드의 일부는 값이 변경되지 않는다는 것을 가정하는데 다른 코드는 이와 다르게 값을 변경한다면, 전자에 해당하는 코드는 우리가 의도한 대로 수행되지 않을 수 있습니다. 특히 후자에 해당되는 코드가 항상 그렇지 않고 가끔 값을 변경하는 경우 나중에 버그의 원

인을 추적하기가 매우 어렵습니다.

Rust에서는 컴파일러가 변경되지 않은 값에 대한 보증을 해주고, 실제로 이는 바뀌지 않습니다. 이것이 의미하는 바는 당신이 코드를 작성하거나 분석할 시에 변수의 값이 어떻게 변경되는지 추적할 필요가 없기 때문에 코드를 더 합리적으로 만들어줍니다.

하지만 가변성은 매우 유용하게 사용될 수 있습니다. 변수는 기본적으로 불변성이지만 우리는 변수명의 접두어로 **mut**을 추가하는 것을 통해 가변성 변수를 선언할 수 있습니다. 이 변수의 값이 변경을 허용하는 것에 추가로 향후 코드를 보는 사람에게 코드의 다른 부분에서 해당 변수의 값을 변경할 것이라는 의도를 주지시킵니다.

예를 들어, *src/main.rs*를 다음과 같이 변경해보도록 합니다.

Filename: *src/main.rs*

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

위의 프로그램을 수행하면 다음과 같은 결과를 얻게 됩니다:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

mut을 사용하여, **x**에 bind된 값을 **5**에서 **6**으로 변경할 수 있습니다. 불변성 변수만을 사용하는 것보다 가변성 변수를 사용하여 보다 쉽게 구현할 수 있을 경우 가변성 변수를 만들어 사용할 수도 있습니다.

이런 의사 결정에 있어서 버그를 예방하는 것 외에도 고려해야 할 요소들이 있습니다. 예를 들어, 대규모 데이터 구조체를 다루는 경우 가변한 인스턴스를 사용하는 것이 새로 인스턴스를 할당하고 반환하는 것보다 빠를 수 있습니다. 데이터 규모가 작을수록 새 인스턴스를 생성하고 함수적 프로그래밍 스타일로 작성하는 것이 더 합리적이고, 그렇기에 약간의 성능 하락을 통해 가독성을 확보할 수 있다면 더 가치있는 선택입니다.

변수와 상수 간의 차이점들

변수의 값을 변경할 수 없다는 사항이 아마 당신에게 다른 언어가 가진 프로그래밍 개념을 떠오르게 하지 않

나요: 상수 불변성 변수와 마찬가지로 상수 또한 이름으로 bound된 후에는 값의 변경이 허용되지 않지만, 상수와 변수는 조금 다릅니다.

첫 째로, 상수에 대해서는 **mut**을 사용하는 것이 허용되지 않습니다: 상수는 기본 설정이 불변성인 것이 아니고 불변성 그 자체입니다.

우리가 상수를 사용하고자 하면 **let** 키워드 대신 **const** 키워드를 사용해야 하고, 값의 유형을 선언해야 합니다. 우리가 사용할 수 있는 유형들과 유형의 선언을 챕터 “Data Types,”에서 다루게 될 것이므로 자세한 사항은 지금 걱정하지 말고, 우리는 반드시 값의 유형을 선언해야 한다는 것을 알고 지나갑시다.

상수는 전체 영역을 포함하여 어떤 영역에서도 선언될 수 있습니다. 이는 코드의 많은 부분에서 사용될 필요가 있는 값을 다루는데 유용합니다.

마지막 차이점은 상수는 오직 상수 표현식만 설정될 수 있지, 함수 호출의 결과값이나 그 외에 실행 시간에 결정되는 값이 설정될 수는 없다는 점입니다.

아래의 **MAX_POINTS**라는 이름을 갖는 상수를 선언하는 예제에서는 값을 100,000으로 설정합니다. (Rust의 상수 명명 규칙에 따라 모든 단어를 대문자로 사용합니다.)

```
const MAX_POINTS: u32 = 100_000;
```

상수는 자신이 선언되어 있는 영역 내에서 프로그램이 실행되는 시간 동안 항상 유효하기에, 당신의 어플리케이션 도메인 전체에 걸쳐 프로그램의 다양한 곳에서 사용되는 값을 상수로 하면 유용합니다. 사용자가 한 게임에서 획득할 수 있는 최대 포인트, 빛의 속도 같은 값 등등...

당신의 프로그램 전체에 걸쳐 하드코드 해야 하는 값을 이름지어 상수로 사용하면 향후 코드를 유지보수하게 될 사람에게 그 의미를 전달할 수 있으므로 유용합니다. 또한 향후 해당 값을 변경해야 하는 경우에 상수로 선언된 값 한 곳만 변경하면 되므로 도움이 될 겁니다.

Shadowing

앞서 우리가 2장에서 추측 게임 예제를 통해 봤듯이, 이전에 선언한 변수와 같은 이름의 새 변수를 선언할 수 있고, 새 변수는 이전 변수를 *shadows*하게 됩니다. Rustaceans들은 이를 첫 변수가 두 번째에 의해 *shadowed* 됐다고 표현하게 됩니다. 해당 변수명은 두 번째 변수의 값을 갖게 된다는 뜻이죠. **let** 키워드를 사용해서 다음처럼 반복하여 같은 변수명으로 변수를 shadow 할 수 있습니다.

Filename: src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("The value of x is: {}", x);
}
```

이 프로그램은 처음 `x`에 값 `5`를 bind 합니다. 이후 반복된 `let x =` 구문으로 `x`를 shadow하고 원본 값에 `1`을 더해서 `x`의 값은 `6`이 됩니다. 세 번째 `let` 문으로 또 `x`를 shadow하고, 이전 값에 `2`를 곱하여 `x`의 최종값은 `12`가 됩니다. 이 프로그램을 실행하면 다음과 같은 결과를 볼 수 있습니다.

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/variables`
The value of x is: 12
```

이와 같은 사용은 변수를 `mut`으로 선언하는 것과는 차이가 있게 됩니다. 왜냐면 `let` 키워드를 사용하지 않고 변수에 새로 값을 대입하려고 하면 컴파일-시에 에러를 얻게 되기 때문이죠. 우리가 몇 번 값을 변경할 수는 있지만 그 이후에 변수는 불변성을 갖게 됩니다.

또 다른 `mut`과 shadowing의 차이는 `let` 키워드를 다시 사용하여 효과적으로 새 변수를 선언하고, 값의 유형을 변경할 수 있으면서도 동일 이름을 사용할 수 있다는 점입니다. 예를 들어, 공백 문자들을 입력받아 얼마나 많은 공백 문자가 있는지 보여주고자 할 때, 실제로는 저장하고자 하는 것은 공백의 개수일테죠.

```
let spaces = "    ";
let spaces = spaces.len();
```

이와 같은 구조가 허용되는 이유는 첫 `spaces` 변수가 문자열 유형이고 두 번째 `spaces` 변수는 첫 번째 것과 동일한 이름을 가진 새롭게 정의된 숫자 유형의 변수이기 때문입니다. Shadowing은 `space_str`이나 `space_num`과 같이 대체된 이름을 사용하는 대신 간단히 `spaces` 이름을 사용할 수 있게 해줍니다. 그러나 우리가 `mut`을 사용하려고 했다면:

```
let mut spaces = "    ";
spaces = spaces.len();
```

우리는 다음처럼 변수의 유형을 변경할 수 없다는 컴파일-시의 에러를 얻게 될 겁니다:

```
error[E0308]: mismatched types
--> src/main.rs:3:14
3 |     spaces = spaces.len();
   |          ^^^^^^^^^^ expected &str, found usize
   |
   = note: expected type `&str`
           found type `usize`
```

변수가 어떻게 동작하는지 탐구했으니, 더 많은 데이터 유형을 살펴보도록 합시다.

데이터 타입들

Rust에서 사용되는 모든 값들은 어떤 *타입*을 갖습니다. 그러니 어떤 형태의 데이터인지 명시하여 Rust에게 알려줘서 이를 통해 데이터를 어떻게 다룰지 알 수 있도록 해야 합니다. 이번 장에서, 우리는 언어에 포함되어 있는 여러 타입들을 살펴보고자 합니다. 타입은 크게 스칼라와 컴파운드, 둘로 나눌 수 있습니다.

이번 장의 전체에 걸쳐 주지해야 할 점은 Rust는 *타입이 고정된 언어*라는 점입니다. 이게 의미하는 바는 모든 변수의 타입이 컴파일 시에 반드시 정해져 있어야 한다는 겁니다. 보통 컴파일러는 우리가 값을 사용하는지에 따라 타입을 추측할 수 있습니다. 2장에서 `String`을 `parse`를 사용하여 숫자로 변환했던 경우처럼 타입의 선택 폭이 넓은 경우는 반드시 타입의 명시를 첨가해야 합니다. 다음처럼:

```
let guess: u32 = "42".parse().expect("Not a number!");
```

여기서 타입 명시를 첨가하지 않은 경우, Rust는 다음과 같은 에러를 발생시킵니다.

이와 같은 에러는 컴파일러가 우리에게 사용하고 싶은 타입이 무엇인지 추가적인 정보를 요구하는 겁니다.

```
error[E0282]: type annotations needed
--> src/main.rs:2:9
2 |     let guess = "42".parse().expect("Not a number!");
   |     ^^^^^^
   |     cannot infer type for `guess`
   |     consider giving `guess` a type
```

우리가 다루고자 하는 다양한 데이터 타입들 각각의 타입 명시를 살펴보겠습니다.

스칼라 타입들

*스칼라*는 하나의 값으로 표현되는 타입입니다. Rust는 정수형, 부동소수점 숫자, boolean, 그리고 문자, 네 가지 스칼라 타입을 보유하고 있습니다. 아마 다른 프로그래밍 언어에서도 본 적이 있겠지만, Rust에서 이들이 어떻게 동작하는지 살펴보도록 합시다.

정수형

정수형은 소수점이 없는 숫자입니다. 우리는 이번 장의 앞부분에서 `u32` 타입인 정수형을 사용했습니다. 해당 타입의 선언은 부호 없는 32비트 변수임을 나타냅니다 (부호 있는 타입은 `u` 대신 `i`로 시작합니다.) 표 3-1은 Rust에서 사용되는 정수형들을 보여줍니다. 부호, 미부호로 나뉜 다른 열의 타입을 사용하여(`i16`처럼) 정수 값의 타입을 선언할 수 있습니다.

Table 3-1: Rust에서의 정수 타입

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
arch	isize	usize

각각의 타입은 부호 혹은 미부호이며 명시된 크기를 갖습니다. 부호 혹은 미부호의 의미는, 숫자가 양수 혹은 음수를 다룰 수 있는지 혹은 없는지를 나타냅니다. 다르게 말하면, 숫자가 부호를 가져야 하는 경우(부호) 혹은 오직 양수만을 가질 것이기에 부호가 없이도 표현할 수 있는가(미부호)를 나타냅니다. 종이에 숫자 기재하는 것과 같죠: 부호와 함께 다뤄야 하는 경우에 숫자는 더하기 혹은 빼기 기호와 함께 표시하죠. 숫자가 양수라고 가정해도 문제 없는 상황에는 부호 없이 표시하게 됩니다. 부호된 숫자는 2의 보수 형태를 사용하여 저장됩니다. (2의 보수가 모른다면 검색해보세요. 이 책에서 다루는 내용이 아닙니다.)

각 부호 변수는 $-(2^{n-1})$ 부터 $2^n - 1$ 까지의 값을 포괄합니다. 여기서 n 은 사용되는 타입의 비트 수입니다. 즉, **i8**은 $-(2^7)$ 에서 $2^7 - 1$ 까지의 값, 즉 -128에서 127 사이의 값을 저장할 수 있습니다. 미부호 타입은 0에서 $2^n - 1$ 까지의 값을 저장할 수 있습니다. 즉, **u8** 타입은 0에서 $2^8 - 1$ 다시 말해, 0에서 255 까지의 값을 저장할 수 있습니다.

추가로, **isize**와 **usize** 타입은 당신의 프로그램이 동작하는 컴퓨터 환경이 64-bits인지 아닌지에 따라 결정됩니다. 64-bit 아키텍처이면 64bit를, 32-bit 아키텍처이면 32bit를 갖게 됩니다.

당신은 테이블 3-2에서 보여주는 형태들처럼 정수형 리터럴을 사용할 수 있습니다. byte 리터럴을 제외하고 모든 정수형 리터럴은 **57u8**과 같은 타입 접미사와 **1_000**과 같이 시각적인 구분을 위한 **_**의 사용을 허용합니다.

Table 3-2: Rust의 정수형 리터럴들

Number literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

그렇다면 어떤 타입의 정수를 사용해야 할까요? 확실하게 정해진 경우가 아니면 Rust의 기본 값인 **i32**가 일반적으로는 좋은 선택입니다. 이는 일반적으로 가장 빠르기 때문이죠. 심지어 64-bit 시스템에서도요.

`isize`나 `usize`는 주로 일부 컬렉션 타입의 색인에 사용됩니다.

부동 소수점 타입

Rust에는 소수점을 갖는 숫자인 **부동소수점 숫자**를 위한 두 가지 기본 타입도 있습니다. Rust의 부동소수점 타입은 `f32`와 `f64`로, 예상하신 대로 각기 32bit와 64bit의 크기를 갖습니다. 기본 타입은 `f64`인데, 그 이유는 최신의 CPU 상에서는 `f64`가 `f32`와 대략 비슷한 속도를 내면서도 더 정밀한 표현이 가능하기 때문입니다.

다음은 부동소수점 숫자가 활용되는 예제입니다:

Filename: src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

부동소수점 숫자는 IEEE-754 표준에 따라 표현됩니다. `f32` 타입은 1배수의 정밀도인 부동소수점이고, `f64`는 2배수의 정밀도인 부동소수점입니다.

수학적 연산들.

Rust가 지원하는 일반적인 기본 수학적 연산은 기대하신 것처럼 모든 숫자 타입에 적용됩니다: 더하기, 빼기, 곱하기, 나누기 등등. 다음의 코드로 보여주려는 것은 각 경우를 `let` 문 내에서 사용할 수 있는 방법입니다.

Filename: src/main.rs

```
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;

    // remainder
    let remainder = 43 % 5;
}
```

위의 문장에서 각 표현식들은 수학 연산자를 사용하여 산출된 값을 변수로 bound 합니다. 부록 B에 Rust에서 제공하는 모든 연산자 목록이 들어있습니다.

Boolean 타입

대부분의 다른 언어들처럼, boolean 타입은 Rust에서 둘 중 하나의 값만 가질 수 있습니다: `true` 와 `false`. boolean 타입은 러스트에서 `bool`로 명시됩니다.

예제:

Filename: src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

boolean 값을 사용하는 주된 방법은 `if`문과 같은 조건문에서 조건으로 사용하는 것입니다. 우리는 `if`문이 Rust에서 동작하는 방식을 “제어 흐름” 장에서 다루게 될 겁니다.

문자 타입

지금까지 숫자 타입만을 살펴봤는데, Rust는 문자 또한 지원합니다. Rust의 `char`는 이 언어의 가장 근본적인 알파벳 타입이고, 다음의 코드는 이를 사용하는 한 가지 방법입니다. 스트링이 큰따옴표를 쓰는 것에 반하여 `char` 타입은 작은따옴표로 쓰는 점을 주목하세요:

Filename: src/main.rs

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let heart_eyed_cat = '😻';
}
```

Rust의 `char` 타입은 Unicode Scalar를 표현하는 값이고 이는 ASCII 보다 많은 표현을 가능하게 합니다. 억양 표시가 있는 문자, 한국어/중국어/일본어 표의 문자, 이모티콘, 넓이가 0인 공백문자 모두가 Rust에서는 `char` 타입으로 사용할 수 있습니다. Unicode Scalar 값의 범위는 `U+0000`에서 `U+D7FF` 그리고 `U+E000`에서 `U+10FFFF`를 포함합니다. 그럼에도 불구하고 “문자”는 Unicode을 위한 개념이 아니기 때문에, 당신의 인간적 직관에 따른 “문자”와 Rust의 `char` 가 동일하지 않을 수 있습니다. 우리는 8장 “Strings” 부에서 이 주제에 대해 상세히 다루게 될 겁니다.

복합 타입들

복합 타입들은 다른 타입의 다양한 값을 하나의 타입으로 묶을 수 있습니다. Rust는 두 개의 기본 타입들을 갖고 있습니다: 튜플과 배열.

값들을 집합시켜서 튜플화하기.

튜플은 다양한 타입의 몇 개의 숫자를 집합시켜 하나의 복합 타입으로 만드는 일반적인 방법입니다.

우리는 괄호 안에 콤마로 구분되는 값들의 목록을 작성하여 튜플을 만듭니다. 튜플에 포함되는 각 값의 타입이 동일할 필요없이 서로 달라도 됩니다. 다음의 예제에 우리는 선택 사항인 타입 명시를 추가했습니다.

Filename: src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

튜플은 단일 요소를 위한 복합계로 고려되었기에 변수 `tup`에는 튜플 전체가 bind 됩니다. 개별 값을 튜플의 밖으로 빼내오기 위해서는, 패턴 매칭을 사용하여 튜플의 값을 구조해체 시키면 됩니다. 다음을 봅시다:

Filename: src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

해당 프로그램은 처음에 튜플을 만들고 변수 `tup`에 bind 시킵니다. 이후 패턴과 `let`을 통해 `tup`을 세개의 분리된 변수 `x`, `y`, 그리고 `z`에 이동시킵니다. 이것을 구조해체라고 부르는 이유는 하나의 튜플을 세 부분으로 나누기 때문입니다. 최종적으로 프로그램은 `y`의 값을 출력할 것이고 이는 `6.4`입니다.

패턴 매칭을 통한 구조해체에 추가로, 우리는 마침표(`.`) 뒤에 우리가 접근하길 원하는 값의 색인을 넣는 것을 통해 튜플의 요소에 직접적으로 접근할 수 있습니다. 예제를 봅시다:

Filename: src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

위의 프로그램은 튜플 `x`를 만들고, 이의 각 요소들을 그들의 색인을 통해 접근하여 새 변수를 만듭니다. 대부분의 언어가 그렇듯이, 튜플의 첫 번째 색인은 0입니다.

배열

여러 값들의 집합체를 만드는 다른 방법은 **배열**입니다. 튜플과는 다르게, 배열의 모든 요소는 모두 같은 타입이여야 합니다. Rust의 배열이 몇 다른 언어들의 배열과 다른 점은 Rust에서는 배열은 고정된 길이를 갖는다는 점입니다: 한번 선언되면, 이들은 크기는 커지거나 작아지지 않습니다.

Rust에서는 대괄호 안에 값을 콤마로 구분하여 나열해서 배열을 만듭니다:

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

배열이 유용할 때는 당신의 데이터를 heap보다 stack에 할당하는 것을 원하거나(stack과 heap에 대해서는 4장에서 다루게 될 것입니다), 당신이 항상 고정된 숫자의 요소를 갖는다고 확신하고 싶을 때입니다. 이들은 벡터 타입처럼 가변적이지 않습니다. 벡터 타입은 유사 집합체로 표준 라이브러리에서 제공되며 확장 혹은 축소가 가능합니다. 배열이나 벡터 중에 뭘 선택해야 할지 확실하지 않은 상황이라면 벡터를 사용하도록 하세요. 8장에서 벡터에 대해 더 자세히 다룹니다.

벡터가 아닌 배열을 선택하게 되는 경우의 예로, 프로그램이 올해의 달 이름을 알고자 할 경우입니다. 프로그램이 달을 추가하거나 삭제하는 경우는 거의 없을 것이므로, 고정적으로 12개의 아이템을 가질테니 배열을 사용하면 됩니다.

```
let months = ["January", "February", "March", "April", "May", "June", "July",
              "August", "September", "October", "November", "December"];
```

배열 요소에 접근하기

배열은 stack에 단일 메모리 뭉치로 할당됩니다. 우리는 색인을 통해 배열의 요소에 접근할 수 있습니다. 이렇게요:

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

이번 예제에서, `first`로 명명된 변수는 값 1이 될텐데, 왜냐면 배열 색인 [0]에 들어있는 값이기 때문이죠. `second`로 명명된 변수는 배열의 색인 [1]의 값인 2가 되겠죠.

유효하지 않은 배열 요소에 대한 접근

만약 우리가 배열의 끝을 넘어선 요소에 접근하려고 하면 어떻게 될까요? 예제를 다음처럼 변경해봤습니다.

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

    println!("The value of element is: {}", element);
}
```

이번 코드를 `cargo run`을 통해 동작시키면 다음의 결과를 얻게 됩니다:

```
$ cargo run
Compiling arrays v0.1.0 (file:///projects/arrays)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/arrays`
thread '' panicked at 'index out of bounds: the len is 5 but the index
is
10', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

컴파일 시에는 아무런 에러도 발생시키지 않습니다만, 프로그램의 결과는 실행 중에 에러가 발생했고 성공적으로 종료되지 못했다고 나옵니다.

색인을 사용하여 요소에 접근하려고 하면 Rust는 지정한 색인이 배열 길이보다 작은지 확인합니다. 색인이 길이보다 길면 Rust는 프로그램이 오류와 함께 종료 될 때 Rust가 사용하는 용어인 *패닉(panic)*합니다.

이것은 Rust의 안전 원칙이 동작하는 첫 번째 예입니다. 많은 저수준 언어에서 이러한 타입의 검사는 수행되지 않으며 잘못된 색인을 제공하면 유효하지 않은 메모리에 액세스 할 수 있습니다. Rust는 메모리 접근을 허용하고 계속 진행하는 대신 즉시 종료하여 이러한 종류의 오류로부터 사용자를 보호합니다. 9 장에서는 Rust의 오류 처리에 대해 자세히 설명합니다.

함수 동작 원리

함수는 Rust에 녹아들어 있습니다. 여러분은 이미 언어에서 가장 중요하게 생각하는 `main` 함수를 보셨습니다. 이는 다수의 프로그램에서 실행 지점입니다. 여러분은 또한 `fn` 키워드도 보셨을텐데, 이는 새로운 함수의 선언을 가능하게 합니다.

Rust 코드는 **뱀 형태**를 변수나 함수 이름의 형식 규칙으로 사용합니다. 뱀 형태에서, 모든 문자는 소문자를 사용하며 밑줄 표시로 단어를 구분합니다. 다음은 예제로 함수를 선언하는 프로그램입니다:

Filename: src/main.rs

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Rust에서의 함수 선언은 `fn`으로 시작하며 함수 이름 뒤에 괄호의 형식으로 되어 있습니다. 중괄호는 컴파일러에게 함수의 시작과 종료 지점을 알려주게 됩니다.

우리는 함수의 이름과 괄호 형식을 기입하는 것을 통해 우리가 선언했던 어떤 함수든 호출할 수 있습니다. `another_function`이 프로그램 내에 정의되어 있으므로, `main` 함수에서 해당 함수를 호출할 수 있습니다. 주의할 점은, 소스 코드 내에서 `another_function`이 `main` 함수 뒤에 정의했다는 점입니다. 우리는 이를 `main` 함수 앞에도 정의할 수 있습니다. Rust는 당신의 함수의 위치를 신경쓰지 않습니다, 어디든 정의만 되어 있으면 됩니다.

함수를 추가로 탐색하기 위해 *functions*이라는 이름의 새로운 바이너리 프로젝트를 시작합시다.

`another_function` 예제를 `src/main.rs`에 넣고 실행해보세요. 다음과 같은 결과가 나타납니다:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 0.28 secs
Running `target/debug/functions`
Hello, world!
Another function.
```

`main` 함수 안의 내용이 줄의 순서대로 수행됩니다. 처음으로, "Hello, world!" 메시지가 출력되고, `another_function`이 호출되고 그의 메시지를 출력합니다.

함수 매개변수

함수는 함수 고유한 부분인 특별한 변수 **매개변수**를 갖는 형식으로 선언될 수 있습니다. 함수가 매개변수를 취할 때, 우리는 상수를 그들의 전달인자로 제공할 수 있습니다. 기술적으로, 여기서 전달되는 상수를 **전달인자**라고 부릅니다만, 사람들은 보통 “전달인자”와 “매개변수”를 혼용해서 사용하는 경향이 있습니다.

다음의 재작성 된 `another_function`은 Rust에서 매개변수가 어떤 것인지 보여줍니다:

Filename: src/main.rs

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {}", x);
}
```

이 프로그램을 실행해보시면 다음과 같은 결과가 출력되는 것을 보게 될 겁니다:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 1.21 secs
Running `target/debug/functions`
The value of x is: 5
```

`another_function`의 선언은 `x`로 명명된 하나의 매개변수를 갖습니다. `x`의 타입은 `i32`로 정의됩니다. `5`가 `another_function`으로 전달되면, `println!` 매크로는 중괄호 짹으로 된 형식 문자열에 `5`를 전달합니다. 함수의 선언부에서, 여러분은 반드시 각 매개변수의 타입을 정의해야 합니다. 이 사항은 Rust를 설계하며 내린 신중한 결정사항입니다: 함수의 정의에 타입을 명시하여 코드 내 다른 부분에서 이들을 사용하는 것을 통해 당신의 의도를 추측하지 않아도 되게 됩니다.

여러분의 함수에 여러 개의 매개변수를 사용하고 싶으면, 매개변수들을 다음처럼 쉼표와 함께 구분해서 사용할 수 있습니다:

Filename: src/main.rs

```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

이 예제는 각각 `i32` 타입인 두 개의 매개변수를 갖는 함수를 생성합니다. 함수는 그의 두 매개변수의 값을 출력합니다. 주의할 점은, 함수 매개변수는 이번 예제처럼 굳이 같은 타입이 아니여도 된다는 점입니다. 한번 코드를 실행해봅시다. 여러분의 *function* 프로젝트의 `src/main.rs` 내용을 위의 예제로 변경한 뒤에, `cargo run`을 통해 수행시키면 됩니다:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/functions`
The value of x is: 5
The value of y is: 6
```

우리는 값 `5`와 `6`을 `x`와 `y`로 전달했기 때문에, 이 값들이 담긴 두 문장을 출력합니다.

함수 본문

함수 본문은 필요에 따라 표현식으로 종결되는 구문의 나열로 구성됩니다. 지금까지 우리는 종결 표현식이 없는 함수만 다뤘기에, 표현식이 구문의 일부처럼 여겨질지 모르겠습니다. Rust가 표현식에 기반한 언어기 때문에, 이것은 이해하셔야 하는 중요한 차이점입니다. 다른 언어들은 이와 같은 차이가 없으니, 구문과 표현식이 함수의 본문에 어떤 식으로 차이나게 적용되는지 살펴보도록 하겠습니다.

구문과 표현식

사실 우리는 이미 구문과 표현식을 사용했습니다. 구문은 어떤 명령들의 나열로 값을 반환하지 않는 어떤 동작을 수행 합니다. 표현식은 결과 값을 산출해냅니다. 다음 몇 개의 예제를 살펴보도록 합시다. `let` 키워드를 통해 변수를 만들고 값을 할당하는 구문을 만듭니다. 항목 3-3의, `let y = 6;`은 구문입니다:

Filename: `src/main.rs`

```
fn main() {
    let y = 6;
}
```

항목 3-3: 하나의 구문을 갖는 `main` 함수를 선언하였다.

함수 정의는 또 하나의 구문입니다; 상기 예제는 자신 그 자체가 구문입니다. 구문은 값을 반환하지 않습니다. 그러나, 여러분은 다음처럼 `let` 구문을 사용해서는 다른 변수에 값을 대입할 수 없습니다:

Filename: src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

여러분이 이 프로그램을 수행하면, 다음과 같은 에러를 보게 될 겁니다:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
--> src/main.rs:2:14
2 |     let x = (let y = 6);
   |           ^^^
   |
= note: variable declaration using `let` is a statement
```

`let y = 6` 구문은 반환 값이 없으므로, `x`에 bind 시킬 것이 없습니다. 이것이 다른 언어인 C나 Ruby와의 차이점입니다. 이들 언어들은 `x = y = 6`과 같은 코드가 `x`와 `y`에 모두 `6`의 값을 대입할 수 있습니다; Rust에서는 허용되지 않습니다. 여러분이 작성하는 Rust 코드의 대부분은 표현식이며 이는 어떤 값을 산출합니다. `5 + 6`과 같은 간단한 수학 연산을 살펴보면, 이는 `11`이란 값을 산출하는 표현식입니다.

표현식은 구문의 부분일 수 있습니다: 항목 3-3은 `let y = 6;`이란 구문을 갖는데, `6`은 `6`이란 값을 산출하는 표현식입니다. 함수를 호출하는 것은 표현식입니다. 매크로를 호출하는 것은 표현식입니다. 예제처럼 새로운 범위를 생성하는데 사용하는 block `{}`은 표현식입니다:

Filename: src/main.rs

```
fn main() {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {}", y);
}
```

표현식 부:

```
{
    let x = 3;
    x + 1
}
```

이번 경우에 해당 block은 4를 산출합니다. 이 값은 `let` 구문의 일부로 `y`에 bound됩니다. 여러분이 앞서 봤던 것과 다르게 `x + 1` 줄의 마지막이 세미콜론으로 끝나지 않은 점을 주목하세요. 표현식은 종결을 나타내는 세미콜론을 사용하지 않습니다. 만약 세미콜론을 표현식 마지막에 추가하면, 이는 구문으로 변경되고 반환 값이 아니게 됩니다. 이후부터 함수의 반환 값과 표현식을 살펴보실 때 이 점을 유의하세요.

반환 값을 갖는 함수

함수는 그들을 호출한 코드에 값을 반환할 수 있습니다. 우리는 반환되는 값을 명명해야 할 필요는 없지만, 그들의 타입은 화살표(`->`) 뒤에 선언해야 합니다. Rust에서 반환 값은 함수 본문의 마지막 표현식의 값과 동일합니다. `return` 키워드와 값을 써서 함수로부터 일찍 반환할 수 있지만, 대부분의 함수들은 암묵적으로 마지막 표현식을 반환합니다. 값을 반환하는 함수의 예를 보겠습니다:

Filename: src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {}", x);
}
```

`five` 함수에는 함수 호출, 매크로, 심지어 `let` 구문도 없이 그저 `5`란 숫자 하나가 있습니다. 이는 Rust에서 완벽하게 함수로 허용됩니다. 함수 반환 값의 타입이 `-> i32`로 명시되어 있다는 점 또한 주목하세요. 해당 코드를 수행하면 다음과 같은 결과를 얻게 될 겁니다:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/functions`
The value of x is: 5
```

`5`은 `five` 함수가 반환한 값이고, 이 때문에 반환 타입을 `i32`으로 한 것이지요. 좀 더 자세히 설명해보겠습니다. 중요한 지점이 두 곳 있습니다: 첫째, `let x = five();` 줄은 우리가 반환 값을 변수의 초기 값으로 사용하는 것을 보여줍니다. `five`의 반환 값이 `5`이기 때문에, 해당 줄은 다음과 동일합니다:

```
let x = 5;
```

둘째, `five` 함수는 매개변수 없이 반환 값에 대한 타입만 정의되어 있지만, 본문에는 `5`만이 세미콜론 없이 외로이 있는 이유는 이것이 우리가 값을 반환하고자 할 때 사용하는 표현식이기 때문입니다. 다른 예제를 통해 살펴보겠습니다:

Filename: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

이 코드를 수행하면 `The value of x is: 6`를 출력하게 됩니다. 우리가 `x + 1` 끝에 세미콜론을 추가하여 표현식을 구문으로 변경하면 어떤 일이 일어날까요?

Filename: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

이 코드를 실행하면 다음과 같은 에러를 얻게 됩니다:

```
error[E0308]: mismatched types
--> src/main.rs:7:28
|
7 |     fn plus_one(x: i32) -> i32 {
|     |-----^
8 |     |     x + 1;
|     |             - help: consider removing this semicolon
9 |     |
|     |     ^ expected i32, found ()
|
= note: expected type `i32`
          found type `()`
```

에러 메시지의 중요 포인트는 “mismatched types,”으로 이 코드의 주요 문제를 보여줍니다. `plus_one` 함수의 정의는 `i32` 값을 반환하겠다고 하였으나, 구문은 값을 산출하지 않기에 `()`처럼 비어있는 튜플로 표현됩니다. 이런 이유로, 반환할 것이 없어서 함수가 정의된 내용과 상충하게 되고 이는 에러를 발생시킵니다. 이번 결과에서는, Rust가 문제를 해결할 수 있도록 도와주는 메시지를 제공합니다: 세미콜론을 제거하면 에러가 교정될 수도 있다고 제안하네요.

주석

모든 프로그래머들은 되도록 이해하기 쉽게 이해되는 코드를 작성하기 위해 노력하지만, 자주 부연 설명이 필요합니다. 이런 경우, 프로그래머들은 메모를 남기거나 소스코드에 컴파일러는 무시하도록 되어 있는 주석을 남겨 소스코드를 읽는 사람이 혜택을 받을 수 있게 합니다.

여기에 간단한 주석이 있습니다:

```
// Hello, world.
```

Rust에서 주석은 두개의 슬래쉬로 시작해야 하고 해당 줄의 끝까지 계속됩니다. 한 줄을 넘는 주석을 작성할 경우, `//`를 각 줄에 포함시켜 사용하면 됩니다, 이런 식으로요:

```
// 우리는 여기에 뭔가 복잡한 것을 적어놓고자 하는데, 그를 위해 충분히 긴 여러 줄의 주석이 필요합니다.  
// 휴! 다행입니다.  
// 이 주석은 그에 대해 설명할테니까요.
```

주석은 코드의 뒷 부분에 위치할 수도 있습니다:

Filename: src/main.rs

```
fn main() {  
    let lucky_number = 7; // I'm feeling lucky today.  
}
```

하지만 주석을 코드와 나눠 앞 줄에 기재되는 형식을 더 자주 보게 될 겁니다.

Filename: src/main.rs

```
fn main() {  
    // I'm feeling lucky today.  
    let lucky_number = 7;  
}
```

러스트는 다른 종류의 주석인 문서화 주석(documentation comments)을 가지고 있는데, 이는 14장에서 논의할 것입니다.

제어문

조건의 상태가 참인지에 따라 어떤 코드의 실행 여부를 결정하거나 조건이 만족되는 동안 반복 수행을 하는 것은 대부분의 프로그래밍 언어의 기초 문법입니다. 우리가 실행 흐름을 제어할 수 있는 가장 보편적인 작성 방식은 **if** 표현식과 반복문입니다.

if 표현식

if 표현식은 우리의 코드가 조건에 따라 분기할 수 있게 합니다. 우리가 조건을 제공하는 것은 다음 서술과 같죠. “만약 조건이 충족되면, 이 코드 블럭을 실행하세요. 만약 충족되지 않았다면 코드 블럭을 실행하지 마세요.”

*branches*로 명명된 새 프로젝트를 우리의 *projects* 디렉토리에 생성하고 **if** 식을 탐구합시다. *src/main.rs* 파일에 다음의 내용을 기입하세요:

Filename: *src/main.rs*

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

모든 **if** 표현식은 **if**란 키워드로 시작하며 뒤이어 조건이 옵니다. 이번 경우에 조건은 변수 **number**가 5보다 작은 값을 가지는지 여부가 됩니다. 조건이 참일 때 실행하는 코드 블록은 조건 바로 뒤 중괄호로 된 블록에 배치됩니다. **if** 식의 조건과 관련된 코드 블럭은 우리가 2장의 “비밀번호 추리 게임”에서 다뤘던 **match** 식의 갈래(arms)와 마찬가지로 갈래(arms)로 불립니다. 선택적으로, 우리는 이번 경우에서 처럼 **else** 식을 포함시킬 수 있는데, 이는 조건이 거짓으로 산출될 경우 실행시킬 코드 블럭을 프로그램에 제공합니다. 당신이 **else** 식을 제공하지 않는데 조건이 거짓이 되면, 프로그램은 **if** 블록을 생략하고 다음 순서의 코드를 실행하게 될 겁니다.

이 코드를 실행해보세요; 다음과 같은 결과를 얻을 수 있을 겁니다:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/branches`
condition was true
```

`number`의 값을 조건을 `거짓`으로 만들 값으로 변경하면 무슨 일이 일어날지 살펴보도록 합시다:

```
let number = 7;
```

프로그램을 다시 실행시키면, 다음과 같은 결과를 보게 됩니다:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/branches`
condition was false
```

주의해야 할 중요한 점은 이번 코드의 조건은 반드시 `bool`이어야 합니다. 만약 `bool`이 아닐 경우 어떤 일이 일어나는지는 다음의 코드를 실행하면 알 수 있을 겁니다:

Filename: src/main.rs

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

`if`의 조건이 `3`으로 산출되고, Rust는 에러를 발생시킵니다.

```
error[E0308]: mismatched types
--> src/main.rs:4:8
  |
4 |     if number {
  |         ^^^^^^ expected bool, found integral variable
  |
= note: expected type `bool`
          found type `<integer>`
```

이 에러가 나타내는 것은 Rust가 `bool`을 기대하였으나 정수형이 왔다는 겁니다. Rust는 boolean 타입이 아닌 것을 boolean 타입으로 자동 변환하지 않습니다. Ruby나 Javascript와는 다르죠. 우리는 반드시 명

시적으로 `boolean`을 `if`의 조건으로 사용해야 합니다. 만약 우리가 `if` 표현식의 코드 블록을 숫자가 0이 아닐 시에 실행하고 싶다면, 다음처럼, 우리는 `if` 표현식을 변경할 수 있습니다.

Filename: src/main.rs

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

이번 코드를 실행시키면 `number was something other than zero`가 출력될 겁니다.

else if와 다수 조건

우리는 `if`와 `else` 사이에 `else if`식을 추가 결합하여 다양한 조건을 다룰 수 있습니다. 예제를 보시죠:

Filename: src/main.rs

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

이번 프로그램은 분기할 수 있는 네 개의 경로를 갖습니다. 이를 수행하면, 다음과 같은 결과를 얻게 될 겁니다:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/branches`
number is divisible by 3
```

이 프로그램이 실행될 때, `if` 식을 차례대로 검사하고 검사 조건이 참일 때의 첫 번째 본문을 실행합니다. 주 목할 점은 6은 2로 나누어 떨어짐에도 불구하고 `number is divisible by 2` 이 출력되지 않는데, `else`의 블럭에 위치한 `number is not divisible by 4, 3, or 2` 도 마찬가지입니다. 이렇게 되는 이유는 Rust가 첫 번째로 조건이 참이 되는 블록만 찾아 실행하고, 한번 찾게 되면 나머지는 검사하지 않기 때문입니다.

너무 많은 `else if` 식의 사용은 당신의 코드를 이해하기 어렵게 하므로, 둘 이상일 경우 코드를 리팩토링하게 될 수도 있습니다. 이런 경우를 위해 6장에서 `match` 라 불리는 강력한 분기 생성자를 다룹니다.

let 구문에서 if 사용하기

`if` 가 표현식이기 때문에, 항목 3-4에서처럼, 우리는 이를 `let` 구문의 우측에 사용할 수 있죠.

Filename: src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    println!("The value of number is: {}", number);
}
```

Listing 3-4: `if` 표현식의 결과값을 변수에 대입하기

변수 `number`에는 `if` 식에서 산출된 값이 bound되게 됩니다. 어떤 일이 일어날지 코드를 실행해보죠:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/branches`
The value of number is: 5
```

기억하세요! 코드 블록은 그들의 마지막에 위치한 표현식을 산출하며 숫자는 그 자체로 표현식이라는 것을요. 이 경우 전체 `if` 식의 값은 실행되는 코드 블럭에 따라 다릅니다. 그렇기에 `if` 식에 속한 각 갈래의 결과는 반드시 같은 타입이여야 합니다. 항목 3-4에서 `if` 갈래와 `else` 갈래는 모두 `i32` 정수형을 결과 값으로 가져집니다. 하지만 만약 다음 예제처럼 유형이 다르면 어떻게 될까요?

Filename: src/main.rs

```
fn main() {
    let condition = true;

    let number = if condition {
        5
    } else {
        "six"
};

    println!("The value of number is: {}", number);
}
```

우리가 이번 코드를 실행시키려고 하면 에러를 얻게 됩니다. `if`와 `else` 간의 값 타입이 호환되지 않고, Rust는 정확히 프로그램의 어느 지점에 문제가 있는지 보여줍니다.

```
error[E0308]: if and else have incompatible types
--> src/main.rs:4:18
|
4 |     let number = if condition {
|     |             ^-----^
5 |     |         5
6 |     |     } else {
7 |     |         "six"
8 |     |     };
|     |____^ expected integral variable, found reference
|
= note: expected type `<integer>`
       found type `&str`
```

`if` 블록이 정수형을 산출하는 식이고 `else` 블록은 문자열을 산출하는 식입니다. 이런 경우가 성립하지 않는 이유는 변수가 가질 수 있는 타입이 오직 하나이기 때문입니다. Rust는 컴파일 시에 `number` 변수의 타입이 뭔지 확실히! 정의해야 합니다. 그래야 `number`가 사용되는 모든 곳에서 유효한지 검증할 수 있으니까요. Rust는 `number`의 타입을 실행 시에 정의되도록 할 수 없습니다. 컴파일러가 모든 변수의 다양한 타입을 추적해서 알아내야 한다면 컴파일러는 보다 복잡해지고 보증할 수 있는 것은 적어지게 됩니다.

반복문과 반복

코드 블록을 한 번 이상 수행하는 것은 자주 유용합니다. 반복 작업을 위해서, Rust는 몇 가지 반복문을 제공합니다. 반복문은 반복문 시작부터 끝까지 수행하고 다시 처음부터 수행합니다. 반복문을 실험해보기 위해 `loops`으로 명명된 새 프로젝트를 작성해 봅시다.

Rust가 제공하는 세 가지 반복문: `loop`, `while`, 그리고 `for`을 모두 사용해 봅시다.

loop와 함께 코드의 반복 수행

`loop` keyword는 Rust에게 그만두라고 명시하여 알려주기 전까지 코드 블럭을 반복 수행합니다. 예제로, 우리의 `loops`디렉토리에 `src/main.rs`를 다음처럼 변경하세요:

Filename: `src/main.rs`

```
fn main() {
    loop {
        println!("again!");
    }
}
```

이 프로그램을 실행시키면, 우리는 프로그램을 강제 정지하기 전까지 `again!`이 반복 출력되는 것을 보게 됩니다. 대부분의 터미널은 단축키 `ctrl-C`를 통해서 무한루프에 빠진 프로그램을 정지시키는 기능을 지원합니다. 한번 시도해 보세요:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Running `target/debug/loops`
again!
again!
again!
again!
again!
^Cagain!
```

기호 `^C`는 우리가 `ctrl-C`를 눌렀을 때의 위치입니다. 코드가 정지 신호를 받은 시점에 따라 `^C` 이후에 `again!`이 출력될 수도 아닐 수도 있습니다.

다행스럽게도, Rust는 보다 안정적으로 루프에서 벗어날 수 있는 방법을 제공합니다. 우리는 `break` keyword를 위치시켜 프로그램이 언제 루프를 멈춰야 하는지 알려줄 수 있습니다. 상기시켜 드리자면 2장 “추리 게임”에서 사용자가 모든 숫자를 정확히 추리했을 경우 프로그램을 종료시키기 위해 사용했습니다.

while과 함께하는 조건부 반복

반복문 내에서 조건을 산출하는 것은 자주 유용합니다. 조건이 참인 동안 반복문을 수행합니다. 조건이 참이 아니게 된 경우에 `break`을 호출하여 반복을 정지시킵니다. 이런 패턴의 반복문을 구현하자면 `loop`, `if`, `else`, 그리고 `break`를 혼합해야 합니다; 원한다면 이렇게 사용해도 됩니다.

하지만, 이런 패턴은 매우 보편적이기 때문에 이와 동일한 구조자가 Rust에는 내장되어 있으며, 이를 `while` 반복문이라 부릅니다. 다음의 예제를 통해 `while`을 사용해 봅시다: 프로그램은 세 번 반복되고, 반복 때마다 카운트 다운됩니다. 마침내 반복이 끝나면 다른 메시지를 출력하고 종료됩니다:

Filename: src/main.rs

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);

        number = number - 1;
    }

    println!("LIFTOFF!!!!");
}
```

이 구조자는 loop, if, else 및 break를 사용하는 경우 필요한 많은 중첩을 제거하며, 더 깔끔합니다. 조건이 true인 동안 코드가 실행되고; 그렇지 않으면 루프에서 벗어납니다.

for와 함께하는 콜렉션 반복하기

우리는 `while` 구조자를 통해 배열과 같은, 콜렉션의 각 요소에 걸쳐 반복 수행 할 수 있습니다. 예를 들어서, Listing 3-5을 살펴봅시다:

Filename: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index = index + 1;
    }
}
```

Listing 3-5: `while` 반복문을 사용해 콜렉션의 각 요소들을 순회하기

여기서, 코드는 배열의 요소에 걸쳐 카운트를 증가시킵니다. 이 색인은 `0`에서 시작하고, 배열의 마지막 순서 까지 반복됩니다 (즉, `index < 5`가 참이 아닐 때까지). 이 코드를 수행하면 배열의 모든 요소가 출력되게 됩니다.

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

예상했던 대로, 5개인 배열 모든 값이 터미널에 표시됩니다. `index` 값이 `5`에 오는 시점에, 그러니까 배열의 6번째 값에 접근하기 전에 반복은 중지되어야 합니다.

그러나 이런 방식은 에러가 발생하기 쉽습니다; 우리가 정확한 길이의 색인을 사용하지 못하면 프로그램은 패닉을 발생합니다. 또한 느린데, 이유는 컴파일러가 실행 간에 반복문을 통해 반복될 때마다 요소에 대한 조건 검사를 수행하는 런타임 코드를 추가하기 때문입니다.

보다 효율적인 대안으로, 우리는 `for` 반복문을 사용하여 콜렉션의 각 요소에 대한 코드를 수행할 수 있습니다. `for` 반복문은 다음 Listing 3-6과 같습니다:

Filename: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", element);
    }
}
```

Listing 3-6: `for` 반복문을 사용해 콜렉션의 각 요소를 순회하기

우리가 이 코드를 수행하면, 항목 3-5와 같은 결과를 볼 수 있습니다. 더 중요한 것은, 우리는 이제 코드의 안전성을 높이고 배열의 끝을 넘어가거나 충분한 길이를 지정하지 못해 일부 아이템이 누락되어 발생할 수 있는 버그의 가능성을 제거했습니다.

예를 들어, 코드 3-5의 코드에서 `a` 배열에서 항목을 제거 했지만 조건을 `while index < 4`로 업데이트하지 않으면 코드는 패닉을 발생합니다. `for` 반복문을 사용하면, 당신이 배열의 수를 변경한 경우에도 다른 코드를 변경해야 할 필요가 없습니다. (역주 : 당신은 살면서 변경한 배열의 수를 기억하고 있는가?)

`for` 반복문이 안전하고 간결하기 때문에 이들은 가장 보편적으로 사용되는 반복문 구조자입니다. 항목 3-5에서처럼 `while` 반복문을 사용하여 특정 횟수만큼 코드를 반복하려는 경우에도, 대부분의 Rust 사용자들은 `for` 반복문을 사용하고자 할 것입니다. 이런 사용을 위해 Rust에서 기본 라이브러리로 제공하는

Range을 사용하게 됩니다. **Range**는 한 숫자에서 다른 숫자 전까지 모든 숫자를 차례로 생성합니다.

여기 **for** 반복문과 아직 설명하지 않은 **range**를 역순하는 **rev** 메소드를 사용하는 카운트다운 프로그램이 있습니다:

Filename: src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{}!", number);
    }
    println!("LIFTOFF!!!!");
}
```

꽤 괜찮은 코드인것 같죠?

결론

해냈어요! 무지 긴 장이었어: 우리는 변수, 스칼라, **if** 식과 반복문까지 배웠어요! 혹시 이번 장에서 나온 내용을 연습해보고 싶으면 다음을 수행하는 프로그램을 만들어 보세요.

- 화씨와 섭씨를 상호 변환.
- n번째 피보나치 수열 생성.
- 크리스마스 캐롤 “The Twelve Days of Christmas”의 가사를 반복문을 활용해 출력.

다음으로 넘어갈 준비가 되셨습니까? 우리는 이제 일반적인 다른 언어에는 존재하지 않는 개념에 대해서 다루고자 합니다 : 소유권.

소유권 이해하기

소유권(Ownership)은 러스트의 가장 유니크한 특성이며, 러스트가 가비지 컬렉터 없이 메모리 안정성 보장을 하게 해줍니다. 그러므로, 소유권이 러스트 내에서 어떻게 동작하는지 이해하는 것은 중요합니다. 이 장에서는 소유권 뿐만 아니라 이와 관련된 특성들: 빌림, 슬라이스, 그리고 러스트가 메모리에 데이터를 저장하는지 등을 알아보겠습니다.

소유권이 뭔가요?

러스트의 핵심 기능은 바로 소유권입니다. 이 기능은 직관적으로 설명할 수 있지만, 언어의 나머지 부분에 깊은 영향을 끼칩니다.

모든 프로그램은 실행하는 동안 컴퓨터의 메모리를 사용하는 방법을 관리해야 합니다. 몇몇 언어들은 프로그램이 실행될 때 더 이상 사용하지 않는 메모리를 끊임없이 찾는 가비지 컬렉션을 갖고 있습니다; 다른 언어들에서는 프로그래머가 직접 명시적으로 메모리를 할당하고 해제해야 합니다. 러스트는 제 3의 접근법을 이용합니다: 메모리는 컴파일 타임에 컴파일러가 체크할 규칙들로 구성된 소유권 시스템을 통해 관리됩니다. 소유권 기능들의 어떤 것도 런타임 비용이 발생하지 않습니다.

소유권이란 개념이 많은 프로그래머들에게 새로운 것이기 때문에, 이해하고 사용하는 데에는 약간의 시간이 걸립니다만, 좋은 소식은 여러분이 러스트와 소유권 시스템의 규칙에 더 많은 경험을 할수록, 여러분은 더 안전하고 더 효율적인 코드를 자연스럽게 개발할 수 있게 될 것이라는 거죠. 견뎌내세요!

여러분이 소유권을 이해했을 때, 여러분은 러스트를 유니크하게 만드는 기능들을 이해하기 위한 견고한 기초를 가지게 될 것입니다. 이 장에서, 여러분은 매우 흔한 데이터 구조인 문자열에 집중된 몇가지 예제를 통해 소유권에 대해 배우게 될 것입니다.

스택과 힙

많은 프로그래밍 언어들 안에서, 우리는 그렇게 자주 스택과 힙에 대한 생각을 할 필요가 없습니다. 그렇지만 러스트와 같은 시스템 프로그래밍 언어에서는, 값이 스택에 있는지 힙에 있는지의 여부가 언어의 동작 방식과 우리의 결단에 더 큰 영향을 줍니다. 우리는 이 장의 뒤쪽에서 스택과 힙에 관계된 소유권의 일부분을 기술할 것이기에, 여기서는 준비 삼아 간략한 설명만 하겠습니다.

스택과 힙 둘다 여러분의 코드상에서 런타임에 사용할 수 있는 메모리의 부분입니다만, 이들은 각기 다른 방식으로 구조화 되어 있습니다. 스택은 값을 받아들인 순서대로 값을 저장하고 반대 방향으로 값을 지웁니다. 이것을 *last in, first out*이라고 하죠. 쌓여있는 접시를 생각해보세요; 여러분이 접시를 더 추가하려면 접시더미의 꼭대기에 쌓아올리고, 여러분이 접시가 필요해지면 꼭대기에서부터 한장 꺼내게 됩니다. 중간이나 밑에서부터 접시를 추가하거나 제거하는 건 잘 안될겁니다! 데이터를 추가하는 것을 *스택에 푸시하기* (*pushing on the stack*)라고 부르고, 데이터를 제거하는 것을 *스택을 팝하기* (*popping off the stack*)라고 부릅니다.

스택은 데이터에 접근하는 방식 덕택에 빠릅니다: 이 방식은 새로운 데이터를 넣어두기 위한 공간 혹은 데이터를 가져올 공간을 검색할 필요가 전혀 없는데, 바로 그 공간이 항상 스택의 꼭대기(*top*)이기 때문입니다. 스택을 빠르게 해주는 또 다른 특성은 스택에 담긴 모든 데이터가 결정되어 있는 고정된 크기를 갖고 있어야 한다는 점입니다.

컴파일 타임에 크기가 결정되어 있지 않거나 크기가 변경될 수 있는 데이터를 위해서는, 힙에 데이터를 저장할 수 있습니다. 힙은 조금 더 복잡합니다: 데이터를 힙에 넣을 때, 먼저 저장할 공간이 있는지 물어봅니다. 그러면 운영체제가 충분히 커다란 힙 안의 빈 어떤 지점을 찾아서 이 곳을 사용중이라고 표시하고, 해당 지점의 포인터를 우리에게 돌려주죠. 이 절차를 힙 공간 할당하기(*allocating on the heap*)라고 부르고, 종종 그냥 "할당(allocating)"으로 줄여 부릅니다. 스택에 포인터를 푸싱하는 것은 할당에 해당되지 않습니다. 포인터는 결정되어 있는 고정된 크기의 값이므로, 우리는 스택에 포인터를 저장할 수 있지만, 실제 데이터를 사용하고자 할 때는 포인터를 따라가야 합니다.

힙에 저장된 데이터에 접근하는 것은 스택에 저장된 데이터에 접근하는 것보다 느린데, 그 이유는 포인터가 가리킨 곳을 따라가야 하기 때문입니다. 현대 프로세서들은 메모리 내부를 덜 뛰어다닐 때 더 빨라집니다. 유사한 예로, 여러 테이블로부터 주문을 받는 레스토랑의 웨이터를 생각해보세요. 다음 테이블로 움직이기 전에 지금 테이블에서 모든 주문을 다 받는 것이 가장 효율적이겠죠. A 테이블에서 하나 주문 받고, 다시 B 테이블로 가서 하나 주문 받고, 다시 A로, 다시 B로 가며 하나씩 주문을 받으면 훨씬 느려질 겁니다. 이와 마찬가지로, 프로세서는 (힙에 있는 데이터와 같이) 멀리 떨어져 있는 데이터들 보다는 (스택에 있는 것과 같이) 붙어있는 데이터들에 대한 작업을 하면 더 빨라집니다. 힙으로부터 큰 공간을 할당받는 것 또한 시간이 걸릴 수 있습니다.

코드의 어느 부분이 힙의 어떤 데이터를 사용하는지 추적하는 것, 힙의 중복된 데이터의 양을 최소화하는 것, 그리고 힙 내에 사용하지 않는 데이터를 제거하여 공간이 모자라지 않게 하는 것은 모두 소유권과 관계된 문제들입니다. 여러분이 소유권을 이해하고 나면, 여러분은 더 이상 스택과 힙에 대한 생각이 자주 필요치 않게 될 겁니다만, 힙 데이터를 관리하는 것이 곧 소유권의 존재 이유임을 알게 되는 것은 이것이 어떤 방식으로 작동하는지 설명하는데 도움을 줄 수 있습니다.

소유권 규칙

먼저, 소유권 규칙을 알아봅시다. 이것들을 설명할 예제들을 보는 내내 다음의 소유권 규칙들을 명심하세요:

1. 러스트의 각각의 값은 해당값의 오너(owner)라고 불리우는 변수를 갖고 있다.
 2. 한번에 딱 하나의 오너만 존재할 수 있다.
 3. 오너가 스코프 밖으로 벗어나는 때, 값은 버려진다(dropped).
-

변수의 스코프

우리는 이미 2장에서 완성된 형태의 러스트 프로그램 예제를 살펴봤습니다. 이제 과거의 기초 문법 형태로 돌아가서, `fn main() { }` 코드를 예제에 붙이지 않을 텐데, 여러분들이 코드를 따라하려면 `main` 함수에 직접 예제들을 넣어야 할 겁니다. 결과적으로, 우리의 예제들은 좀더 간략해져서 보일러 플레이트 코드에 비해 실

제 디테일에 초점을 맞출 수 있도록 해줄 것입니다.

소유권에 대한 첫 예제로서, 변수들의 스코프를 보겠습니다. 스코프란 프로그램 내에서 아이템이 유효함을 표시하기 위한 범위입니다. 아래처럼 생긴 변수가 있다고 해봅시다:

```
let s = "hello";
```

변수 `s`는 스트링 리터럴을 나타내는데, 스트링 리터럴의 값은 우리의 프로그램의 텍스트 내에 하드코딩되어 있습니다. 변수는 선언된 시점부터 현재의 스코프가 끝날 때까지 유효합니다. 아래 예제 Listing 4-1은 변수 `s`가 유효한 지점을 주석으로 표시했습니다:

```
{
    // s는 유효하지 않습니다. 아직 선언이 안됐거든요.
    let s = "hello"; // s는 이 지점부터 유효합니다.

    // s를 가지고 뭔가 합니다.
} // 이 스코프는 이제 끝이므로, s는 더이상 유효하지 않습니다.
```

Listing 4-1: 변수와 이 변수가 유효한 스코프

바꿔 말하면, 두가지 중요한 지점이 있습니다:

1. 스코프 안에서 `s`가 등장하면, 유효합니다.
2. 이 유효기간은 스코프 밖으로 벗어날 때까지 지속됩니다.

이 지점에서, 스코프와 변수가 유효한 시점 간의 관계는 다른 프로그래밍 언어와 비슷합니다. 이제 우리는 이에 대한 이해를 기초로 하여 `String` 타입을 소개함으로써 계속 쌓아나갈 것입니다.

String 타입

소유권 규칙을 설명하기 위하여, 우리는 3장에서 다른 바 있는 타입보다 더 복잡한 데이터 타입이 필요합니다. 우리가 이전에 봤던 모든 데이터 타입들은 스택에 저장되었다가 스코프를 벗어날 때 스택으로부터 팝 됩니다만, 우리는 이제 힙에 저장되는 데이터를 관찰하고 러스트는 과연 어떻게 이 데이터를 비워내는지 설명할 필요가 있습니다.

우리는 여기서 `String`을 예제로 활용하되, 소유권과 관련된 `String` 내용의 일부분에 집중할 것입니다. 이러한 관점은 표준 라이브러리나 여러분들이 만들 다른 복잡한 데이터 타입에도 적용됩니다. `String`에 대해서는 8장에서 더 자세히 다루겠습니다.

스트링 리터럴을 이미 봤는데, 이 값은 프로그램 안에 하드코딩 되어 있습니다. 문자열 값은 편리하지만, 여러분이 텍스트를 필요로 하는 모든 경우에 대해 항상 적절하지 않습니다. 그 중 한가지 이유로, 문자열 값은 불변입니다(immutable). 또 다른 이유는 모든 문자열이 우리가 프로그래밍 하는 시점에서 다 알수 있는 것이

아니란 점입니다: 예를 들면, 사용자의 입력을 받아 저장하고 싶다면요? 이러한 경우들에 대해서, 러스트는 두번째 문자열 타입인 **String**을 제공합니다. 이 타입은 힙에 할당되고 그런고로 컴파일 타임에는 우리가 알 수 없는 양의 텍스트를 저장할 수 있습니다. 여러분은 스트링 리터럴로부터 **from**이라는 함수를 이용해서 **String**을 아래처럼 만들 수 있습니다:

```
let s = String::from("hello");
```

더블 콜론(::<>)은 우리가 **string_from**과 같은 이름을 쓰기 보다는 **String** 타입 아래의 **from** 함수를 특정지을 수 있도록 해주는 네임스페이스 연산자입니다. 우리는 이러한 문법에 대해 5장의 "메소드 문법" 부분에서 더 자세히 다룰 것이고, 모듈에서의 네임스페이스와 관련한 이야기는 7장에서 할 것입니다.

이러한 종류의 문자열은 변경 가능합니다:

```
let mut s = String::from("hello");
s.push_str(", world!"); // push_str()은 해당 스트링 리터럴을 스트링에 붙여줍니다.
println!("{}", s); // 이 부분이 `hello, world!`를 출력할 겁니다.
```

그러니까, 여기서 어떤게 달라졌나요?, 왜 **String**은 변할 수 있는데 스트링 리터럴은 안될까요? 차이점은 두 타입이 메모리를 쓰는 방식에 있습니다.

메모리와 할당

스트링 리터럴의 경우, 우리는 내용물을 컴파일 타임에 알 수 있으므로 텍스트가 최종 실행파일에 직접 하드 코딩 되었고, 이렇게 하면 스트링 리터럴이 빠르고 효율적이 됩니다. 그러나 이는 문자열이 변경되지 않는 것을 전제로 하는 특성입니다. 불행하게도, 우리는 컴파일 타임에 크기를 알 수 없는 경우 및 실행 중 크기가 변할 수도 있는 경우의 텍스트 조각을 바이너리 파일에 집어넣을 수 없습니다.

String 타입은 변경 가능하고 커질 수 있는 텍스트를 지원하기 위해 만들어졌고, 우리는 힙에서 컴파일 타임에는 알 수 없는 어느 정도 크기의 메모리 공간을 할당받아 내용물을 저장할 필요가 있습니다. 이는 즉 다음을 의미합니다:

1. 런타임에 운영체제로부터 메모리가 요청되어야 한다.
2. **String**의 사용이 끝났을 때 운영체제에게 메모리를 반납할 방법이 필요하다.

첫번째는 우리가 직접 수행합니다: 우리가 **String::from**을 호출하면, 구현부분에서 필요한 만큼의 메모리를 요청합니다. 이는 프로그래밍 언어들 사이에서 매우 일반적입니다.

하지만, 두번째는 다릅니다. 가비지 컬렉터(GC)를 갖고 있는 언어들의 경우, GC가 더이상 사용하지 않는 메

모리 조각을 계속해서 찾고 지워주며, 우리는 프로그래머로서 이와 관련한 생각을 안해도 됩니다. GC가 없을 경우, 할당받은 메모리가 더 필요없는 시점을 알아서 명시적으로 이를 반납하는 코드를 호출하는 것은 프로그래머의 책임입니다. 이를 올바르게 하는 것은 역사적으로 어려운 문제로 취급받았습니다. 우리가 잊어먹으면? 메모리를 낭비하는 것이죠. 너무 빨리 반납해버리면? 유효하지 않은 변수를 갖게 될 겁니다. 만일 반납을 두번하면? 이것도 버그죠. 우리는 딱 한번의 `allocate`와 한번의 `free` 쌍을 사용해야 합니다.

러스트는 다른 방식으로 이 문제를 다룹니다: 메모리는 변수가 소속되어 있는 스코프 밖으로 벗어나는 순간 자동으로 반납됩니다. 여기 스트링 리터럴 대신 `String`을 사용한 Listing 4-1의 스코프 예제가 있습니다:

```
{
    let s = String::from("hello"); // s는 여기서부터 유효합니다
                                // s를 가지고 뭔가 합니다
}
                                // 이 스코프는 끝났고, s는 더 이상
                                // 유효하지 않습니다
```

`String`이 요구한 메모리를 운영체제에게 반납하는 자연스러운 지점이 있죠: `s`가 스코프 밖으로 벗어날 때입니다. 변수가 스코프 밖으로 벗어나면, 러스트는 우리를 위해 특별한 함수를 호출합니다. 이 함수를 `drop`이라고 부르고, `String`의 개발자가 메모리를 반환하도록 하는 코드를 집어넣을 수 있습니다. 러스트는 `}` 괄호가 닫힐때 자동적으로 `drop`을 호출합니다.

노트: C++에서는 이렇게 아이템의 수명주기의 끝나는 시점에 자원을 해제하는 패턴을 종종 **자원 습득이 곧 초기화**(*Resource Acquisition Is Initialization, RAII*)라고 부릅니다. 러스트의 `drop` 함수는 여러분이 RAII 패턴을 경험해본 적 있다면 익숙할 것입니다.

이 패턴은 러스트 코드가 작성되는 방법에 깊은 영향을 줍니다. 지금은 단순해 보이시겠지만, 우리가 힘에 할 당시킨 데이터를 사용하는 여러 개의 변수를 사용하고자 할 경우와 같이 좀더 복잡한 상황에서, 코드의 동작은 예기치 못할 수 있습니다. 이제 그런 경우들을 좀더 탐험해봅시다.

변수와 데이터가 상호작용하는 방법: 이동(move)

여러 개의 변수들은 러스트에서 서로 다른 방식으로 같은 데이터에 대해 상호작용을 할 수 있습니다. Listing 4-2의 정수형을 이용한 예제를 한번 보겠습니다:

```
let x = 5;
let y = x;
```

Listing 4-2: 변수 `x`의 정수값을 `y`에 대입하기

우리는 아마도 다른 언어들에서의 경험을 토대로 어떤 일이 벌어지는지 추측할 수 있습니다: “정수값 5를 `x`에 묶어놓고; `x`의 값의 복사본을 만들어 `y`에 묶는다.” 우리는 이제 `x`와 `y` 두 개의 변수를 갖게 되었고, 둘 다 5와 같습니다. 정수값이 결정되어 있는 고정된 크기의 단순한 값이고, 5라는 값들이 스택에 푸쉬되기 때문에, 실제로도 이렇게 됩니다.

이제 `String` 버전을 봅시다:

```
let s1 = String::from("hello");
let s2 = s1;
```

이 코드는 이전의 코드와 매우 유사해 보여서, 동작하는 방식도 동일할 것이라고 가정할지도 모르겠습니다: 즉, 두번째 줄이 `s1`의 복사본을 만들어서 `s2`에 묶어놓는 식으로 말이죠. 그렇지만 이는 실제 동작과 다른 생각입니다.

좀 더 완전히 설명하기 위해, `String`이 Figure 4-3에서와 같이 생겼다는 것을 주목합시다. `String`은 그림의 왼쪽과 같이 세 개의 부분으로 이루어져 있습니다: 문자열의 내용물을 담고 있는 메모리의 포인터, 길이, 그리고 용량입니다. 이 데이터의 그룹은 스택에 저장됩니다. 내용물을 담은 오른쪽의 것은 힙 메모리에 있습니다.

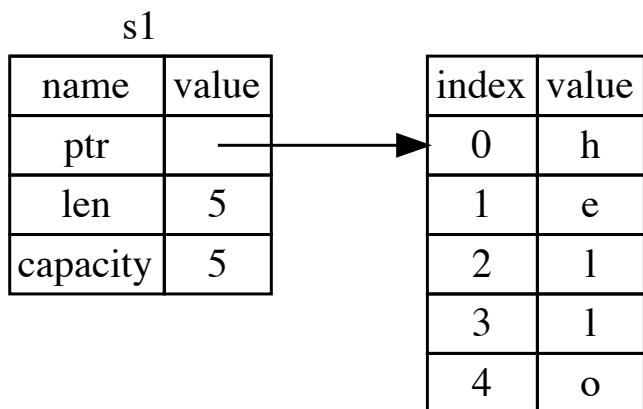


Figure 4-3: `s1` 변수에 `"hello"` 값이 저장된 `String`의 메모리 구조

길이값은 바이트 단위로 `String`의 내용물이 얼마나 많은 메모리를 현재 사용하고 있는지를 말합니다. 용량값은 바이트 단위로 `String`이 운영체제로부터 얼마나 많은 양의 메모리를 할당 받았는지를 말합니다. 길이와 용량의 차이는 중요합니다만, 이번 내용에서는 아닙니다. 그러니까 현재로서는 용량값을 무시하셔도 좋겠습니다.

`s2`에 `s1`을 대입하면, `String` 데이터가 복사되는데, 이는 스택에 있는 포인터, 길이값, 그리고 용량값이 복사된다는 의미입니다. 포인터가 가리키고 있는 힙 메모리 상의 데이터는 복사되지 않습니다. 달리 말하면, 메모리 내의 데이터 구조는 Figure 4-4와 같이 됩니다.

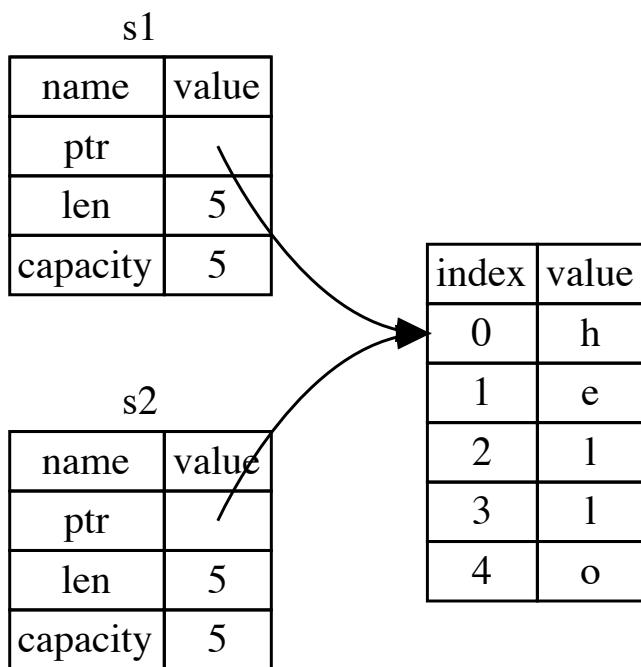


Figure 4-4: `s1`의 포인터, 길이값, 용량값이 복사된 `s2` 변수의 메모리 구조

메모리 구조는 Figure 4-5와 같지 않는데, 이 그림은 러스트가 힙 메모리 상의 데이터까지도 복사한다면 벌어질 일입니다. 만일 러스트가 이렇게 동작한다면, 힙 안의 데이터가 클 경우 `s2 = s1` 연산은 런타임 상에서 매우 느려질 가능성이 있습니다.

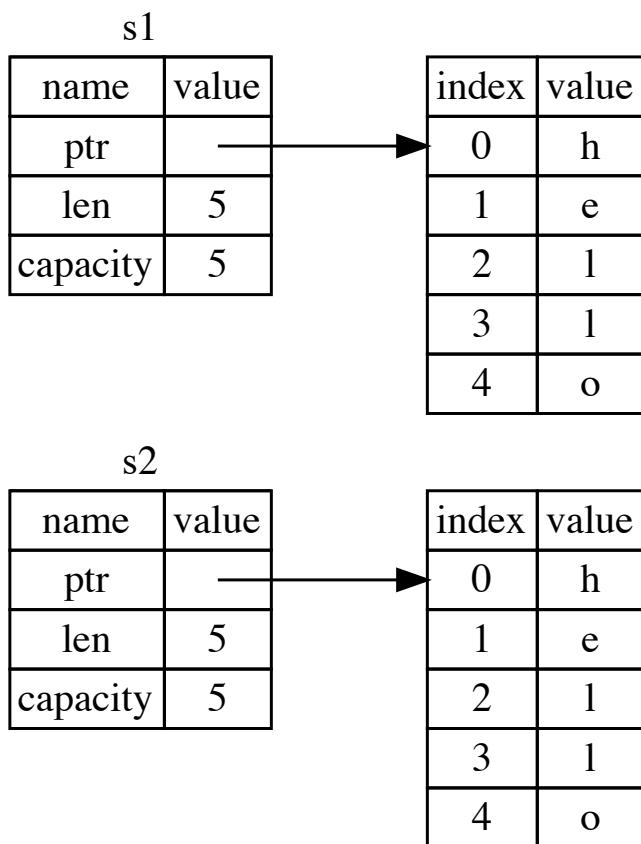


Figure 4-5: 러스트가 힙 데이터까지 복사하게 될 경우 `s2 = s1` 가 만들 또 다른 가능성

앞서 우리는 변수가 스코프 밖으로 벗어날 때, 러스트는 자동적으로 `drop` 함수를 호출하여 해당 변수가 사용하는 힙 메모리를 제거한다고 했습니다. 하지만 Figure 4-4에서는 두 데이터 포인터가 모두 같은 곳을 가리키고 있는 것이 보입니다. 이는 곧 문제가 됩니다: `s2`와 `s1`이 스코프 밖으로 벗어나게 되면, 둘 다 같은 메모리를 해제하려 할 것입니다. 이는 *두번 해제(double free)* 오류라고 알려져 있으며 이전에 언급한 바 있는 메모리 안정성 버그들 중 하나입니다. 메모리를 두번 해제하는 것은 메모리 손상(memory corruption)의 원인이 되는데, 이는 보안 취약성 문제를 일으킬 가능성이 있습니다.

메모리 안정성을 보장하기 위해서, 러스트에서는 이런 경우 어떤 일이 일어나는지 한가지 더 디테일이 있습니다. 할당된 메모리를 복사하는 것을 시도하는 대신, 러스트에서는 `s1`이 더 이상 유효하지 않다고 간주하고, 그러므로 러스트는 `s1` 가 스코프 밖으로 벗어났을 때 아무것도 해제할 필요가 없어집니다. `s1`을 `s2` 가 만들어진 후에 사용하려고 할 때 어떤 일이 벌어지는지 확인해 볼시다:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{} , world!", s1);
```

여러분은 아래와 같은 에러 메세지를 보게 될텐데, 그 이유는 러스트가 여러분으로부터 유효하지 않은 참조자를 사용하는 것을 막기 때문입니다:

```

error[E0382]: use of moved value: `s1`
--> src/main.rs:4:27
3 |     let s2 = s1;
   |         -- value moved here
4 |     println!("{}{}, world!", s1);
   |             ^^^ value used here after move
   |
   = note: move occurs because `s1` has type `std::string::String`,
which does not implement the `Copy` trait

```

만일 여러분이 다른 언어로 프로그래밍 하는 동안 “얕은 복사(shallow copy)”와 “깊은 복사(deep copy)”라는 용어를 들어보셨다면, 데이터의 복사 없이 포인터와 길이값 및 용량값만 복사한다는 개념이 얕은 복사와 비슷하게 들릴지도 모르겠습니다. 하지만 리스트는 첫번째 변수를 무효화 시키기도 하기 때문에, 이를 얕은 복사라고 부르는 대신 이동(move)이라 말합니다. 여기서 우리는 `s1`이 `s2`로 이동되었다고 말하는 식으로 위 코드를 읽을 것입니다. 그러므로 실제로 일어낸 일은 Figure 4-6과 같습니다.

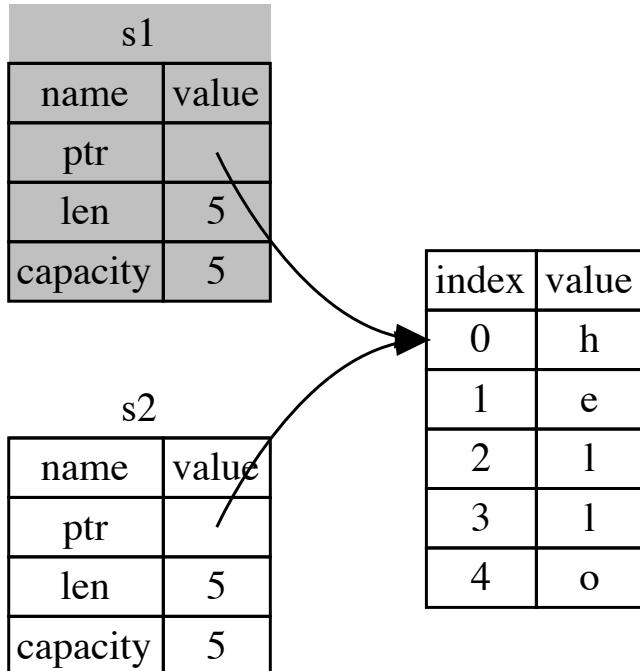


Figure 4-6: `s1`이 무효화된 후의 메모리 구조

이것이 우리 문제를 해결해줍니다! 오직 `s2`만 유효한 상황에서, 스코프 밖으로 벗어나면 혼자 메모리를 해제할 것이고, 일이 잘 처리되겠습니다.

여기에 더해서, 이러한 경우가 함축하는 디자인 선택이 있습니다: 리스트는 결코 자동적으로 여러분의 데이터에 대한 “깊은” 복사본을 만들지 않을 것입니다. 그러므로, 어떠한 자동적인 복사라도 런타임 실행 과정에서 효율적일 것이라 가정할 수 있습니다.

변수와 데이터가 상호작용하는 방법: 클론

만일 `String`의 스택 데이터 만이 아니라, 힙 데이터를 깊이 복사하기를 정말 원한다면, `clone`이라 불리우는 공용 메소드를 사용할 수 있습니다. 이 메소드 문법에 대해서는 5장에서 다루게 될 것입니다만, 이 메소드가 많은 프로그래밍 언어들 사이에서 혼란 특성이기 때문에, 여러분은 아마도 전에 이런 것들을 본적이 있을지도 모르겠습니다.

`clone` 메소드가 동작하는 예제를 보겠습니다:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

이 코드는 잘 동작하고 Figure 4-5가 나타내는, 즉 힙 데이터가 정말로 복사되는 동작을 여러분이 명시적으로 만들어낼 수 있는 방법입니다.

`clone`을 호출하는 부분을 보면, 어떤 비용이 많이 들어갈지도 모르는 코드가 실행되는 중이란 것을 알 수 있게 됩니다. 이는 무언가 다른 동작이 수행되는 것을 알려주는 시각적인 지시자입니다.

스택에만 있는 데이터: 복사

우리가 아직 다루지 않은 또다른 부분이 있습니다. 아래 코드는 앞서 Listing 4-2에서 본 정수값을 이용하는 코드로, 잘 동작하며 유효합니다:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

하지만 이 코드는 우리가 방금 배운 것과 대립되는 것처럼 보입니다: `clone`을 호출하지 않았지만, `x`도 유효하며 `y`로 이동하지도 않았지요.

그 이유는 정수형과 같이 컴파일 타임에 결정되어 있는 크기의 타입은 스택에 모두 저장되기 때문에, 실제 값의 복사본이 빠르게 만들어질 수 있습니다. 이는 변수 `y`가 생성된 후에 `x`가 더 이상 유효하지 않도록 해야 할 이유가 없다는 뜻입니다. 바꿔 말하면, 여기서는 깊은 복사와 얕은 복사 간의 차이가 없다는 것으로, `clone`을 호출하는 것이 보통의 얕은 복사와 아무런 차이점이 없어 우리는 이를 그냥 버릴 수 있다는 것입니다.

러스트는 정수형과 같이 스택에 저장할 수 있는 타입에 대해 달수 있는 `Copy` 트레이트라고 불리우는 특별한 어노테이션(annotation)을 가지고 있습니다 (트레이트에 관해서는 10장에서 더 자세히 보겠습니다). 만일 어

떤 타입이 **Copy** 트레이잇을 갖고 있다면, 대입 과정 후에도 예전 변수를 계속 사용할 수 있습니다. 러스트는 만일 그 타입 혹은 그 타입이 가지고 있는 부분 중에서 **Drop** 트레이잇을 구현한 것이 있다면 **Copy** 트레이잇을 어노테이션 할 수 없게끔 합니다. 만일 어떤 타입이 스코프 밖으로 벗어났을 때 어떤 특수한 동작을 필요로 하고 우리가 그 타입에 대해 **Copy** 어노테이션을 추가한다면, 컴파일 타임 오류를 보게 됩니다. **Copy** 어노테이션을 여러분의 타입에 어떤 식으로 추가하는지 알고 싶다면, 부록 C의 파생 가능한 트레이잇(Derivable Traits)을 보세요.

그래서 어떤 타입이 **Copy** 가 될까요? 여러분은 주어진 타입에 대해 확신을 하기 위해 문서를 확인할 수도 있겠지만, 일반적인 규칙으로서 단순한 스칼라 값들의 묶음은 **Copy** 가 가능하고, 할당이 필요하거나 어떤 자원의 형태인 경우 **Copy** 를 사용할 수 없습니다. **Copy** 가 가능한 몇가지 타입을 나열해 보겠습니다:

- `u32` 와 같은 모든 정수형 타입들
- `true` 와 `false` 값을 갖는 부울린 타입 `bool`
- `f64` 와 같은 모든 부동 소수점 타입들
- **Copy** 가 가능한 타입만으로 구성된 튜플들. `(i32, i32)` 는 **Copy** 가 되지만, `(i32, String)` 은 안됩니다.

소유권과 함수

함수에게 값을 넘기는 의미론(semantics)은 값을 변수에 대입하는 것과 유사합니다. 함수에게 변수를 넘기는 것은 대입과 마찬가지로 이동하거나 복사될 것입니다. Listing 4-7은 변수가 스코프 안으로 들어갔다 밖으로 벗어나는 것을 주석과 함께 보여주는 예입니다:

Filename: src/main.rs

```

fn main() {
    let s = String::from("hello"); // s가 스코프 안으로 들어왔습니다.

    takes_ownership(s); // s의 값이 함수 안으로 이동했습니다...
    // ... 그리고 이제 더이상 유효하지 않습니다.

    let x = 5; // x가 스코프 안으로 들어왔습니다.

    makes_copy(x); // x가 함수 안으로 이동했습니다만,
    // i32는 Copy가 되므로, x를 이후에 계속
    // 사용해도 됩니다.

} // 여기서 x는 스코프 밖으로 나가고, s도 그 후 나갑니다. 하지만 s는 이미 이동되었으므로,
// 별다른 일이 발생하지 않습니다.

fn takes_ownership(some_string: String) { // some_string이 스코프 안으로 들어왔습니다.
    println!("{}", some_string);
} // 여기서 some_string이 스코프 밖으로 벗어났고 `drop`이 호출됩니다. 메모리는
// 해제되었습니다.

fn makes_copy(some_integer: i32) { // some_integer이 스코프 안으로 들어왔습니다.
    println!("{}", some_integer);
} // 여기서 some_integer가 스코프 밖으로 벗어났습니다. 별다른 일은 발생하지 않습니다.

```

Listing 4-7: 소유권과 스코프에 대한 설명이 주석으로 달린 함수들

만일 우리가 `s`를 `takes_ownership` 함수를 호출한 이후에 사용하려 한다면, 러스트는 컴파일 타임 오류를 낼 것입니다. 이러한 정적 확인은 여러 실수들을 방지해 줍니다. 이후에 변수들을 사용할 수 있는지, 그리고 그러한 것을 소유권 규칙이 막아주는지를 확인해보려면 `main` 안에 `s`와 `x`에 관한 코드를 추가해보세요.

반환 값과 스코프

값의 반환 또한 소유권을 이동시킵니다. Listing 4-7과 비슷한 주석이 달린 예제를 하나 봅시다:

Filename: src/main.rs

```

fn main() {
    let s1 = gives_ownership();           // gives_ownership은 반환값을 s1에게
                                         // 이동시킵니다.

    let s2 = String::from("hello");       // s2가 스코프 안에 들어왔습니다.

    let s3 = takes_and_gives_back(s2);   // s2는 takes_and_gives_back 안으로
                                         // 이동되었고, 이 함수가 반환값을 s3으로도
                                         // 이동시켰습니다.

} // 여기서 s3는 스코프 밖으로 벗어났으며 drop이 호출됩니다. s2는 스코프 밖으로
// 벗어났지만 이동되었으므로 아무 일도 일어나지 않습니다. s1은 스코프 밖으로
// 벗어나서 drop이 호출됩니다.

fn gives_ownership() -> String {        // gives_ownership 함수가 반환 값을
                                         // 호출한 쪽으로 이동시킵니다.

    let some_string = String::from("hello"); // some_string이 스코프 안에 들어왔습니다.

    some_string                         // some_string이 반환되고, 호출한 쪽
                                         // 함수로 이동됩니다.

}

// takes_and_gives_back 함수는 String을 하나 받아서 다른 하나를 반환합니다.
fn takes_and_gives_back(a_string: String) -> String { // a_string이 스코프
                                         // 안으로 들어왔습니다.

    a_string // a_string은 반환되고, 호출한 쪽의 함수로 이동됩니다.
}

```

변수의 소유권은 모든 순간 똑같은 패턴을 따릅니다: 어떤 값을 다른 변수에 대입하면 값이 이동됩니다. 힙에 데이터를 갖고 있는 변수가 스코프 밖으로 벗어나면, 해당 값은 데이터가 다른 변수에 의해 소유되도록 이동하지 않는한 `drop`에 의해 제거될 것입니다.

모든 함수가 소유권을 가졌다가 반납하는 것은 조금 지루해 보입니다. 만일 함수에게 값을 사용할 수 있도록 하되 소유권은 갖지 않도록 하고 싶다면요? 함수의 본체로부터 얻어진 결과와 더불어 우리가 넘겨주고자 하는 어떤 값을 다시 쓰고 싶어서 함께 반환받아야 한다면 꽤나 짜증나겠지요.

이게 아래와 같이 튜플을 이용하여 여러 값을 돌려받는 식으로 가능하긴 합니다:

Filename: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len()함수는 문자열의 길이를 반환합니다.

    (s, length)
}
```

하지만 이건 너무 많이 나간 의례절차고 일반적인 개념로서는 과한 작업이 됩니다. 운좋게도, 러스트는 이를 위한 기능을 갖고 있으며, 참조자(*references*)라고 부릅니다.

참조자(References)와 빌림(Borrowing)

앞 절의 마지막에 등장한 튜플을 이용하는 이슈는 `String`을 호출하는 함수 쪽으로 반환함으로써 `calculate_length`를 호출한 이후에도 여전히 `String`을 이용할 수 있도록 하는 것인데, 그 이유는 `String`이 `calculate_length` 안쪽으로 이동되었기 때문입니다.

여기 값의 소유권을 넘기는 대신 개체에 대한 참조자(reference)를 인자로 사용하는 `calculate_length` 함수를 정의하고 이용하는 방법이 있습니다:

Filename: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

첫번째로, 변수 선언부와 함수 반환값에 있던 튜플 코드가 모두 없어진 것에 주목하세요. 두번째로, `calculate_length` 함수에 `&s1`를 넘기고, 함수의 정의 부분에는 `String`이 아니라 `&String`을 이용했다는 점을 기억하세요.

이 엠퍼센드(&) 기호가 참조자이며, 이는 여러분이 어떤 값을 소유권을 넘기지 않고 참조할수 있도록 해줍니다. Figure 4-8은 이에 대한 다이어그램입니다.

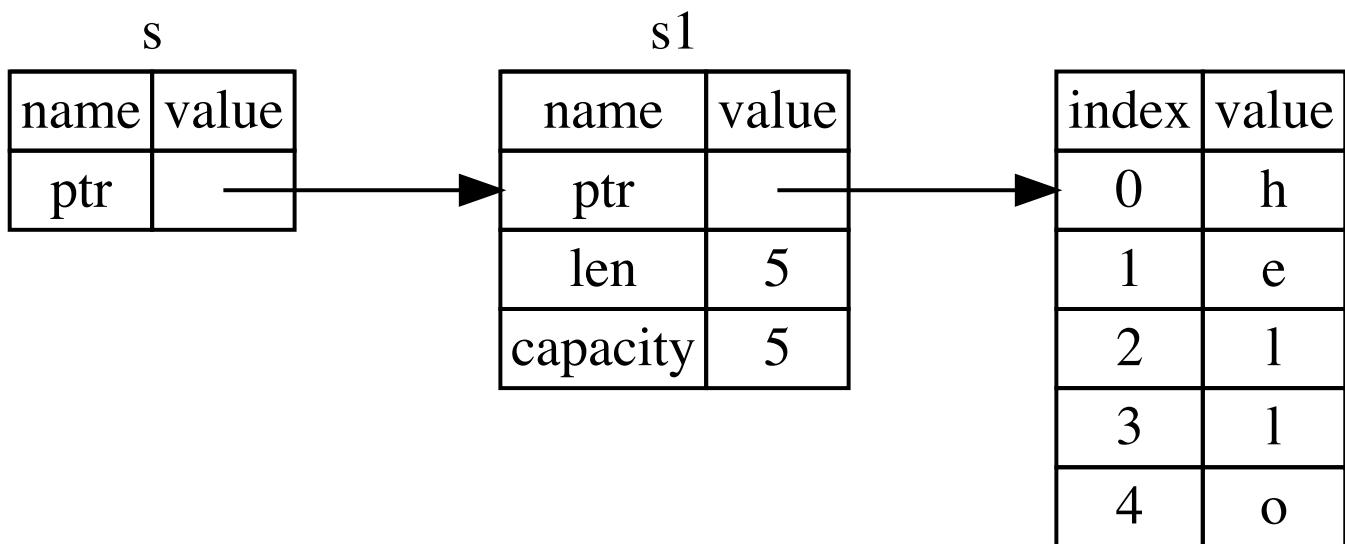


Figure 4-8: `String s1` 을 가리키고 있는 `&String s`

함수 호출 부분을 좀더 자세히 봅시다:

```
let s1 = String::from("hello");
let len = calculate_length(&s1);
```

`&s1` 문법은 우리가 `s1`의 값을 참조하지만 소유하지는 않는 참조자를 생성하도록 해줍니다. 소유권을 갖고 있는 않기 때문에, 이 참조자가 가리키는 값은 참조자가 스코프 밖으로 벗어났을 때도 메모리가 반납되지 않을 것입니다.

비슷한 이치로, 함수 시그니처도 `&`를 사용하여 인자 `s`의 타입이 참조자라는 것을 나타내고 있습니다. 설명을 위한 주석을 달아봅시다:

```
fn calculate_length(s: &String) -> usize { // s는 String의 참조자입니다
    s.len()
} // 여기서 s는 스코프 밖으로 벗어났습니다. 하지만 가리키고 있는 값에 대한 소유권이 없기
// 때문에, 아무런 일도 발생하지 않습니다.
```

변수 `s`가 유효한 스코프는 여느 함수의 파라미터의 스코프와 동일하지만, 소유권을 갖고 있지 않으므로 이 참조자가 스코프 밖으로 벗어났을 때 참조자가 가리키고 있는 값은 버리지 않습니다. 또한 실제 값 대신 참조자를 파라미터로 갖고 있는 함수는 소유권을 갖고 있지 않기 때문에 소유권을 되돌려주기 위해 값을 다시 반환할 필요도 없다는 뜻이 됩니다.

함수의 파라미터로 참조자를 만드는 것을 빌림이라고 부릅니다. 실제 생활에서 만일 어떤 사람이 뭔가를 소유하고 있다면, 여러분은 그걸 빌릴 수 있습니다. 여러분의 용무가 끝났을 때는 그것을 돌려주어야 합니다.

그러니까 만일 우리가 빌린 무언가를 고치려고 시도한다면 무슨 일이 생길까요? Listing 4-9의 코드를 시험해보세요. 스포일러 경고: 작동이 안될겁니다!

Filename: src/main.rs

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Listing 4-9: 빌린 값을 고치려 해보기

여기 오류를 보시죠:

```
error: cannot borrow immutable borrowed content `*some_string` as mutable
--> error.rs:8:5
8 |     some_string.push_str(", world");
|     ^^^^^^^^^^
```

변수가 기본적으로 불변인 것처럼, 참조자도 마찬가지입니다. 우리가 참조하는 어떤 것을 변경하는 것은 허용되지 않습니다.

가변 참조자(Mutable References)

Listing 4-9의 코드를 살짝만 바꾸면 오류를 고칠 수 있습니다:

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

먼저 `s`를 `mut`로 바꿔야 합니다. 그리고 `&mut s`로 가변 참조자를 생성하고 `some_string: &mut String`으로 이 가변 참조자를 받아야 합니다.

하지만 가변 참조자는 딱 한가지 큰 제한이 있습니다: 특정한 스코프 내에 특정한 데이터 조각에 대한 가변 참조자를 딱 하나만 만들 수 있다는 겁니다. 아래 코드는 실패할 겁니다:

Filename: src/main.rs

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;
```

여기 오류를 보시죠:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
--> borrow_twice.rs:5:19
   |
4 |     let r1 = &mut s;
   |             - first mutable borrow occurs here
5 |     let r2 = &mut s;
   |             ^ second mutable borrow occurs here
6 | }
   | - first borrow ends here
```

이 제한 사항은 가변을 허용하긴 하지만 매우 통제된 형식으로 허용합니다. 이것이 새로운 러스트인들이 힘들어하는 부분인데, 대부분의 언어들은 여러분이 원하는대로 값을 변형하도록 해주기 때문입니다. 하지만 이러한 제한이 가지는 이점은 바로 러스트가 컴파일 타임에 데이터 레이스(data race)를 방지할 수 있도록 해준다는 것입니다.

데이터 레이스는 아래에 정리된 세 가지 동작이 발생했을 때 나타나는 특정한 레이스 조건입니다:

1. 두 개 이상의 포인터가 동시에 같은 데이터에 접근한다.
2. 그 중 적어도 하나의 포인터가 데이터를 쓴다.
3. 데이터에 접근하는데 동기화를 하는 어떠한 메커니즘도 없다.

데이터 레이스는 정의되지 않은 동작을 일으키고 런타임에 이를 추적하고자 할 때는 이를 진단하고 고치기 어려울 수 있습니다; 러스트는 데이터 레이스가 발생할 수 있는 코드가 컴파일 조차 안되기 때문에 이 문제의 발생을 막아버립니다!

항상 우리는 새로운 스코프를 만들기 위해 중괄호를 사용하는데, 이는 그저 동시에 만드는 것이 아니게 해줌으로써, 여러 개의 가변 참조자를 만들 수 있도록 해줍니다.

```
let mut s = String::from("hello");

{
    let r1 = &mut s;

} // 여기서 r1은 스코프 밖으로 벗어났으므로, 우리는 아무 문제 없이 새로운 참조자를 만들 수 있습니다.

let r2 = &mut s;
```

가변 참조자와 불변 참조자를 혼용할 경우에 대한 비슷한 규칙이 있습니다. 아래 코드는 컴파일 오류가 발생합니다:

```
let mut s = String::from("hello");

let r1 = &s; // 문제 없음
let r2 = &s; // 문제 없음
let r3 = &mut s; // 큰 문제
```

여기 오류 메세지를 보시죠:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
--> borrow_thrice.rs:6:19
   |
4 |     let r1 = &s; // 문제 없음
   |             - immutable borrow occurs here
5 |     let r2 = &s; // 문제 없음
6 |     let r3 = &mut s; // 큰 문제
   |                 ^ mutable borrow occurs here
7 | }
```

어휴! 우리는 불변 참조자를 가지고 있을 동안에도 역시 가변 참조자를 만들 수 없습니다. 불변 참조자의 사용자는 사용중인 동안에 값이 값자기 바뀌리라 예상하지 않습니다! 하지만 여러 개의 불변 참조자는 만들 수 있는데, 데이터를 그냥 읽기만하는 것은 다른 것들이 그 데이터를 읽는데에 어떠한 영향도 주지 못하기 때문입니다.

때때로 이러한 오류들이 여러분을 좌절시킬지라도, 이것이 러스트 컴파일러가 (런타임이 아니라 컴파일 타임에) 일찌감치 잠재된 버그를 찾아내고, 왜 여러분의 데이터가 여러분 생각대로의 값을 갖고 있지 않은지 추적해 내려가는 대신 어느 지점이 문제인지를 정확히 보여주는 기능이란 점을 기억하세요.

댕글링 참조자(Dangling References)

포인터가 있는 언어에서는 자칫 잘못하면 **댕글링 포인터(dangling pointer)**를 만들기 쉬운데, 댕글링 포인터란 어떤 메모리를 가리키는 포인터를 보존하는 동안, 그 메모리를 해제함으로써 다른 개체에게 사용하도록 줘버렸을지도 모를 메모리를 참조하고 있는 포인터를 말합니다. 이와는 반대로, 리스트에서는 컴파일러가 모든 참조자들이 댕글링 참조자가 되지 않도록 보장해 줍니다: 만일 우리가 어떤 데이터의 참조자를 만들었다면, 컴파일러는 그 참조자가 스코프 밖으로 벗어나기 전에는 데이터가 스코프 밖으로 벗어나지 않을 것임을 확인해 줄 것입니다.

댕글링 참조자를 만드는 시도를 해봅시다:

Filename: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

위 코드의 오류 메세지입니다:

```
error[E0106]: missing lifetime specifier
--> dangle.rs:5:16
   |
5 | fn dangle() -> &String {
   |          ^^^^^^
   |
   = help: this function's return type contains a borrowed value, but there is
no
   value for it to be borrowed from
   = help: consider giving it a 'static lifetime

error: aborting due to previous error
```

이 오류 메세지는 우리가 아직 다루지 못한 특성을 인용하고 있습니다: 바로 **라이프타임(lifetime)**입니다. 라이프타임에 대한 것은 10장에서 자세히 다룰 것입니다. 하지만 여러분이 라이프타임에 대한 부분을 무시한다면, 이 메세지는 이 코드가 왜 문제인지를 알려줄 열쇠를 줬고 있습니다:

this function's return type contains a borrowed value, but there is no value
for it to be borrowed from.
(해석: 이 함수의 반환 타입은 빌린 값을 포함하고 있는데, 빌려온 실제 값은 없습니다.)

dangle 코드 부분의 각 단계에서 어떤 일이 벌어지는지 더 면밀히 들여다봅시다:

```
fn dangle() -> &String { // dangle은 String의 참조자를 반환합니다
    let s = String::from("hello"); // s는 새로운 String입니다
    &s // 우리는 String s의 참조자를 반환합니다.
} // 여기서 s는 스코프를 벗어나고 버려집니다. 이것의 메모리는 사라집니다.
// 위험하군요!
```

s가 **dangle** 안에서 만들어졌기 때문에, **dangle**의 코드가 끝이나면 **s**는 할당 해제됩니다. 하지만 우리는 이것의 참조자를 반환하려고 했습니다. 이는 곧 이 참조자가 어떤 무효화된 **String**을 가리키게 될 것이라 뜻이 아닙니까! 별로 안 좋죠. 러스트는 우리가 이런 짓을 못하게 합니다.

여기서의 해법은 **String**을 직접 반환하는 것입니다:

```
fn no_dangle() -> String {
    let s = String::from("hello");
    s
}
```

이 코드는 아무런 문제없이 동작합니다. 소유권이 밖으로 이동되었고, 아무것도 할당 해제되지 않습니다.

참조자의 규칙

우리가 참조자에 대해 논의한 것들을 정리해 봅시다:

1. 어떠한 경우이든 간에, 여러분은 아래 둘 다는 아니고 둘 중 하나만 가질 수 있습니다:
 - 하나의 가변 참조자
 - 임의 개수의 불변 참조자들
2. 참조자는 항상 유효해야만 한다.

다음으로, 우리는 다른 종류의 참조자인 슬라이스(slice)를 볼 것입니다.

슬라이스(Slices)

소유권을 갖지 않는 또 다른 데이터 타입은 **슬라이스**입니다. 슬라이스는 여러분이 컬렉션(collection) 전체가 아닌 컬렉션의 연속된 일련의 요소들을 참조할 수 있게 합니다.

여기 작은 프로그래밍 문제가 있습니다: 스트링을 입력 받아 그 스트링에서 찾은 첫번째 단어를 반환하는 함수를 작성하세요. 만일 함수가 공백문자를 찾지 못한다면, 이는 전체 스트링이 한 단어라는 의미이고, 이때는 전체 스트링이 반환되어야 합니다.

이 함수의 시그니처(signature)에 대해 생각해봅시다:

```
fn first_word(s: &String) -> ?
```

이 함수 `first_word`는 `&String`을 파라미터로 갖습니다. 우리는 소유권을 원하지 않으므로, 이렇게 해도 좋습니다. 하지만 뭘 반환해야 할까요? 우리는 스트링의 일부에 대해 표현할 방법이 없습니다. 하지만 단어의 끝부분의 인덱스를 반환할 수는 있겠습니다. Listing 4-10의 코드처럼 시도해 봅시다:

Filename: src/main.rs

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

Listing 4-10: `String` 파라미터의 바이트 인덱스 값을 반환하는 `first_word` 함수

이 코드를 쪼개서 봅시다. 입력된 `String`를 요소별로 보면서 그 값이 공백인지 확인할 필요가 있기 때문에, `String`은 `as_bytes` 메소드를 이용하여 바이트 배열로 변환됩니다:

```
let bytes = s.as_bytes();
```

다음으로, `iter` 메소드를 이용하여 바이트 배열의 반복자(iterator)를 생성합니다:

```
for (i, &item) in bytes.iter().enumerate() {
```

반복자에 대한 것은 13장에서 더 자세히 다루겠습니다. 지금은 `iter`가 컬렉션의 각 요소를 반환하는 함수이며, `enumerate`은 `iter`의 결과값을 직접 반환하는 대신 이를 감싸서 튜플의 일부로 만들어 반환한다는 정도만 알아두세요. 반환된 튜플의 첫번째 요소는 인덱스이며, 두번째 요소는 요소에 대한 참조값입니다. 이는 우리 스스로 인덱스를 계산하는 것보다 조금 더 편리합니다.

`enumerate` 메소드가 튜플을 반환하기 때문에, 우리는 러스트의 다른 모든 부분에서 그려하듯이 이 튜플을 해체하기 위해 패턴을 이용할 수 있습니다. 따라서 `for` 루프 내에서, `i`는 튜플 내의 인덱스에 대응하고 `&item`은 튜플 내의 한 바이트에 대응하는 패턴을 기술한 것입니다. `.iter().enumerate()`의 요소에 대한 참조자를 갖는 것이므로, `&`을 패턴 내에 사용했습니다.

우리는 바이트 리터럴 문법을 이용하여 공백 문자를 나타내는 바이트를 찾습니다. 공백 문자를 찾았다면, 이 위치를 반환합니다. 그렇지 않으면 `s.len()`을 통해 스트링의 길이값을 반환합니다:

```
if item == b' ' {
    return i;
}
s.len()
```

이제 우리에게 스트링의 첫번째 단어의 끝부분의 인덱스를 찾아낼 방법이 생겼습니다. `usize`를 그대로 반환하고 있지만, 이는 `&String`의 내용물 내에서만 의미가 있습니다. 바꿔 말하면, 이것이 `String`로부터 분리되어 있는 숫자이기 때문에, 이것이 나중에도 여전히 유효한지를 보장할 길이 없습니다. Listing 4-10의 `first_word` 함수를 사용하는 Listing 4-11의 프로그램을 보시죠:

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word는 5를 갖게 될 것입니다.

    s.clear(); // 이 코드는 String을 비워서 ""로 만들게 됩니다.

    // word는 여기서 여전히 5를 갖고 있지만, 5라는 값을 의미있게 쓸 수 있는 스트링은 이제 없습니다.
    // word는 이제 완전 유효하지 않습니다!
}
```

Listing 4-11: `first_word` 함수를 호출하여 결과를 저장한 뒤 `String`의 내용물을 바꾸기

이 프로그램은 아무런 오류 없이 컴파일되고, `s.clear()`을 호출한 뒤 `word`를 사용한다 해도 역시 컴파일될 것입니다. `word`는 `s`의 상태와 전혀 연결되어 있지 않으므로, `word`는 여전히 값 `5`를 담고 있습니다. 우리는 첫번째 단어를 추출하고자 하기 위해 `s`와 값 `5`를 사용할 수 있지만, `word`에 `5`를 저장한 뒤

`s`의 내용물이 변경되었기 때문에 이러한 사용은 버그가 될 것입니다.

`word`의 인덱스가 `s`의 데이터와 싱크가 안맞을 것을 걱정하는 건 지겹고 쉽게 발생할 수 있는 오류입니다! 이러한 인덱스들을 관리하는 것은 우리가 `second_word` 함수를 작성했을 때 더더욱 다루기 어려워집니다. 이 함수의 시그니처는 아래와 같은 모양이 되어야 할 것입니다:

```
fn second_word(s: &String) -> (usize, usize) {
```

이제 우리는 시작, 그리고 끝 인덱스를 추적하고 있고, 특정 상태에 있는 데이터로부터 계산되었지만 그 상태와 전혀 묶여있지 않은 더 많은 값들을 갖게 됩니다. 이제 우리는 동기화를 유지할 필요가 있는 주위를 떠다니는 세 개의 관련없는 변수들을 갖게 되었습니다.

운좋게도, 리스트는 이러한 문제에 대한 해결책을 갖고 있습니다: 바로 스트링 슬라이스(string slice)입니다.

스트링 슬라이스

스트링 슬라이스는 `String`의 일부분에 대한 참조자고, 아래와 같이 생겼습니다:

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```

이는 전체 `String`의 참조자를 갖는 것과 비슷하지만, 추가적으로 `[0..5]`라는 코드가 붙어 있습니다. 전체 `String`에 대한 참조자 보다는, `String`의 일부분에 대한 참조자입니다. `start..end` 문법은 `start`부터 시작하여 `end`를 포함하지 않는 연속된 범위를 기술합니다.

우리는 대괄호 내에 `[starting_index..ending_index]`를 특정한 범위를 이용하여 슬라이스를 만들 수 있는데, 여기서 `starting_index`는 슬라이스에 포함되는 첫번째 위치이고 `ending_index`는 슬라이스에 포함될 마지막 위치보다 1을 더한 값입니다. 내부적으로 슬라이스 데이터 구조는 시작 위치와 슬라이스의 길이를 저장하는데, 이 길이 값은 `ending_index`에서 `starting_index`를 뺀 값입니다. 따라서 `let world = &[6..11];`의 경우, `world`는 `s`의 6번째 바이트를 가리키고 있는 포인터와 길이값 5를 갖고 있는 슬라이스가 될 것입니다.

Figure 4-12는 이를 다이어그램으로 보여줍니다.

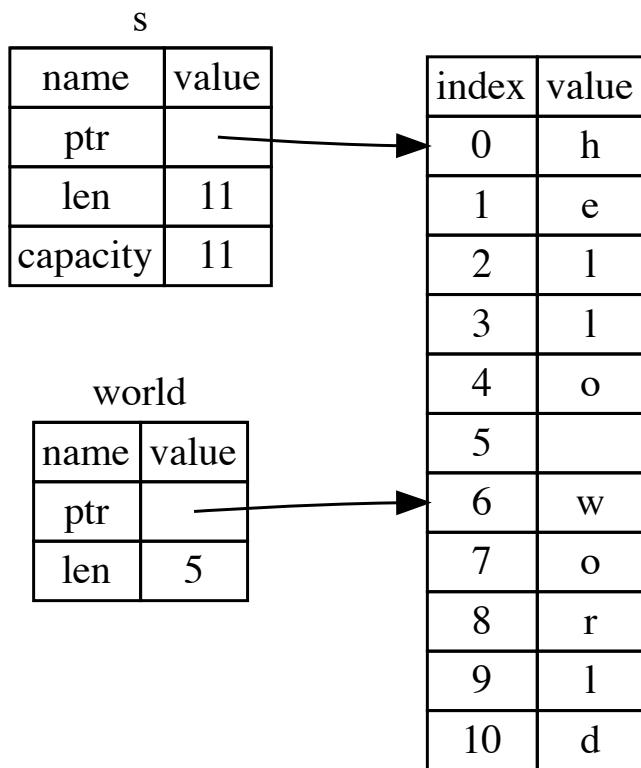


Figure 4-12: `String`의 일부를 참조하는 스트링 슬라이스

러스트의 `..` 범위 문법을 사용하여, 여러분이 만일 첫번째 인덱스(즉 0)에서부터 시작하길 원한다면, 두 개의 마침표 전의 값은 생략할 수 있습니다. 다시 말하면, 아래의 두 줄은 동일한 표현입니다:

```
let s = String::from("hello");
let slice = &s[0..2];
let slice = &s[..2];
```

비슷한 이치로, 만일 여러분의 슬라이스가 `String`의 마지막 바이트까지 포함한다면, 여러분은 끝의 숫자를 생략할 수 있습니다. 이는 아래 두 줄의 표현이 동일하다는 의미입니다:

```
let s = String::from("hello");
let len = s.len();
let slice = &s[3..len];
let slice = &s[3..];
```

여러분은 또한 전체 스트링의 슬라이스를 만들기 위해 양쪽 값을 모두 생략할 수 있습니다. 따라서 아래 두 줄의 표현은 동일합니다:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

이 모든 정보를 잘 기억하시고, `first_word`가 슬라이스를 반환하도록 다시 작성해봅시다. “스트링 슬라이스”를 나타내는 타입은 `&str`로 씁니다:

Filename: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

우리는 Listing 4-10에서 작성한 것과 같은 방법으로 공백 문자가 첫번째로 나타난 지점을 찾아서 단어의 끝 인덱스를 얻어냅니다. 공백 문자를 찾으면, 스트링의 시작과 공백 문자의 인덱스를 각각 시작과 끝 인덱스로 사용하여 스트링 슬라이스를 반환합니다.

이제 `first_word`가 호출되면, 해당 데이터와 묶여있는 하나의 값을 반환받게 되었습니다. 이 값은 슬라이스의 시작 위치에 대한 참조자와 슬라이스의 요소 개수로 이루어져 있습니다.

`second_word` 함수에 대해서도 마찬가지로 슬라이스를 반환하는 형식이 잘 동작할 것입니다:

```
fn second_word(s: &String) -> &str {
```

우리는 이제 영망이 되기 훨씬 힘든 직관적인 API를 갖게 되었는데, 이는 컴파일러가 `String`에 대한 참조자들이 유효한 상태로 남아있게끔 보장할 것이기 때문입니다. 첫번째 단어의 끝 인덱스를 찾았지만, 그 후 스트링을 비워버려서 인덱스가 유효하지 않게되는 Listing 4-11의 프로그램 내의 버그를 기억하시나요? 그런 코드는 논리적으로 맞지 않지만 어떠한 즉각적인 오류도 보여주지 못합니다. 그런 문제는 우리가 비어 있는 스트링에 대해 첫번째 단어의 인덱스를 사용하고자 시도할 경우에나 나타나게 될 것입니다. 슬라이스는 이러한 버그를 불가능하게 만들고 우리가 코드 내에서 발생할 수 있는 문제를 훨씬 일찍 알게 해줍니다.

`first_word`의 슬라이스 버전을 이용하는 것은 컴파일 타임 오류를 발생시킬 것입니다:

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // Error!

    println!("the first word is: {}", word);
}
```

여기 컴파일 오류 메세지를 보시죠:

```
17:6 error: cannot borrow `s` as mutable because it is also borrowed as
          immutable [E0502]
    s.clear(); // Error!
    ^
15:29 note: previous borrow of `s` occurs here; the immutable borrow prevents
          subsequent moves or mutable borrows of `s` until the borrow ends
    let word = first_word(&s);
                ^
18:2 note: previous borrow ends here
fn main() {

}
^
```

빌림 규칙에서 우리가 만일 무언가에 대한 불변 참조자를 만들었을 경우, 가변 참조자를 만들 수 없다는 점을 상기해보세요. `clear` 함수가 `String`을 잘라낼 필요가 있기 때문에, 이 함수는 가변 참조자를 갖기 위한 시도를 할 것이고, 이는 실패하게 됩니다. 러스트는 우리의 API를 사용하기 쉽게 해줄 뿐만 아니라 이러한 종류의 오류 전체를 컴파일 타임에 제거해 줍니다!

스트링 리터럴은 슬라이스입니다

스트링 리터럴이 바이너리 안에 저장된다고 하는 얘기를 상기해봅시다. 이제 슬라이스에 대해 알았으니, 우리는 스트링 리터럴을 적합하게 이해할 수 있습니다:

```
let s = "Hello, world!";
```

여기서 `s`의 타입은 `&str`입니다: 이것은 바이너리의 특정 지점을 가리키고 있는 슬라이스입니다. 이는 왜 스트링 리터럴이 불변인가도 설명해줍니다; `&str`은 불변 참조자이기 때문입니다.

파라미터로서의 스트링 슬라이스

여러분이 리터럴과 `String`의 슬라이스를 얻을 수 있다는 것을 알게 되었다면 `first_word` 함수를 한번 더 개선시킬 수 있는데, 바로 이 함수의 시그니처입니다:

```
fn first_word(s: &String) -> &str {
```

더 경험이 많은 러스트인이라면 대신 아래와 같이 작성하는데, 그 이유는 `&String`과 `&str` 둘 모두에 대한 같은 함수를 사용할 수 있도록 해주기 때문입니다.

```
fn first_word(s: &str) -> &str {
```

만일 우리가 스트링 슬라이스를 갖고 있다면, 이를 바로 넘길 수 있습니다. `String`을 갖고 있다면, 이 `String`의 전체 슬라이스를 넘길 수 있습니다. 함수가 `String`의 참조자 대신 스트링 슬라이스를 갖도록 정의하는 것은 우리의 API를 어떠한 기능적인 손실 없이도 더 일반적이고 유용하게 해줍니다:

Filename: src/main.rs

```
fn main() {
    let my_string = String::from("hello world");

    // first_word가 `String`의 슬라이스로 동작합니다.
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word가 스트링 리터럴의 슬라이스로 동작합니다.
    let word = first_word(&my_string_literal[..]);

    // 스트링 리터럴은 *또한* 스트링 슬라이스이기 때문에,
    // 아래 코드도 슬라이스 문법 없이 동작합니다!
    let word = first_word(my_string_literal);
}
```

그 밖의 슬라이스들

스트링 슬라이스는 여러분이 상상하는 바와 같이, 스트링에 특정되어 있습니다. 하지만 더 일반적인 슬라이스 타입도 역시 있습니다. 아래 배열을 보시죠:

```
let a = [1, 2, 3, 4, 5];
```

우리가 스트링의 일부를 참조하고 싶어할 수 있는 것처럼, 배열의 일부를 참조하고 싶을 수 있고, 그러면 아래와 같이 쓸 수 있습니다:

```
let a = [1, 2, 3, 4, 5];  
  
let slice = &a[1..3];
```

이 슬라이스는 `&[i32]` 타입을 갖습니다. 이는 스트링 슬라이스가 동작하는 방법과 똑같이, 슬라이스의 첫 번째 요소에 대한 참조자와 슬라이스의 길이를 저장하는 방식으로 동작합니다. 여러분은 다른 모든 종류의 컬렉션들에 대하여 이런 종류의 슬라이스를 이용할 수 있습니다. 벡터에 대해서 8장에서 이야기할 때 이러한 컬렉션에 대해 더 자세히 다루겠습니다.

정리

소유권, 빌림, 그리고 슬라이스의 개념은 러스트 프로그램의 메모리 안정성을 컴파일 타임에 보장하는 것입니다. 러스트 언어는 다른 시스템 프로그래밍 언어와 같이 여러분의 메모리 사용에 대한 제어권을 주지만, 데이터의 소유자가 스코프 밖으로 벗어났을 때 소유자가 자동적으로 데이터를 버리도록 하는 것은 곧 여러분이 이러한 제어를 위해 추가적인 코드 작성이나 디버깅을 하지 않아도 된다는 뜻입니다.

소유권은 러스트의 다른 수많은 부분이 어떻게 동작하는지에 영향을 주므로, 이 책의 남은 부분 전체에 걸쳐 이 개념들에 대해 더 이야기할 것입니다. 다음 장으로 넘어가서 데이터들을 함께 그룹짓는 `struct`를 보겠습니다.

연관된 데이터들을 구조체로 다루기

구조체(*struct*)는 사용자들이 연관된 여러 값을 묶어서 의미있는 데이터 단위를 정의할 수 있게 합니다. 객체지향 언어를 사용해 본 경험이 있으시다면, 구조체(*struct*)는 객체의 데이터 속성 같은 것으로 보시면 됩니다. 이번 장에서는 튜플과 구조체를 비교해 보고, 구조체를 어떻게 사용하는지 알아보며, 메소드와 구조체 데이터의 동작과 관련된 연관함수(*associated functions*)의 정의 방법에 대해 알아보도록 하겠습니다. 구조체와 열거형(6장에서 살펴볼 것입니다)에 대한 개념은 여러분의 프로그램 도메인 상에서 새로운 타입을 만들기 위한 기초 재료로서, 러스트의 컴파일 시점 타입 검사 기능을 최대한 활용합니다.

구조체를 정의하고 초기화하기

구조체는 3장에서 학습한 튜플과 비슷합니다. 튜플과 유사하게, 구조체의 구성요소들은 각자 다른 타입을 지닐 수 있습니다. 그러나 튜플과는 다르게 각 구성요소들은 명명할 수 있어 값이 의미하는 바를 명확하게 인지 할 수 있습니다. 구조체는 각 구성요소들에 명명을 할 수 있다는 점 덕분에 튜플보다 유연하게 다룰 수 있습니다. 구조체 내의 특정 요소 데이터 명세를 기술하거나, 접근할 때 순서에 의존할 필요가 없기 때문입니다.

구조체를 정의할 때는 `struct` 키워드를 먼저 입력하고 명명할 구조체명을 입력하면 됩니다. 구조체의 이름은 함께 묶이게 되는 구성요소들의 의미를 내포할 수 있도록 짓는 것이 좋습니다. 이후 중괄호 안에서는, 필드(*field*)라 불리는 각 구성요소들의 타입과 접근할 수 있는 이름을 정의합니다.

아래 예제 5-1에서는 사용자 계정에 대한 정보를 저장하는 구조체를 정의합니다.

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

Listing 5-1: 사용자 계정정보를 저장하는 `User` 구조체 정의

정의한 구조체를 사용하려면, 각 필드의 값을 명세한 인스턴스(*instance*)를 생성해야 합니다. 인스턴스는 구조체의 이름을 명시함으로써 사용할 수 있고, 필드를 식별할 수 있는 이름인 키와 그 키에 저장하고자 하는 값의 쌍(`key:value`)을 이어지는 중괄호 안에 추가하여 생성할 수 있습니다.

구조체를 정의할 때 필드들의 순서가 정의한 필드의 순서와 같을 필요는 없습니다. 달리 서술하자면, 구조체 정의는 무엇이 들어가야 하는지 대략적으로 정의된 양식 정도라고 생각하시면 되고, 인스턴스는 그것에 특정한 값을 넣어 실체화한 것이라 생각하시면 됩니다. 아래 예제 5-2에서는 특정 사용자를 선언하는 과정을 보여 줍니다.

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

Listing 5-2: 구조체 `User`의 인스턴스 생성하기

구조체에서 특정한 값을 읽어오려면, 점(.) 표기법을 사용하시면 됩니다. 사용자의 이메일 값을 얻고자 하면, `user1.email` 과 같은 방식으로 접근하실 수 있습니다. 변경이 가능한 구조체 인스턴스에 들어있는 값을

바꾸고자 할 때는, 점(.) 표기법을 사용하여 특정 필드에 새 값을 할당할 수 있습니다.

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```

Listing 5-3: `User` 인스턴스의 `email` 필드 변경하기

인스턴스는 반드시 **변경 가능(mutable)** 해야합니다. Rust에서는 특정 필드만 변경할 수 있도록 허용하지 않습니다. 다른 표현식과 마찬가지로, 함수 본문의 마지막에 새 인스턴스 구조체를 표현식(expressions)으로 생성하여 새 인스턴스를 바로 반환 할 수 있습니다.

Listing 5-4에서는 주어진 `email`과 `user_name`으로 `User` 인스턴스를 반환하는 `build_user` 함수를 보여줍니다. 활성 필드는 `true` 값을 가져오고 `sign_in_count`는 `1` 값을 가져옵니다.

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

예제 5-4: 사용자의 이메일과 이름을 받아 `User` 구조체의 인스턴스를 반환하는 `build_user` 함수

구조체 필드와 동일한 이름으로 함수 매개 변수의 이름을 지정하는 것이 합리적이긴 하지만, `email` 및 `username` 필드 이름과 변수를 반복해야하는 것은 비효율적입니다. 구조체에 더 많은 필드가 많다면, 더욱 성가실 것입니다. 다행히도 편리한 방법이 있습니다!

변수명이 필드명과 같을 때 간단하게 필드 초기화하기

변수명과 구조체의 필드명이 같다면, 필드 초기화 축약법(*field init shorthand*)을 이용할 수 있습니다. 이를 활용하면 구조체를 생성하는 함수를 더 간단히 작성할 수 있게 됩니다. 아래 예제 5-5의 `build_user` 함수에는 `email`과 `username`라는 매개변수가 있습니다. 함수는 `User` 구조체가 구현된 인스턴스를 반환합니다.

매개변수인 `email`과 `username`이 `User` 구조체의 필드명과 같기 때문에, 함수 `build_user`에서 `email`과 `username`를 명시하는 부분을 예제 5-4와 같이 다시 작성할 필요가 없습니다.

예제 5-5의 `build_user` 함수는 예제 5-4와 같은 방식으로 동작합니다. 필드 초기화를 이러한 방식으로 수행하는 문법은 간결한 코드를 작성하는데 도움이 되고, 많은 필드의 값이 정의되어야 할 때 특히 유용합니다.

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

예제 5-5: 매개변수 `email`과 `username` 가 구조체의 필드와 이름이 같아, 함수 내에서 특별히 명시하지 않고 초기화한 예인 `build_user` 함수

`email` 필드와 `email` 매개 변수의 이름이 같기 때문에 `email:email` 대신 `email` 만 작성하면 됩니다!

구조체 갱신법을 이용하여 기존 구조체 인스턴스로 새 구조체 인스턴스 생성하기

존재하는 인스턴스에서 기존 값의 대부분은 재사용하고, 몇몇 값만 바꿔 새로운 인스턴스를 정의하는 방법은 유용합니다. 예제 5-6는 변수 `user2`에 `email`과 `username`은 새로 할당하고, 나머지 필드들은 예제 5-2에서 정의한 `user1`의 값을 그대로 사용하는 방식으로 `User` 인스턴스를 생성하는 것을 보여줍니다.

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

예제 5-6: `user1`의 일부 값을 재사용하여, 구조체 `User`의 인스턴스 `user2`를 새로 생성

구조체 갱신법(*struct update syntax*)은 예제 5-6에서 작성한 짧은 코드와 같은 효과를 낼 수 있습니다. 구조체 갱신법은, 입력으로 주어진 인스턴스와 변화하지 않는 필드들을 명시적으로 할당하지 않기 위해 `..` 구문을 사용합니다. 예제 5-7의 코드는 `user1` 인스턴스와 `active`, `sign_in_count` 필드의 값은 같고, `email`과 `username` 필드들은 같은 다른 `user2` 인스턴스를 생성할 때 구조체 갱신법을 사용하는 것을 보여줍니다.

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

예제 5-7: 인스턴스 갱신 문법의 사용 예시 - 새 `User` 구조체 생성 시 `email`과 `username` 필드에는 새 값을 할당하고, 나머지 필드는 `user1`에서 재사용

이름이 없고 필드마다 타입은 다르게 정의 가능한 튜플 구조체

구조체명을 통해 의미를 부여할 수 있으나 필드의 타입만 정의할 수 있고 명명은 할 수 없는, 튜플 구조체 (*tuple structs*)라 불리는 튜플과 유사한 형태의 구조체도 정의할 수 있습니다.

튜플 구조체는 일반적인 구조체 정의방법과 똑같이 `struct` 키워드를 통해 정의할 수 있고, 튜플의 타입 정의가 키워드 뒤에서 이루어지면 됩니다. 아래는 튜플 구조체인 `Color`, `Point`의 정의와 사용 예시입니다.

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

다른 튜플 구조체이기 때문에, `black`과 `origin`이 다른 타입이란 것을 유념해 두셔야 합니다. 구조체 내의 타입이 모두 동일하더라도 각각의 구조체는 고유의 타입이기 때문입니다. 한편 튜플 구조체 인스턴스는, 3장에서 살펴 본 튜플과 비슷하게 동작합니다.

필드가 없는 유사 유닛 구조체

또한 어떤 필드도 없는 구조체 역시 정의할 수 있습니다! 이는 유닛 타입인 `()`와 비슷하게 동작하고, 그 때문에 유사 유닛 구조체(*unit-like structs*)라 불립니다. 유사 유닛 구조체는 특정한 타입의 트레잇(trait)을 구현해야하지만 타입 자체에 데이터를 저장하지 않는 경우에 유용합니다. 트레잇(trait)에 대해서는 10장에서 더 살펴보도록 하겠습니다.

구조체 데이터의 소유권(Ownership)

예제 5-1에서의 `User` 구조체 정의에서는, `&str` 문자 슬라이스 타입 대신 `String` 타입을 사용했

습니다. 이는 의도적인 선택으로, 구조체 전체가 유효한 동안 구조체가 그 데이터를 소유하게 하고자 합니다.

구조체가 소유권이 없는 데이터의 참조를 저장할수는 있지만, 10장에서 언급 될 라이프타임 (*lifetimes*)의 사용을 전제로 합니다. 라이프타임은 구조체가 존재하는동안 참조하는 데이터를 계속 존재할 수 있도록 합니다. 라이프타임을 사용하지 않고 참조를 저장하고자 하면 아래와 같은 일이 발생 합니다.

Filename: src/main.rs

```
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

컴파일러는 라이프타임이 명시되어야 한다고 에러를 발생시킵니다.

```
error[E0106]: missing lifetime specifier
-->
|
2 |     username: &str,
|             ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
-->
|
3 |     email: &str,
|             ^ expected lifetime parameter
```

참조가 저장이 불가능한 위 에러 개선에 대해서는 10장에서 살펴보도록 하겠습니다. 지금은 `&str` 대신 `String` 을 사용하는 방식으로 에러를 고치도록 하겠습니다.

구조체를 이용한 예제 프로그램

어느 시점에 구조체를 이용하기를 원하게 될지를 이해해보기 위해서, 사각형의 넓이를 계산하는 프로그램을 작성해봅시다. 단일 변수들로 구성된 프로그램으로 시작한 뒤, 이 대신 구조체를 이용하기까지 프로그램을 리팩토링해 볼 것입니다.

Cargo로 픽셀 단위로 명시된 사각형의 길이와 너비를 입력받아서 사각형의 넓이를 계산하는 *rectangles*라는 이름의 새로운 바이너리 프로젝트를 만듭시다. Listing 5-7은 우리 프로젝트의 *src/main.rs* 내에 설명한 동작을 수행하는 한 방법을 담은 짧은 프로그램을 보여줍니다:

Filename: *src/main.rs*

```
fn main() {
    let length1 = 50;
    let width1 = 30;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(length1, width1)
    );
}

fn area(length: u32, width: u32) -> u32 {
    length * width
}
```

Listing 5-7: 길이와 너비가 각각의 변수에 지정된 사각형의 넓이 계산하기

이제 이 프로그램을 `cargo run`으로 실행해보세요:

```
The area of the rectangle is 1500 square pixels.
```

튜플을 이용한 리팩터링

비록 Listing 5-7가 잘 동작하고 각 차원축의 값을 넣은 `area` 함수를 호출함으로써 사각형의 넓이를 알아냈을지라도, 이것보다 더 좋게 할 수 있습니다. 길이와 너비는 함께 하나의 사각형을 기술하기 때문에 서로 연관되어 있습니다.

이 방법에 대한 사안은 `area`의 시그니처에서 여실히 나타납니다:

```
fn area(length: u32, width: u32) -> u32 {
```

area 함수는 어떤 사각형의 넓이를 계산하기로 되어있는데, 우리가 작성한 함수는 두 개의 파라미터들을 가지고 있습니다. 파라미터들은 연관되어 있지만, 우리 프로그램 내의 어디에도 표현된 바 없습니다. 길이와 너비를 함께 묶는다면 더 읽기 쉽고 관리하기도 좋을 것입니다. 페이지 XX, 3장의 튜플로 값들을 묶기 절에서 이런 일을 하는 한가지 방법을 이미 다루었습니다: 바로 튜플을 이용하는 것이지요. Listing 5-8은 튜플을 이용한 우리 프로그램의 또다른 버전을 보여줍니다:

Filename: src/main.rs

```
fn main() {
    let rect1 = (50, 30);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Listing 5-8: 튜플을 이용하여 사각형의 길이와 너비를 명시하기

어떤 면에서는 프로그램이 더 좋아졌습니다. 튜플은 한 조각의 구조체를 추가할 수 있게 해주고, 우리는 이제 단 하나의 인자만 넘기게 되었습니다. 그러나 다른 한편으로 이 버전은 덜 명확합니다: 튜플은 요소에 대한 이름이 없어서, 튜플 내의 값을 인덱스로 접근해야 하기 때문에 우리의 계산이 더 혼란스러워 졌습니다.

면적 계산에 대해서는 길이와 너비를 혼동하는 것이 큰 문제가 아니겠으나, 만일 우리가 화면에 이 사각형을 그리고 싶다면, 문제가 됩니다! 우리는 `length`가 튜플 인덱스 `0`이고 `width`가 튜플 인덱스 `1`이라는 점을 꼭 기억해야 할 것입니다. 만일 다른 누군가가 이 코드를 이용해서 작업한다면, 그들 또한 이 사실을 알아내어 기억해야 할테지요. 우리의 코드 내에 데이터의 의미를 전달하지 않았기 때문에, 이 값을 잊어먹거나 혼동하여 에러를 발생시키는 일이 쉽게 발생할 것입니다.

구조체를 이용한 리팩터링: 의미를 더 추가하기

우리는 데이터에 이름표를 붙여 의미를 부여하기 위해 구조체를 이용합니다. Listing 5-9에서 보시는 바와 같이, 우리가 사용중인 튜플은 전체를 위한 이름 뿐만 아니라 부분들을 위한 이름들도 가지고 있는 데이터 타입으로 변형될 수 있습니다:

Filename: src/main.rs

```

struct Rectangle {
    length: u32,
    width: u32,
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.length * rectangle.width
}

```

Listing 5-9: `Rectangle` 구조체 정의하기

여기서 우리는 구조체를 정의하고 이를 `Rectangle`이라 명명했습니다. `{}` 안에서 `length`와 `width`를 필드로 정의했는데, 둘 모두 `u32` 타입입니다. 그런 다음 `main` 함수 안에서 길이 50 및 너비 30인 특정한 `Rectangle` 인스턴스(instance)를 생성했습니다.

우리의 `area` 함수는 이제 하나의 파라미터를 갖도록 정의되었는데, 이는 `rectangle`이라는 이름이고, `Rectangle` 구조체 인스턴스의 불변 참조자 타입입니다. 4장에서 언급했듯이, 우리는 구조체의 소유권을 얻기 보다는 빌리기를 원합니다. 이 방법으로, `main`은 그 소유권을 유지하고 `rect1`을 계속 이용할 수 있는데, 이는 우리가 함수 시그니처 내에서와 함수 호출시에 `&`를 사용하게 된 이유입니다.

`area` 함수는 `Rectangle` 인스턴스 내의 `length`와 `width` 필드에 접근합니다. `area`에 대한 우리의 함수 시그니처는 이제 정확히 우리가 의미한 바를 나타냅니다: `length`와 `width` 필드를 사용하여 `Rectangle`의 넓이를 계산한다는 뜻 말이죠. 이는 길이와 너비가 서로 연관되어 있음을 잘 전달하며, `0`과 `1`을 사용한 튜플 인덱스 값을 이용하는 대신에 값들에 대해서 서술적인 이름을 사용합니다 - 명확성 측면에서 승리입니다.

파생 트레이트(derived trait)으로 유용한 기능 추가하기

우리가 프로그램을 디버깅하는 동안 구조체 내의 모든 값을 보기 위해서 `Rectangle`의 인스턴스를 출력할 수 있다면 도움이 될 것입니다. Listing 5-10은 우리가 이전 장들에서 해왔던 것처럼 `println!` 매크로를 이용한 것입니다:

Filename: src/main.rs

```

struct Rectangle {
    length: u32,
    width: u32,
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!("rect1 is {}", rect1);
}

```

Listing 5-10: `Rectangle` 인스턴스 출력 시도하기

이 코드를 실행시키면, 다음과 같은 핵심 메세지와 함께 에러가 발생합니다:

```
error[E0277]: the trait bound `Rectangle: std::fmt::Display` is not satisfied
```

`println!` 매크로는 다양한 종류의 포맷을 출력할 수 있으며, 기본적으로 `{}`은 `println!`에게 `Display`라고 알려진 포맷팅을 이용하라고 전달해줍니다: 직접적인 최종 사용자가 사용하도록 의도된 출력이지요. 여지껏 우리가 봄은 기본 타입들은 `Display` 가 기본적으로 구현되어 있는데, 이는 1 혹은 다른 기본 타입을 유저에게 보여주고자 하는 방법이 딱 한 가지기 때문입니다. 하지만 구조체를 사용하는 경우, `println!`이 출력을 형식화하는 방법은 덜 명확한데 이는 표시 방법의 가능성이 더 많기 때문입니다: 여러분은 쉽표를 이용하길 원하나요, 혹은 그렇지 않은가요? 여러분은 중괄호를 출력하길 원하나요? 모든 필드들이 다 보여지는 편이 좋은가요? 이러한 모호성 때문에, 러스트는 우리가 원하는 것을 추론하는 시도를 하지 않으며 구조체는 `Display`에 대한 기본 제공 되는 구현체를 가지고 있지 않습니다.

계속 에러를 읽어나가면, 아래와 같은 도움말을 찾게 될 것입니다:

```
note: `Rectangle` cannot be formatted with the default formatter; try using
`{:?}` instead if you are using a format string
```

한번 시도해보죠! `println!` 매크로 호출은 이제 `println!("rect1 is {:?}", rect1);`; 처럼 보이게 될 것입니다. `{}` 내에 `:?` 명시자를 집어넣는 것은 `println!`에게 `Debug`라 불리우는 출력 포맷을 사용하고 싶다고 말해줍니다. `Debug`는 개발자에게 유용한 방식으로 우리의 구조체를 출력할 수 있도록 해줘서 우리 코드를 디버깅 하는 동안 그 값을 볼수 있게 해주는 트레이잇입니다.

이 변경을 가지고 코드를 실행해보세요. 젠장! 여전히 에러가 납니다:

```
error: the trait bound `Rectangle: std::fmt::Debug` is not satisfied
```

하지만 또다시, 컴파일러가 우리에게 도움말을 제공합니다:

```
note: `Rectangle` cannot be formatted using `{:?}`; if it is defined in your
crate, add `#[derive(Debug)]` or manually implement it
```

러스트는 디버깅 정보를 출력하는 기능을 포함하고 있는 것이 맞지만, 우리 구조체에 대하여 해당 기능을 활성화하도록 명시적인 사전동의를 해주어야 합니다. 그러기 위해서, Listing 5-11에서 보는 바와 같이 구조체 정의부분 바로 전에 `#[derive(Debug)]` 어노테이션을 추가합니다:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    length: u32,
    width: u32,
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!("rect1 is {:?}", rect1);
}
```

Listing 5-11: `Debug` 트레이잇을 파생시키기 위한 어노테이션의 추가 및 디버그 포맷팅을 이용한 `Rectangle` 인스턴스의 출력

이제 프로그램을 실행시키면, 에러는 사라지고 다음과 같은 출력을 보게될 것입니다:

```
rect1 is Rectangle { length: 50, width: 30 }
```

좋아요! 이게 제일 예쁜 출력은 아니지만, 이 인스턴스를 위한 모든 필드의 값을 보여주는데, 이는 디버깅하는 동안 분명히 도움이 될 것입니다. 우리가 더 큰 구조체를 가지게 됐을 때는, 읽기 좀 더 수월한 출력을 쓰는 것이 유용합니다; 그러한 경우, `println!` 스트링 내에 `{:?:}` 대신 `{:#?}`을 사용할 수 있습니다. 예제 내에서 `{:#?}` 스타일을 이용하게 되면, 출력이 아래와 같이 생기게 될 것입니다:

```
rect1 is Rectangle {
    length: 50,
    width: 30
}
```

러스트는 우리를 위해 `derive` 어노테이션을 이용한 여러 트레이잇을 제공하여 우리의 커스텀 타입에 대해 유용한 동작을 추가할 수 있도록 해줍니다. 이 트레이잇들과 그 동작들은 부록 C에서 그 목록을 찾을 수 있습니다. 10장에서는 이 트레이잇들을 커스터마이징된 동작을 수행하도록 구현하는 방법 뿐만 아니라 우리만의 트레이잇을 만드는 방법에 대해 다룰 것입니다.

우리의 `area` 함수는 매우 특정되어 있습니다: 딱 사각형의 면적만 계산합니다. 이 동작을 우리의 `Rectangle` 구조체와 더 가까이 묶을 수 있다면 유용할텐데요, 그 이유는 이 함수가 다른 타입과는 작동하지 않기 때문입니다. `area` 함수를 `Rectangle` 타입 내에 정의된 `area` 메소드로 바꾸어서 이 코드를 어떻게 더 리팩터링할 수 있는지 살펴봅시다.

메소드 문법

메소드(method)는 함수와 유사합니다: 이들은 `fn` 키워드와 이름을 가지고 선언되고, 파라미터와 반환값을 가지고 있으며, 다른 어딘가로부터 호출되었을 때 실행될 어떤 코드를 담고 있습니다. 하지만, 메소드는 함수와는 달리 구조체의 내용 안에 정의되며 (혹은 열거형이나 트레이트 객체 안에 정의되는데, 이는 6장과 17장에서 각각 다루겠습니다), 첫번째 파라미터가 언제나 `self` 인데, 이는 메소드가 호출되고 있는 구조체의 인스턴스를 나타냅니다.

메소드 정의하기

Listing 5-12에서 보는 바와 같이 `Rectangle` 인스턴스를 파라미터로 가지고 있는 `area` 함수를 바꿔서 그 대신 `Rectangle` 구조체 위에서 정의된 `area` 메소드를 만들어 봅시다:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Listing 5-12: `Rectangle` 구조체 상에 `area` 메소드 정의하기

`Rectangle`의 내용 안에 함수를 정의하기 위해서, `impl` (구현: *implementation*) 블록을 시작합니다. 그 다음 `area` 함수를 `impl` 중괄호 안으로 옮기고 시그니처 및 본체 내의 모든 곳에 있는 첫번째 파라미터 (지금의 경우에는 유일한 파라미터)를 `self`로 변경시킵니다. 우리가 `area` 함수를 호출하고 여기에 `rect1`을 인자로 넘기고 있는 `main` 함수에서는, 이 대신 `Rectangle` 인스턴스 상의 `area` 메소드를 호출하기

위해서 메소드 문법(method syntax)를 이용할 수 있습니다. 메소드 문법은 인스턴스 다음에 위치합니다: 점을 추가하고 그 뒤를 이어 메소드 이름, 괄호, 인자들이 따라옵니다.

`area`의 시그니처 내에서는, `rectangle: &Rectangle` 대신 `&self`가 사용되었는데 이는 메소드가 `impl Rectangle` 내용물 안에 위치하고 있어 러스트가 `self`의 타입이 `Rectangle`라는 사실을 알 수 있기 때문입니다. 우리가 `&Rectangle`이라고 썼던 것처럼, `self` 앞에도 여전히 `&`를 사용할 필요가 있음을 주목하세요. 메소드는 `self`의 소유권을 가져갈 수도, 여기서처럼 `self`를 변경 불가능하게 빌릴 수도, 혹은 다른 파라미터와 비슷하게 변경이 가능하도록 빌려올 수도 있습니다.

여기서는 함수 버전에서 `&Rectangle`을 이용한 것과 같은 이유로 `&self`를 택했습니다: 우리는 소유권을 가져오는 것을 원하지 않으며, 다만 구조체 내의 데이터를 읽기만 하고, 쓰고 싶지는 않습니다. 만일 그 메소드가 동작하는 과정에서 메소드 호출에 사용된 인스턴스가 변하기를 원했다면, 첫번째 파라미터로 `&mut self`를 썼을테지요. 그냥 `self`를 첫번째 파라미터로 사용하여 인스턴스의 소유권을 가져오는 메소드를 작성하는 일은 드뭅니다; 이러한 테크닉은 보통 해당 메소드가 `self`을 다른 무언가로 변형시키고 이 변형 이후에 호출하는 측에서 원본 인스턴스를 사용하는 것을 막고 싶을 때 종종 쓰입니다.

함수 대신 메소드를 이용하면 생기는 주요 잇점은, 메소드 문법을 이용하여 모든 메소드 시그니처 내에서마다 `self`의 타입을 반복하여 타이핑하지 않아도 된다는 점과 더불어, 조직화에 관한 점입니다. 우리 코드를 향후 사용할 사람들이 우리가 제공하는 라이브러리 내의 다양한 곳에서 `Rectangle`이 사용 가능한 지점을 찾도록 하는 것보다 하나의 `impl` 블록 내에 해당 타입의 인스턴스로 할 수 있는 모든 것을 모아두었습니다.

→ 연산자는 어디로 갔나요?

C++ 같은 언어에서는, 메소드 호출을 위해서 서로 다른 두 개의 연산자가 사용됩니다: 만일 어떤 객체의 메소드를 직접 호출하는 중이라면 `.`를 이용하고, 어떤 객체의 포인터에서의 메소드를 호출하는 중이고 이 포인터를 역참조할 필요가 있다면 `->`를 쓰지요. 달리 표현하면, 만일 `object`가 포인터라면, `object->something()`은 `(*object).something()`과 비슷합니다.

러스트는 `->` 연산자와 동치인 연산자를 가지고 있지 않습니다; 대신, 러스트는 자동 참조 및 역참조 (*automatic referencing and dereferencing*)이라는 기능을 가지고 있습니다. 메소드 호출은 이 동작을 포함하는 몇 군데 중 하나입니다.

동작 방식을 설명해보겠습니다: 여러분이 `object.something()`이라고 메소드를 호출했을 때, 러스트는 자동적으로 `&`나 `&mut`, 혹은 `*`을 붙여서 `object`가 해당 메소드의 시그니처와 맞도록 합니다. 달리 말하면, 다음은 동일한 표현입니다:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

첫번째 표현이 훨씬 깔끔해 보입니다. 이러한 자동 참조 동작은 메소드가 명확한 수신자-즉 `self`의 타입을 가지고 있기 때문에 동작합니다. 수신자와 메소드의 이름이 주어질 때, 러스트는 해당 메소드가 읽는지 (`&self`) 혹은 변형시키는지 (`&mut self`), 아니면 소비하는지 (`self`)를 결정론적으로 알 아낼 수 있습니다. 러스트가 메소드 수신자를 암묵적으로 빌리도록 하는 사실은 실사용 환경에서 소유권을 인간공학적으로 만드는 중요한 부분입니다.

더 많은 파라미터를 가진 메소드

`Rectangle` 구조체의 두번째 메소드를 구현하여 메소드 사용법을 연습해 봅시다. 이번에는 `Rectangle`의 인스턴스가 다른 `Rectangle` 인스턴스를 가져와서 이 두번째 `Rectangle`이 `self`내에 완전히 안에 들어갈 수 있다면 `true`를 반환하고, 그렇지 않으면 `false`를 반환하고 싶어합니다. 즉, `can_hold` 메소드를 정의했다면, Listing 5-13에서 제시하는 프로그램을 작성할 수 있기를 원합니다:

Filename: src/main.rs

```
fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };
    let rect2 = Rectangle { length: 40, width: 10 };
    let rect3 = Rectangle { length: 45, width: 60 };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Listing 5-13: 아직 작성하지 않은 `can_hold` 메소드를 이용하는 데모

그리고 기대하는 출력은 아래와 같게 될 것인데, 이는 `rect2`의 두 차원축은 모두 `rect1`의 것보다 작지만, `rect3`은 `rect1`에 비해 가로로 더 넓기 때문입니다:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

메소드를 정의하기를 원한다는 것을 인지하고 있으니, 이는 `impl Rectangle` 블록 내에 오게될 것입니다. 메소드의 이름은 `can_hold`이고, 또다른 `Rectangle`의 불변 참조자를 파라미터로 갖을 것입니다. 파라미터의 타입이 어떤 것이 될지는 메소드를 호출하는 코드를 살펴봄으로써 알 수 있습니다:

`rect1.can_hold(&rect2)`는 `&rect2`를 넘기고 있는데, 이는 `Rectangle`의 인스턴스인 `rect2`의 불변성 빌림입니다. 이는 우리가 `rect2`를 그냥 읽기만 하길 원하기 때문에 타당하며 (쓰기를 원하는 것은 아니지요. 이는 곧 가변 빌림이 필요함을 의미합니다), `main`이 `rect2`의 소유권을 유지하여 `can_hold` 메소드 호출 이후에도 이를 다시 사용할 수 있길 원합니다. `can_hold`의 반환값은 부울린이 될 것이고, 이

구현은 `self`의 길이와 너비가 다른 `Rectangle`의 길이와 너비보다 둘다 각각 큰지를 검사할 것입니다. Listing 5-14에서 보는 것처럼, 이 새로운 `can_hold` 메소드를 Listing 5-12의 `impl` 블록에 추가해 봅시다:

Filename: src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

Listing 5-14: 또 다른 `Rectangle` 인스턴스를 파라미터로 갖는 `can_hold` 메소드를 `Rectangle` 상에 구현하기

Listing 5-13에 있는 `main` 함수와 함께 이 코드를 실행하면, 원하는 출력을 얻을 수 있을 것입니다. 메소드는 `self` 파라미터 뒤에 추가된 여러 개의 파라미터를 가질 수 있으며, 이 파라미터들은 함수에서의 파라미터와 동일하게 기능합니다.

연관 함수

`impl` 블록의 또 다른 유용한 기능은 `self` 파라미터를 갖지 않는 함수도 `impl` 내에 정의하는 것이 허용된다는 점입니다. 이를 *연관 함수 (associated functions)*라고 부르는데, 그 이유는 이 함수가 해당 구조체와 연관되어 있기 때문입니다. 이들은 메소드가 아니라 여전히 함수인데, 이는 함께 동작할 구조체의 인스턴스를 가지고 있지 않아서 그렇습니다. 여러분은 이미 `String::from` 연관 함수를 사용해본 적이 있습니다.

연관 함수는 새로운 구조체의 인스턴스를 반환해주는 생성자로서 자주 사용됩니다. 예를 들면, 하나의 차원값 파라미터를 받아서 이를 길이와 너비 양쪽에 사용하여, 정사각형 `Rectangle`을 생성할 때 같은 값을 두번 명시하도록 하는 것보다 쉽게 해주는 연관 함수를 제공할 수 있습니다:

Filename: src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { length: size, width: size }
    }
}
```

이 연관 함수를 호출하기 위해서는 `let sq = Rectangle::square(3);` 처럼, 구조체 이름과 함께 `::` 문법을 이용합니다. 이 함수는 구조체의 이름공간 내에 있습니다: `::` 문법은 연관 함수와 모듈에 의해 생성된 이름공간 두 곳 모두에서 사용되는데, 모듈에 대해서는 7장에서 다룰 것입니다.

정리

구조체는 우리의 문제 영역에 대해 의미있는 커스텀 타입을 만들수 있도록 해줍니다. 구조체를 이용함으로써, 우리는 연관된 데이터의 조각들을 서로 연결하여 유지할 수 있으며 각 데이터 조각에 이름을 붙여 코드를 더 명확하게 만들어 줄 수 있습니다. 메소드는 우리 구조체의 인스턴스가 가지고 있는 동작을 명시하도록 해주며, 연관 함수는 이용 가능한 인스턴스 없이 우리의 구조체에 특정 기능을 이름공간 내에 넣을 수 있도록 해줍니다.

하지만 구조체가 커스텀 타입을 생성할 수 있는 유일한 방법은 아닙니다: 러스트의 열거형 기능으로 고개를 돌려 우리의 도구상자에 또다른 도구를 추가하도록 합니다.

열거형과 패턴 매칭

이번 장에서는 열거(enumerations)에 대해 살펴볼 것입니다. 열거형(enums)이라고도 합니다. 열거형은 하나의 타입이 가질 수 있는 값들을 열거 함으로써 타입을 정의할 수 있도록 합니다. 우선, 하나의 열거형을 정의하고 사용해 봄으로써, 어떻게 열거형에 의미와 함께 데이터를 담을 수 있는지 보여줄 것입니다. 다음으로, **Option** 이라고 하는 특히 유용한 열거형을 자세히 볼 텐데, 이것은 어떤 값을 가질 수도 있고, 갖지 않을 수도 있습니다. 그다음으로, 열거형의 값에 따라 쉽게 다른 코드를 실행하기 위해 **match** 표현식에서 패턴 매칭을 사용하는 방법을 볼 것입니다. 마지막으로, 코드에서 열거형을 편하고 간결하게 다루기 위한 관용 표현인 **if let** 구문을 다룰 것입니다.

열거형은 다른 언어들에서도 볼 수 있는 특징이지만, 각 언어마다 열거형으로 할 수 있는 것들이 다릅니다. 러스트의 열거형은 F#, OCaml, Haskell과 같은 함수형 언어의 **대수 데이터 타입**과 가장 비슷합니다.

열거형 정의하기

코드를 작성할 때, 열거형이 구조체보다 유용하고 적절하게 사용되는 상황에 대해서 살펴볼 것입니다. IP 주소를 다뤄야 하는 경우를 생각해 봅시다. 현재 IP 주소에는 두 개의 주요한 표준이 있습니다: 버전 4와 버전 6입니다. 프로그램에서 다룰 IP 주소의 경우의 수는 이 두 가지가 전부입니다: 모든 가능한 값을 나열 (*enumerate*) 할 수 있으며, 이 경우를 열거라고 부를 수 있습니다.

IP 주소는 버전 4나 버전 6 중 하나이며, 동시에 두 버전이 될 수는 없습니다. IP 주소의 속성을 보면 열거형 자료 구조가 적절합니다. 왜냐하면, 열거형의 값은 variants 중 하나만 될 수 있기 때문입니다. 버전 4나 버전 6은 근본적으로 IP 주소이기 때문에, 이 둘은 코드에서 모든 종류의 IP 주소에 적용되는 상황을 다룰 때 동일한 타입으로 처리되는 것이 좋습니다.

`IpAddrKind`이라는 열거형을 정의하면서 포함할 수 있는 IP 주소인 `V4` 과 `V6`를 나열함으로써 이 개념을 코드에 표현할 수 있습니다. 이것들은 열거형의 *variants*라고 합니다:

```
enum IpAddrKind {
    V4,
    V6,
}
```

이제 `IpAddrKind`은 우리의 코드 어디에서나 쓸 수 있는 데이터 타입이 되었습니다.

열거형 값

아래처럼 `IpAddrKind`의 두 개의 variants에 대한 인스턴스를 만들 수 있습니다:

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

열거형의 variants는 열거형을 정의한 식별자에 의해 이름 공간이 생기며, 두 개의 콜론을 사용하여 둘을 구분할 수 있습니다. `IpAddrKind::V4`와 `IpAddrKind::V6`의 값은 동일한 타입이기 때문에, 이 방식이 유용합니다: `IpAddrKind`. 이제 `IpAddrKind` 타입을 인자로 받는 함수를 정의할 수 있습니다:

```
fn route(ip_type: IpAddrKind) { }
```

그리고, variant 중 하나를 사용해서 함수를 호출할 수 있습니다:

```
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

열거형을 사용하면 이점이 더 있습니다. IP 주소 타입에 대해 더 생각해 볼 때, 지금으로써는 실제 IP 주소 `0/E`를 저장할 방법이 없습니다. 단지 어떤 종류 인지만 알 뿐입니다. 5장에서 구조체에 대해 방금 공부했다고 한다면, 이 문제를 Listing 6-1에서 보이는 것처럼 풀려고 할 것입니다:

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

Listing 6-1: `struct` 를 사용해서 IP 주소의 데이터와 `IpAddrKind` variant 저장하기

여기서 두 개의 필드를 갖는 `IpAddr` 를 정의했습니다: `IpAddrKind` 타입(이전에 정의한 열거형)인 `kind` 필드와 `String` 타입인 `address` 필드입니다. 구조체에 대한 두 개의 인스턴스가 있습니다. 첫 번째 `home` 은 `kind` 의 값으로 `IpAddrKind::V4` 을 갖고 연관된 주소 데이터로 `127.0.0.1` 를 갖습니다. 두 번째 `loopback` 은 `IpAddrKind` 의 다른 variant 인 `V6` 을 값으로 갖고, 연관된 주소로 `::1` 를 갖습니다. `kind` 와 `address` 의 값을 함께 사용하기 위해 구조체를 사용했습니다. 그렇게 함으로써 variant 가 연관된 값을 갖게 되었습니다.

각 열거형 variant 에 데이터를 직접 넣는 방식을 사용해서 열거형을 구조체의 일부로 사용하는 방식보다 더 간결하게 동일한 개념을 표현할 수 있습니다. `IpAddr` 열거형의 새로운 정의에서는 두 개의 `V4` 와 `V6` variant 는 연관된 `String` 타입의 값을 갖게 됩니다.

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

열거형의 각 variant에 직접 데이터를 붙임으로써, 구조체를 사용할 필요가 없어졌습니다.

구조체 보다 열거형을 사용할 때 다른 장점이 있습니다. 각 variant는 다른 타입과 다른 양의 연관된 데이터를 가질 수 있습니다. 버전 4 타입의 IP 주소는 항상 0 ~ 255 사이의 숫자 4개로 된 구성요소를 갖게 될 것입니다. `V4` 주소에 4개의 `u8` 값을 저장하길 원하지만, `V6` 주소는 하나의 String 값으로 표현되길 원한다면, 구조체로는 이렇게 할 수 없습니다. 열거형은 이런 경우를 쉽게 처리합니다:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

두 가지 다른 종류의 IP 주소를 저장하기 위해 코드상에서 열거형을 정의하는 몇 가지 방법을 살펴봤습니다. 그러나, 누구나 알듯이 IP 주소와 그 종류를 저장하는 것은 흔하기 때문에, 표준 라이브러리에 사용할 수 있는 정의가 있습니다!

표준 라이브러리에서 `IpAddr`를 어떻게 정의하고 있는지 살펴봅시다.

위에서 정의하고 사용했던 것과 동일한 열거형과 variant를 갖고 있지만, variant에 포함된 주소 데이터는 두 가지 다른 구조체로 되어 있으며, 각 variant마다 다르게 정의하고 있습니다:

```
struct Ipv4Addr {
    // details elided
}

struct Ipv6Addr {
    // details elided
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

이 코드에서 보듯이 열거형 variant에 어떤 종류의 데이터라도 넣을 수 있습니다: 예를 들면 문자열, 숫자 타입, 혹은 구조체. 다른 열거형 조차도 포함할 수 있습니다! 또한 표준 라이브러리 타입들은 어떤 경우에는 해결책으로 생각한 것보다 훨씬 더 복잡하지 않습니다.

현재 스코프에 표준 라이브러리를 가져오지 않았기 때문에, 표준 라이브러리에 `IpAddr` 정의가 있더라도, 동일한 이름의 타입을 만들고 사용할 수 있습니다. 타입을 가져오는 것에 대해서는 7장에서 더 살펴볼 것입니다.

다.

Listing 6-2에 있는 열거형의 다른 예제를 살펴봅시다: 이 예제에서는 각 variants에 다양한 유형의 타입들이 포함되어 있습니다:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Listing 6-2: `Message` 열거형은 각 variants가 다른 타입과 다른 양의 값을 저장함.

이 열거형에는 다른 데이터 타입을 갖는 네 개의 variants가 있습니다:

- `Quit`은 연관된 데이터가 전혀 없습니다.
- `Move`은 익명 구조체를 포함합니다.
- `Write`은 하나의 `String`을 포함합니다.
- `ChangeColor`은 세 개의 `i32`을 포함합니다.

Listing 6-2에서처럼 variants로 열거형을 정의하는 것은 다른 종류의 구조체들을 정의하는 것과 비슷합니다. 열거형과 다른 점은 `struct` 키워드를 사용하지 않는다는 것과 모든 variants가 `Message` 타입으로 그룹화된다는 것입니다. 아래 구조체들은 이전 열거형의 variants가 갖는 것과 동일한 데이터를 포함할 수 있습니다:

```
struct QuitMessage; // 유닛 구조체
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // 튜플 구조체
struct ChangeColorMessage(i32, i32, i32); // 튜플 구조체
```

각기 다른 타입을 갖는 여러 개의 구조체를 사용한다면, 이 메시지 중 어떤 한 가지를 인자로 받는 함수를 정의하기 힘들 것입니다. Listing 6-2에 정의한 `Message` 열거형은 하나의 타입으로 이것이 가능합니다.

열거형과 구조체는 한 가지 더 유사한 점이 있습니다: 구조체에 `impl`을 사용해서 메소드를 정의한 것처럼, 열거형에도 정의할 수 있습니다. 여기 `Message` 열거형에 정의한 `call`이라는 메소드가 있습니다:

```
impl Message {
    fn call(&self) {
        // 메소드 내용은 여기 정의할 수 있습니다.
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

열거형의 값을 가져오기 위해 메소드 안에서 `self` 를 사용할 것입니다. 이 예제에서 생성한 변수 `m` 은 `Message::Write(String::from("hello"))` 값을 갖게 되고, 이 값은 `m.call()` 이 실행될 때, `call` 메소드 안에서 `self` 가 될 것입니다.

표준 라이브러리에 있는 매우 흔하게 사용하고 유용한 열거형을 살펴봅시다: `Option`.

Option 열거형과 Null 값 보다 좋은 점들.

이전 절에서, `IpAddr` 열거형을 사용하여 작성한 프로그램에서는 러스트 타입 시스템을 사용하여 데이터뿐만 아니라 더 많은 정보를 담을 수 있는 방법을 살펴보았습니다.

이번 절에서는 표준 라이브러리에서 열거형으로 정의된 또 다른 타입인 `Option` 에 대한 사용 예를 살펴볼 것입니다. `Option` 타입은 많이 사용되는데, 값이 있거나 없을 수도 있는 아주 흔한 상황을 나타내기 때문입니다. 이 개념을 타입 시스템의 관점으로 표현하자면, 컴파일러가 발생할 수 있는 모든 경우를 처리했는지 체크할 수 있습니다. 이렇게 함으로써 버그를 방지할 수 있고, 이것은 다른 프로그래밍 언어에서 매우 흔합니다.

프로그래밍 언어 디자인은 가끔 어떤 특성들이 포함되었는지의 관점에서 생각되기도 하지만, 포함되지 않은 특성들도 역시 중요합니다. 러스트는 다른 언어들에서 흔하게 볼 수 있는 `null` 특성이 없습니다. `Null` 은 값이 없다는 것을 표현하는 하나의 값입니다. `null` 을 허용하는 언어에서는, 변수는 항상 두 상태중 하나가 될 수 있습니다: `null` 혹은 `null` 이 아님.

`null` 을 고안한 Tony Hoare 의 "Null 참조 : 10 억 달러의 실수"에서 다음과 같이 말합니다:

나는 그것을 나의 10억 달러의 실수라고 생각한다. 그 당시 객체지향 언어에서 처음 참조를 위한 포괄적인 타입 시스템을 디자인하고 있었다. 내 목표는 컴파일러에 의해 자동으로 수행되는 체크를 통해 모든 참조의 사용은 절대적으로 안전하다는 것을 확인하는 것이었다. 그러나 `null` 참조를 넣고 싶은 유혹을 참을 수 없었다. 간단한 이유는 구현이 쉽다는 것이었다. 이것은 수없이 많은 오류와 취약점들, 시스템 종료를 유발했고, 지난 40년간 10억 달러의 고통과 손실을 초래했을 수도 있다.

`null` 값으로 발생하는 문제는, `null` 값을 `null` 이 아닌 값처럼 사용하려고 할 때 여러 종류의 오류가 발생할

수 있다는 것입니다. null이나 null이 아닌 속성은 어디에나 있을 수 있고, 너무나도 쉽게 이런 종류의 오류를 만들어냅니다.

그러나, null이 표현하려고 하는 것은 아직까지도 유용합니다: null은 현재 어떤 이유로 유효하지 않고, 존재하지 않는 하나의 값입니다.

문제는 실제 개념에 있기보다, 특정 구현에 있습니다. 이와 같이 러스트에는 null이 없지만, 값의 존재 혹은 부재의 개념을 표현할 수 있는 열거형이 있습니다. 이 열거형은 `Option<T>`이며, 다음과 같이 표준 라이브러리에 정의되어 있습니다:

```
enum Option<T> {
    Some(T),
    None,
}
```

`Option<T>` 열거형은 매우 유용하며 기본적으로 포함되어 있기 때문에, 명시적으로 가져오지 않아도 사용할 수 있습니다. 또한 variants도 마찬가지입니다: `Option::`를 앞에 붙이지 않고, `Some`과 `None`을 바로 사용할 수 있습니다. `Option<T>`는 여전히 일반적인 열거형이고, `Some(T)`과 `None`도 여전히 `Option<T>`의 variants입니다.

`<T>`는 러스트의 문법이며 아직 다루지 않았습니다. 제너릭 타입 파라미터이며, 제너릭에 대해서는 10장에서 더 자세히 다룰 것입니다. 지금은 단지 `<T>`가 `Option` 열거형의 `Some` variant가 어떤 타입의 데이터라도 가질 수 있다는 것을 의미한다는 것을 알고 있으면 됩니다. 여기 숫자 타입과 문자열 타입을 갖는 `Option` 값에 대한 예들이 있습니다:

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

`Some`이 아닌 `None`을 사용한다면, `Option<T>`이 어떤 타입을 가질지 러스트에게 알려줄 필요가 있습니다. 컴파일러는 `None`만 보고는 `Some` variant가 어떤 타입인지 추론할 수 없습니다.

`Some` 값을 얻게 되면, 값이 있다는 것과 `Some`이 갖고 있는 값에 대해 알 수 있습니다. `None` 값을 사용하면, 어떤 면에서는 null과 같은 의미를 갖게 됩니다: 유효한 값을 갖지 않습니다. 그렇다면 왜 `Option<T>`가 null을 갖는 것보다 나을까요?

간단하게 말하면, `Option<T>`와 `T` (`T`는 어떤 타입이던 될 수 있음)는 다른 타입이며, 컴파일러는 `Option<T>`값을 명확하게 유효한 값처럼 사용하지 못하도록 합니다. 예를 들면, 아래 코드는 `Option<i8>`에 `i8`을 더하려고 하기 때문에 컴파일되지 않습니다:

```
let x: i8 = 5;
let y: Optioni8 = Some(5);

let sum = x + y;
```

이 코드를 실행하면, 아래와 같은 에러 메시지가 출력됩니다:

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Optioni8>` is
not satisfied
-->
|
7 | let sum = x + y;
|     ^^^^^^
```

주목하세요! 실제로, 이 에러 메시지는 러스트가 `Option<i8>` 와 `i8` 를 어떻게 더해야 하는지 모른다는 것을 의미하는데, 둘은 다른 타입이기 때문입니다. 러스트에서 `i8` 과 같은 타입의 값을 가질 때, 컴파일러는 항상 유효한 값을 갖고 있다는 것을 보장할 것입니다. 값을 사용하기 전에 null 인지 확인할 필요도 없이 자신 있게 사용할 수 있습니다. 단지 `Option<i8>` 을 사용할 경우엔 (혹은 어떤 타입 이건 간에) 값이 있을지 없을지에 대해 걱정할 필요가 있으며, 컴파일러는 값을 사용하기 전에 이런 케이스가 처리되었는지 확인해 줄 것입니다.

다르게 얘기하자면, `T` 에 대한 연산을 수행하기 전에 `Option<T>` 를 `T` 로 변환해야 합니다. 일반적으로, 이런 방식은 null 과 관련된 가장 흔한 이유 중 하나를 발견하는데 도움을 줍니다: 실제로 null 일 때, null 이 아니라고 가정하는 경우입니다.

null 이 아닌 값을 갖는다는 가정을 놓치는 경우에 대해 걱정할 필요가 없게 되면, 코드에 더 확신을 갖게 됩니다. null 일 수 있는 값을 사용하기 위해서, 명시적으로 값의 타입을 `Option<T>` 로 만들어 줘야 합니다. 그다음엔 값을 사용할 때 명시적으로 null 인 경우를 처리해야 합니다. 값의 타입이 `Option<T>` 가 아닌 모든 곳은 값이 null 아 아니라고 안전하게 가정할 수 있습니다. 이것은 null을 너무 많이 사용하는 문제를 제한하고 러스트 코드의 안정성을 높이기 위한 러스트의 의도된 디자인 결정사항입니다.

그럼 `Option<T>` 타입인 값을 사용할 때, `Some` variant에서 `T` 값을 어떻게 가져와서 사용할 수 있을까요? `Option<T>` 열거형에서 다양한 상황에서 유용하게 사용할 수 있는 많은 메소드들이 있습니다; 문서에서 확인할 수 있습니다. `Option<T>` 의 메소드들에 익숙해지는 것은 러스트를 사용하는데 매우 유용할 것입니다.

일반적으로, `Option<T>` 값을 사용하기 위해서는 각 variant를 처리할 코드가 필요할 것입니다.

`Some(T)` 값일 경우만 실행되는 코드가 필요하고, 이 코드는 안에 있는 `T` 를 사용할 수 있습니다. 다른 코드에서는 `None` 값일 때 실행되는 코드가 필요가 하기도 하며, 이 코드에서는 사용할 수 있는 `T` 값이 없습니다. `match` 표현식은 제어 흐름을 위한 구분으로, 열거형과 함께 사용하면 이런 일들을 할 수 있습니다: 열거형이 갖는 variant에 따라 다른 코드를 실행할 것이고, 그 코드는 매칭 된 값에 있는 데이터를 사용할 수

있습니다.

match 흐름 제어 연산자

러스트는 `match`라고 불리는 극도로 강력한 흐름 제어 연산자를 가지고 있는데 이는 우리에게 일련의 패턴에 대해 어떤 값을 비교한 뒤 어떤 패턴에 매치되었는지를 바탕으로 코드를 수행하도록 해줍니다. 패턴은 리터럴 값, 변수명, 와일드카드, 그리고 많은 다른 것들로 구성될 수 있습니다; 18장에서 다른 모든 종류의 패턴들과 이것들로 할 수 있는 것에 대해 다룰 것입니다. `match`의 힘은 패턴의 표현성으로부터 오며 컴파일러는 모든 가능한 경우가 다루어지는지를 검사합니다.

`match` 표현식을 동전 분류기와 비슷한 종류로 생각해보세요: 동전들은 다양한 크기의 구멍들이 있는 트랙으로 미끄러져 내려가고, 각 동전은 그것에 맞는 첫 번째 구멍을 만났을 때 떨어집니다. 동일한 방식으로, 값들은 `match` 내의 각 패턴을 통과하고, 해당 값에 “맞는” 첫 번째 패턴에서, 그 값은 실행 중에 사용될 연관된 코드 블록 안으로 떨어질 것입니다.

우리가 방금 동전들을 언급했으니, `match`를 이용한 예제로 동전들을 이용해봅시다! Listing 6-3에서 보는 바와 같이, 우리는 익명의 미국 동전을 입력받아서, 동전 계수기와 동일한 방식으로 그 동전이 어떤 것이고 센트로 해당 값을 반환하는 함수를 작성할 수 있습니다.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Listing 6-3: 열거형과 열거형의 variant를 패턴으로서 사용하는 `match` 표현식

`value_in_cents` 함수 내의 `match`를 쪼개 봅시다. 먼저, `match` 키워드 뒤에 표현식을 써줬는데, 위의 경우에는 `coin` 값입니다. 이는 `if`를 사용한 표현식과 매우 유사하지만, 큰 차이점이 있습니다: `if`를 사용하는 경우, 해당 표현식은 부울린 값을 반환할 필요가 있습니다. 여기서는 어떤 타입이든 가능합니다. 위 예제에서 `coin`의 타입은 Listing 6-3에서 정의했던 `Coin` 열거형입니다.

다음은 `match` 갈래(arm)들입니다. 하나의 갈래는 두 부분을 갖고 있습니다: 패턴과 어떤 코드로 되어 있죠. 여기서의 첫 번째 갈래는 값 `Coin::Penny`로 되어있는 패턴을 가지고 있고 그 후에 패턴과 실행되는 코드를 구분해주는 `=>` 연산자가 있습니다. 위의 경우에서 코드는 그냥 값 `1`입니다. 각 갈래는 그다음 갈래와

쉼표로 구분됩니다.

match 표현식이 실행될 때, 결과 값을 각 갈래의 패턴에 대해서 순차적으로 비교합니다. 만일 어떤 패턴이 그 값과 매치되면, 그 패턴과 연관된 코드가 실행됩니다. 만일 그 패턴이 값과 매치되지 않는다면, 동전 분류 기와 비슷하게 다음 갈래로 실행을 계속합니다.

각 갈래와 연관된 코드는 표현식이고, 이 매칭 갈래에서의 표현식의 결과 값은 전체 **match** 표현식에 대해 반환되는 값입니다.

각 갈래가 그냥 값을 리턴하는 Listing 6-3에서처럼 매치 갈래의 코드가 짧다면, 중괄호는 보통 사용하지 않습니다. 만일 매치 갈래 내에서 여러 줄의 코드를 실행시키고 싶다면, 중괄호를 이용할 수 있습니다. 예를 들어, 아래의 코드는 **Coin::Penny** 와 함께 메소드가 호출될 때마다 “Lucky penny!”를 출력하지만 여전히 해당 블록의 마지막 값인 **1**을 반환할 것입니다:

```
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

값들을 바인딩하는 패턴들

매치 갈래의 또 다른 유용한 기능은 패턴과 매치된 값들의 부분을 바인딩할 수 있다는 것입니다. 이것이 열거형 variant로부터 어떤 값들을 추출할 수 있는 방법입니다.

한 가지 예로서, 우리의 열거형 variant 중 하나를 내부에 값을 들고 있도록 바꿔봅시다. 1999년부터 2008년까지, 미국은 각 50개 주마다 한쪽 면의 디자인이 다른 쿼터 동전을 주조했습니다. 다른 동전들은 주의 디자인을 갖지 않고, 따라서 오직 쿼터 동전들만 이 특별 값을 갖습니다. 우리는 이 정보를 **Quarter** variant 내에 **UsState** 값을 포함하도록 우리의 **enum**을 변경함으로써 추가할 수 있는데, 이는 Listing 6-4에서 한 바와 같습니다:

```
#[derive(Debug)] // So we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // ... etc
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Listing 6-4: `Quarter` variant가 `UsState` 값 또한 들고 있는 `Coin` 열거형

우리의 친구가 모든 50개 주 쿼터 동전을 모으기를 시도하는 중이라고 상상해봅시다. 동전의 종류에 따라 동전을 분류하는 동안, 우리는 또한 각 쿼터 동전에 연관된 주의 이름을 외쳐서, 만일 그것이 우리 친구가 가지고 있지 않은 것이라면, 그 친구는 자기 컬렉션에 그 동전을 추가할 수 있겠지요.

이 코드를 위한 매치 표현식 내에서는 variant `Coin::Quarter`의 값과 매치되는 패턴에 `state`라는 이름의 변수를 추가합니다. `Coin::Quarter`이 매치될 때, `state` 변수는 그 쿼터 동전의 주에 대한 값에 바인드 될 것입니다. 그러면 우리는 다음과 같이 해당 갈래에서의 코드 내에서 `state`를 사용할 수 있습니다:

```
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}", state);
            25
        },
    }
}
```

만일 우리가 `value_in_cents(Coin::Quarter(UsState::Alaska))`를 호출했다면, `coin`은 `Coin::Quarter(UsState::Alaska)`가 될 테지요. 각각의 매치 갈래들과 이 값을 비교할 때, `Coin::Quarter(state)`에 도달할 때까지 아무것도 매치되지 않습니다. 이 시점에서, `state`에 대한 바인딩은 값 `UsState::Alaska`가 될 것입니다. 그러면 이 바인딩을 `println!` 표현식 내에서 사용할 수 있고, 따라서 `Quarter`에 대한 `Coin` 열거형 variant로부터 내부의 주에 대한 값을 얻었습니다.

Option<T>를 이용하는 매칭

이전 절에서 `Option<T>`을 사용할 때 `Some` 케이스로부터 내부의 `T` 값을 얻을 필요가 있었습니다; 우리는 `Coin` 열거형을 가지고 했던 것처럼 `match`를 이용하여 `Option<T>`를 다룰 수 있습니다! 동전들을 비교하는 대신, `Option<T>`의 variant를 비교할 것이지만, `match` 표현식이 동작하는 방법은 동일하게 남아있습니다.

`Option<i32>`를 파라미터로 받아서, 내부에 값이 있으면, 그 값에 1을 더하는 함수를 작성하고 싶다고 칡시다. 만일 내부에 값이 없으면, 이 함수는 `None` 값을 반환하고 다른 어떤 연산도 수행하는 시도를 하지 않아야 합니다.

`match`에 감사하게도, 이 함수는 매우 작성하기 쉽고, Listing 6-5와 같이 보일 것입니다:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Listing 6-5: `Option<i32>` 상에서 `match`을 이용하는 함수

Some(T) 매칭 하기

`plus_one`의 첫 번째 실행을 좀 더 자세히 시험해봅시다. `plus_one(five)`가 호출될 때, `plus_one`의 본체 내의 변수 `x`는 값 `Some(5)`를 갖게 될 것입니다. 그런 다음 각각의 매치 갈래에 대하여 이 값을 비교합니다.

`None => None,`

`Some(5)` 값은 패턴 `None`과 매칭 되지 않으므로, 다음 갈래로 계속 갑니다.

`Some(i) => Some(i + 1),`

`Some(5)` 가 `Some(i)` 랙 매칭 되나요? 예, 바로 그렇습니다! 동일한 variant를 갖고 있습니다. `Some` 내부에 담긴 값은 `i`에 바인드 되므로, `i`는 값 `5`를 갖습니다. 그런 다음 매치 갈래 내의 코드가 실행되므로, `i`의 값에 1을 더한 다음 최종적으로 `6`을 담은 새로운 `Some` 값을 생성합니다.

None 매칭 하기

이제 `x`가 `None`인 Listing 6-5에서의 `plus_one`의 두 번째 호출을 살펴봅시다. `match` 안으로 들어와서 첫 번째 갈래와 비교합니다.

```
None => None,
```

매칭 되었군요! 더할 값은 없으므로, 프로그램은 멈추고 `=>`의 우측 편에 있는 `None` 값을 반환합니다. 첫 번째 갈래에 매칭 되었으므로, 다른 갈래와는 비교하지 않습니다.

`match`과 열거형을 조합하는 것은 다양한 경우에 유용합니다. 여러분은 러스트 코드 내에서 이러한 패턴을 많이 보게 될 것입니다: 열거형에 대한 `match`, 내부의 데이터에 변수 바인딩, 그런 다음 그에 대한 수행 코드 말이지요. 처음에는 약간 까다롭지만, 여러분이 일단 익숙해지면, 이를 모든 언어에서 쓸 수 있게 되기를 바랄 것입니다. 이것은 꾸준히 사용자들이 가장 좋아하는 기능입니다.

매치는 하나도 빠뜨리지 않습니다

우리가 논의할 필요가 있는 `match`의 다른 관점이 있습니다. `plus_one` 함수의 아래 버전을 고려해 보세요:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

여기서는 `None` 케이스를 다루지 않았고, 따라서 이 코드는 버그를 일으킬 것입니다. 다행히도, 이는 러스트가 어떻게 잡는지 알고 있는 버그입니다. 이 코드를 컴파일하고자 시도하면, 아래와 같은 에러를 얻게 됩니다:

```
error[E0004]: non-exhaustive patterns: `None` not covered
-->
|
6 |         match x {
|             ^ pattern `None` not covered
```

러스트는 우리가 다루지 않은 모든 가능한 경우를 알고 있고, 심지어 우리가 어떤 패턴을 잊어먹었는지도 알고 있습니다! 러스트에서 매치는 **하나도 빠뜨리지 않습니다(exhaustive)**: 코드가 유효해지기 위해서는 모든 마지막 가능성까지 살살이 다루어야 합니다. 특히 `Option<T>`의 경우, 즉 러스트가 우리로 하여금 `None` 케이스를 명시적으로 다루는 일을 막는 것을 방지하는 경우에는, Null 일지도 모를 값을 가지고 있음을 가정하여, 앞서 논의했던 수십억 달러짜리 실수를 하는 일을 방지해줍니다.

笔记 变경자 placeholder)

러스트는 또한 우리가 모든 가능한 값을 나열하고 싶지 않을 경우에 사용할 수 있는 패턴을 가지고 있습니다. 예를 들어, `u8`은 0에서부터 255까지 유효한 값을 가질 수 있습니다. 만일 우리가 1, 3, 5, 그리고 7 값에 대해서만 신경 쓰고자 한다면, 나머지 0, 2, 4, 6, 8, 그리고 9부터 255까지를 모두 나열하고 싶진 않을 겁니다. 다행히도, 그럴 필요 없습니다: 대신 특별 패턴인 `_`를 이용할 수 있습니다.

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

`_` 패턴은 어떠한 값과도 매칭 될 것입니다. 우리의 다른 갈래 뒤에 이를 집어넣음으로써, `_`는 그전에 명시하지 않은 모든 가능한 경우에 대해 매칭 될 것입니다. `()`는 단지 단위 값이므로, `_` 케이스에서는 어떤 일도 일어나지 않을 것입니다. 결과적으로, 우리가 `_` 변경자 이전에 나열하지 않은 모든 가능한 값들에 대해서는 아무것도 하고 싶지 않다는 것을 말해줄 수 있습니다.

하지만 `match` 표현식은 우리가 단 한 가지 경우에 대해 고려하는 상황에서는 다소 장황할 수 있습니다. 이러한 상황을 위하여, 러스트는 `if let`을 제공합니다.

if let을 사용한 간결한 흐름 제어

`if let` 문법은 `if`와 `let`을 조합하여 하나의 패턴만 매칭 시키고 나머지 경우는 무시하는 값을 다루는 덜 수다스러운 방법을 제공합니다. 어떤 `Option<u8>` 값을 매칭 하지만 그 값이 3일 경우에만 코드를 실행시키고 싶어 하는 Listing 6-6에서의 프로그램을 고려해 보세요:

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

Listing 6-6: 어떤 값이 `Some(3)` 일 때에만 코드를 실행하도록 하는 `match`

우리는 `Some(3)`에 매칭되는 경우에만 원가를 하지만 다른 `Some<u8>` 값 혹은 `None` 값인 경우에는 아무것도 하지 않고 싶습니다. 이러한 `match` 표현식을 만족시키기 위해, `_ => ()`을 단 하나의 variant를 처리한 다음에 추가해야 하는데, 이는 추가하기에 너무 많은 보일러 플레이트 코드입니다.

그 대신, `if let`을 이용하여 이 코드를 더 짧게 쓸 수 있습니다. 아래의 코드는 Listing 6-6에서의 `match`과 동일하게 동작합니다:

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

`if let`은 `=`로 구분된 패턴과 표현식을 입력받습니다. 이는 `match`과 동일한 방식으로 작동하는데, 여기서 표현식은 `match`에 주어지는 것이고 패턴은 이 `match`의 첫 번째 갈래와 같습니다.

`if let`을 이용하는 것은 여러분이 덜 타이핑하고, 덜 들여 쓰기 하고, 보일러 플레이트 코드를 덜 쓰게 된다는 뜻입니다. 하지만, `match` 가 강제했던 하나도 빠짐없는 검사를 잃게 되었습니다. `match`와 `if let` 사이에서 선택하는 것은 여러분의 특정 상황에서 여러분이 하고 있는 것에 따라, 그리고 간결함을 얻는 것이 전수 조사를 잃는 것에 대한 적절한 거래인지에 따라 달린 문제입니다.

바꿔 말하면, 여러분은 `if let`를 어떤 값이 하나 패턴에 매칭되었을 때 코드를 실행하고 다른 값들에 대해서는 무시하는 `match` 문을 위한 문법적 설탕(syntax sugar)으로 생각할 수 있습니다.

`if let`과 함께 `else`를 포함시킬 수 있습니다. `else` 뒤에 나오는 코드 블록은 `match` 표현식에서 `_` 케이스 뒤에 나오는 코드 블록과 동일합니다. Listing 6-4에서 `Quarter` variant가 `UsState` 값도 들고 있었던 `Coin` 열거형 정의부를 상기해 보세요. 만일 우리가 쿼터가 아닌 모든 동전을 세고 싶은 동시에 쿼터 동전일 경우 또한 알려주고 싶었다면, 아래와 같이 `match` 문을 쓸 수 있었을 겁니다:

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

혹은 아래와 같이 `if let`과 `else` 표현식을 이용할 수도 있겠지요:

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

만일 여러분의 프로그램이 `match`로 표현하기에는 너무 수다스러운 로직을 가지고 있는 경우에 놓여 있다면, 여러분의 러스트 도구 상자에는 또한 `if let`이 있음을 기억하세요.

정리

지금까지 우리는 열거한 값들의 집합 중에서 하나가 될 수 있는 커스텀 타입을 만들기 위해서 열거형을 사용하는 방법을 다뤄보았습니다. 우리는 표준 라이브러리의 `Option<T>` 타입이 에러를 방지하기 위해 어떤 식으로 타입 시스템을 이용하도록 도움을 주는지 알아보았습니다. 열거형 값들이 내부에 데이터를 가지고 있을 때는, `match`나 `if let`을 이용하여 그 값을 추출하고 사용할 수 있는데, 둘 중 어느 것을 이용할지는 여러분이 다루고 싶어 하는 경우가 얼마나 많은지에 따라 달라집니다.

여러분의 러스트 프로그램은 이제 구조체와 열거형을 이용해 여러분의 영역 내의 개념을 표현할 수 있습니다. 여러분의 API 내에서 사용할 커스텀 타입을 생성하는 것은 타입 안전성을 보장합니다: 컴파일러는 여러분의 특정 함수들이 예상하는 특정 타입의 값만 갖도록 만들어줄 것입니다.

사용하기 직관적이고 여러분의 사용자가 필요로 할 것만 정확히 노출된 잘 조직화된 API를 여러분의 사용들에게 제공하기 위해서, 이제 러스트의 모듈로 넘어갑시다.

모듈을 사용하여 코드를 재사용하고 조직화하기

여러분이 러스트로 프로그램을 작성하기 시작했을 때, 여러분의 코드는 오로지 `main` 함수 안에만 있을지도 모르겠습니다. 코드가 커짐에 따라서, 여러분은 재사용 및 더 나은 조직화를 위하여 결국 어떤 기능을 다른 함수로 이동시킬 것입니다. 코드를 더 작은 덩어리로 쪼갬으로서, 각각의 덩어리들은 개별적으로 이해하기 더 수월해집니다. 하지만 함수가 너무 많으면 어떤 일이 벌어질까요? 러스트는 조직화된 방식으로 코드의 재사용을 할 수 있게 해주는 모듈(module) 시스템을 갖추고 있습니다.

코드 몇 줄을 함수로 추출하는 것과 같은 방식으로, 여러분은 함수 (혹은 구조체나 열거형 같은 다른 코드들)를 다른 모듈로 뽑아낼 수 있으며, 여러분은 이것들의 정의가 모듈의 바깥쪽에서 볼 수 있도록 하거나(`public`) 혹은 보이지 않게 하도록 (`private`) 선택할 수 있습니다. 모듈이 어떤 식으로 동작하는지에 대한 개요를 봅시다:

- `mod` 키워드는 새로운 모듈을 선언합니다. 모듈 내의 코드는 이 선언 바로 뒤에 중괄호로 묶여서 따라 오거나 다른 파일에 놓일 수 있습니다.
- 기본적으로, 함수, 타입, 상수, 그리고 모듈은 `private`입니다. `pub` 키워드가 어떤 아이템을 `public`하게 만들어줘서 이것의 네임스페이스 바깥쪽에서도 볼 수 있도록 합니다.
- `use` 키워드는 모듈이나 모듈 내의 정의들을 스코프 안으로 가져와서 이들을 더 쉽게 참조할 수 있도록 합니다.

각각의 부분들을 살펴보면서 이것들이 전체적으로 어떻게 맞물리는지 살펴봅시다.

mod와 파일 시스템

먼저 카고를 이용해서 새로운 프로젝트를 만드는 것으로 모듈 예제를 시작하려고 하는데, 바이너리 크레이트(crate)을 만드는 대신에 라이브러리 크레이트를 만들 것입니다. 여기서 라이브러리 크레이트이란 다른 사람들이 자신들의 프로젝트에 디펜던시(dependency)로 추가할 수 있는 프로젝트를 말합니다. 예를 들어, 2장의 `rand` 크레이트은 우리가 추리 게임 프로젝트에서 디펜던시로 사용했던 라이브러리 크레이트입니다.

우리는 몇가지 일반적인 네트워크 기능을 제공하는 라이브러리의 빠대를 만들 것입니다; 여기서는 모듈들과 함수들의 조직화에 집중할 것이고, 함수의 본체에 어떤 코드가 들어가야 하는지는 신경쓰지 않겠습니다. 이 라이브러리를 `communicator`라고 부르겠습니다. 라이브러리를 만들기 위해서는 `--bin` 대신 `--lib` 옵션을 넘기세요:

```
$ cargo new communicator --lib
$ cd communicator
```

카고가 `src/main.rs` 대신 `src/lib.rs`을 생성했음을 주목하세요. `src/lib.rs` 내부를 보면 다음과 같은 코드를 찾을 수 있습니다:

Filename: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

카고는 우리가 만든 라이브러리의 작성 시작을 돋기 위해 빈 테스트를 만드는데, 이는 `--bin` 옵션을 사용했을 때 “Hello, world!” 바이너리를 만들어준 것과 사뭇 다릅니다. `#[]` 와 `mod tests` 문법은 이 장의 `“super”`를 이용하여 부모 모듈에 접근하기” 절에서 더 자세히 다룰 것이지만, 당장은 `src/lib.rs`의 아래쪽에 이 코드를 남겨두겠습니다.

`src/main.rs` 파일이 없기 때문에, `cargo run` 커맨드로 카고가 실행할 것이 없습니다. 따라서, 여기서는 라이브러리 크레이트의 코드를 컴파일하기 위해 `cargo build`를 사용할 것입니다.

이제 여러분이 작성하는 코드의 의도에 따라 만들어지는 다양한 상황에 알맞도록 라이브러리 코드를 조직화하는 다양한 옵션들을 살펴보겠습니다.

모듈 정의

우리의 `communicator` 네트워크 라이브러리를 위해서, 먼저 `connect`라는 이름의 함수가 정의되어 있는 `network`라는 이름의 모듈을 정의하겠습니다. 러스트 내 모듈 정의는 모두 `mod`로 시작됩니다. 이 코드를 `src/lib.rs`의 시작 부분, 즉 테스트 코드의 윗 쪽에 추가해봅시다:

Filename: `src/lib.rs`

```
mod network {
    fn connect() {
    }
}
```

`mod` 키워드 뒤에, 모듈의 이름 `network`가 쓰여지고 중괄호 안에 코드 블록이 옵니다. 이 블록 안의 모든 것은 이름공간 `network` 안에 있습니다. 위의 경우 `connect`라는 이름의 함수 하나가 있습니다. 이 함수를 `network` 모듈 바깥의 스크립트에서 호출하고자 한다면, 우리는 모듈을 특정할 필요가 있으므로 이름공간 문법 `::`를 이용해야 합니다: `connect()` 이렇게만 하지 않고 `network::connect()` 이런 식으로요.

또한 같은 `src/lib.rs` 파일 내에 여러 개의 모듈을 나란히 정의할 수도 있습니다. 예를 들어, `connect`라는 이름의 함수를 갖고 있는 `client` 모듈을 정의하려면, Listing 7-1에 보시는 바와 같이 이를 추가할 수 있습니다:

Filename: `src/lib.rs`

```
mod network {
    fn connect() {
    }
}

mod client {
    fn connect() {
    }
}
```

Listing 7-1: `src/lib.rs` 내에 나란히 정의된 `network` 모듈과 `client` 모듈

이제 우리는 `network::connect` 함수와 `client::connect` 함수를 갖게 되었습니다. 이들은 완전히 다른 기능을 갖고 있을 수 있고, 서로 다른 모듈에 정의되어 있기 때문에 함수 이름이 서로 부딪힐 일은 없습니다.

이 경우, 우리가 라이브러리를 만드는 중이기 때문에, 라이브러리의 시작 지점으로서 제공되는 파일은 `src/lib.rs`입니다. 하지만 모듈을 만드는 것에 관하여 `src/lib.rs`는 특별할 것이 없습니다. 우리는 라이브러리 크레이트의 `src/lib.rs` 내에 모듈을 만드는 것과 똑같은 방식으로 바이너리 크레이트의 `src/main.rs` 내에도 모듈을 만들 수 있습니다. 사실 모듈 안에 다른 모듈을 집어넣는 것도 가능한데, 이는 여러분의 모듈이 커짐에 따라 관련된 기능이 잘 조직화 되도록 하는 한편 각각의 기능을 잘 나누도록 하는데 유용할 수 있습니다. 여러분

분의 코드를 어떻게 조직화 할 것인가에 대한 선택은 여러분이 코드의 각 부분 간의 관계에 대해 어떻게 생각하고 있는지에 따라 달라집니다. 예를 들어, Listing 7-2와 같이 `client` 모듈과 `connect` 함수가 `network` 이름공간 내에 있다면 우리의 라이브러리 사용자가 더 쉽게 이해할지도 모릅니다:

Filename: src/lib.rs

```
mod network {
    fn connect() {
    }

    mod client {
        fn connect() {
        }
    }
}
```

Listing 7-2: `client` 모듈을 `network` 모듈 안으로 이동

`src/lib.rs` 파일에서 Listing 7-2와 같이 `client` 모듈이 `network` 모듈의 내부 모듈이 되도록 `mod network`와 `mod client`의 위치를 바꿔 봅시다. 이제 우리는 `network::connect`와 `network::client::connect` 함수를 갖게 되었습니다: 다시 말하지만, `connect`라는 이름의 두 함수는 서로 다른 이름공간에 있으므로 부딪힐 일이 없습니다.

이런 식으로 모듈들은 계층을 구성하게 됩니다. `src/lib.rs`의 내용은 가장 위의 층을 이루고, 서브 모듈들은 그 보다 낮은 층에 있습니다. Listing 7-1 예제에서의 조직화가 계층 구조를 생각했을 때 어떻게 보일지 살펴봅시다:

```
communicator
└── network
    └── client
```

그리고 Listing 7-2 예제에 대응되는 계층 구조는 이렇습니다:

```
communicator
└── network
    └── client
```

Listing 7-2에서 계층 구조는 `client`가 `network`의 형제이기 보다는 자식임을 보여줍니다. 더 복잡한 프로젝트는 많은 수의 모듈을 갖고 있을 수 있고, 이들은 지속적인 트래킹을 위해 논리적으로 잘 조직화될 필요가 있을 것입니다. 여러분의 프로젝트 내에서 “논리적으로”가 의미하는 것은 여러분에게 달려 있는 것이며, 여러분과 여러분의 라이브러리 사용자들이 프로젝트 도메인에 대해 어떻게 생각하는지에 따라 달라집니다. 여러분이 선호하는 어떤 형태의 구조이건 간에 여기서 보여준 나란한 모듈 및 중첩된(nested) 모듈을 만드는

테크닉을 이용해 보세요.

모듈을 다른 파일로 옮기기

모듈은 계층적인 구조를 형성하는데, 여러분이 익숙하게 사용하고 있는 다른 구조와 매우 닮았습니다: 바로 파일 시스템이죠! 러스트에서는 프로젝트를 잘게 나누기 위해 여러 개의 파일 상에서 모듈 시스템을 사용할 수 있어, 모든 것들이 `src/lib.rs`나 `src/main.rs` 안에 존재하지 않게 할 수 있습니다. 이러한 예를 위해서, Listing 7-3에 있는 코드를 시작해봅시다:

Filename: `src/lib.rs`

```
mod client {
    fn connect() {
    }
}

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

Listing 7-3: 세 개의 모듈 `client`, `network`, `network::server`가 모두 `src/lib.rs`에 정의되어 있음

파일 `src/lib.rs`는 아래와 같은 모듈 계층을 갖고 있습니다:

```
communicator
└── client
    └── network
        └── server
```

만일 이 모듈들이 여러 개의 함수들을 갖고 있고, 이 함수들이 길어지고 있다면, 우리가 작업하고자 하는 코드를 찾으려고 이 파일을 스크롤 하기가 까다로워질 것입니다. 함수들은 하나 혹은 그 이상의 `mod` 블록 안에 포함되어 있기 때문에, 함수 내의 코드 라인들 또한 길어지기 시작할 것입니다. 이는 `client`, `network`, 그리고 `server` 모듈을 `src/lib.rs`로부터 떼어내어 각자를 위한 파일들에 위치시키기 좋은 이유가 되겠습니다.

먼저 `client` 모듈의 코드를 `client` 모듈의 선언 부분만 남겨두는 것으로 바꾸세요. 그러니까 여러분의

`src/lib.rs`는 아래와 같이 될 것입니다:

Filename: `src/lib.rs`

```
mod client;

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

여기서는 여전히 `client` 모듈을 선언하고 있지만, 코드 블록을 세미콜론으로 대체함으로써, 우리는 러스트에게 `client` 모듈의 스코프 내에 정의된 코드를 다른 위치에서 찾으라고 말하는 것입니다. 달리 말하면, `mod client;`라는 라인의 뜻은 이렇습니다:

```
mod client {
    // contents of client.rs
}
```

이제 모듈의 이름과 같은 이름을 가진 외부 파일을 만들 필요가 있습니다. `client.rs` 파일을 여러분의 `src/` 디렉토리에 생성하고 여세요. 그런 뒤 아래와 같이 앞 단계에서 제거했던 `client` 모듈내의 `connect` 함수를 입력해세요:

Filename: `src/client.rs`

```
fn connect() {
```

이미 `src/lib.rs` 안에다 `client` 모듈을 `mod`를 이용하여 선언을 했기 때문에, 이 파일 안에는 `mod` 선언이 필요없다는 점을 기억하세요. 이 파일은 단지 `client` 모듈의 내용물만 제공할 뿐입니다. 만일 `mod client`를 여기에 또 집어넣는다면, 이는 `client` 모듈 내에 서브모듈 `client`를 만들게 됩니다!

러스트는 기본적으로 `src/lib.rs`만 찾아볼줄 압니다. 만약에 더 많은 파일을 프로젝트에 추가하고 싶다면, `src/lib.rs` 내에서 다른 파일을 찾아보라고 러스트에게 말해줄 필요가 있습니다; 이는 `mod client`라는 코드가 왜 `src/lib.rs` 내에 정의될 필요가 있는지, 그리고 `src/client.rs` 내에는 정의될 수 없는지에 대한 이유입니다.

이제 몇 개의 컴파일 경고가 생기지만, 프로젝트는 성공적으로 컴파일 되어야 합니다. 우리가 바이너리 크레

이트 대신 라이브러리 크레이트를 만드는 중이므로 `cargo run` 대신 `cargo build`를 이용해야 한다는 점을 기억해두세요:

```
$ cargo build
Compiling communicator v0.1.0 (file:///projects/communicator)

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/client.rs:1:1
1 | fn connect() {
| ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/lib.rs:4:5
4 |     fn connect() {
|     ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/lib.rs:8:9
8 |         fn connect() {
|         ^
```

이 경고들은 사용된 적이 없는 함수가 있음을 우리에게 알려줍니다. 지금은 이 경고들을 너무 걱정하지 마세요: 이 장의 뒤에 나오는 “`pub`”을 이용하여 가시성 제어하기”절에서 이 문제에 대해 알아볼 것입니다. 좋은 소식은 이들이 그냥 경고일 뿐이란 것입니다; 우리 프로젝트는 성공적으로 빌드됐습니다!

다음으로 같은 방식을 이용하여 `network` 모듈을 개별 파일로 추출해봅시다. `src/lib.rs` 안에서, 아래와 같이 `network` 모듈의 몸체를 지우고 선언부의 끝부분에 세미콜론을 붙이세요:

Filename: `src/lib.rs`

```
mod client;

mod network;
```

그리고나서 새로운 `src/network.rs` 파일을 만들어서 아래를 입력하세요:

Filename: `src/network.rs`

```
fn connect() {  
}  
  
mod server {  
    fn connect() {  
    }  
}
```

이 모듈 파일 내에는 `mod` 선언이 여전히 있음을 주목하세요; 이는 `server` 가 `network` 의 서브모듈로서 여전히 필요하기 때문입니다.

`cargo build` 를 다시 실행시키세요. 성공! 여기 또 추출할만한 모듈이 하나 더 있습니다: `server` 말이죠. 이것이 서브모듈(즉, 모듈 내의 모듈)이기 때문에, 모듈을 파일로 추출해서 파일 이름을 모듈 이름으로 사용하는 전략은 사용하기 힘듭니다. 어쨌든 시도해서 에러를 확인해보겠습니다. 먼저, `src/network.rs` 내에서 `server` 모듈의 내용물 대신에 `mod server` 을 쓰세요:

Filename: `src/network.rs`

```
fn connect() {  
}  
  
mod server;
```

그후 `src/server.rs` 파일을 만들고 추출해둔 `server` 모듈의 내용물을 입력하세요:

Filename: `src/server.rs`

```
fn connect() {  
}
```

`cargo build` 를 실행해보면, Listing 7-4와 같은 에러를 얻게 됩니다:

```
$ cargo build
   Compiling communicator v0.1.0 (file:///projects/communicator)
error: cannot declare a new module at this location
--> src/network.rs:4:5
  |
4 | mod server;
  | ^^^^^^
  |
note: maybe move this module `network` to its own directory via
`network/mod.rs`
--> src/network.rs:4:5
  |
4 | mod server;
  | ^^^^^^
note: ... or maybe `use` the module `server` instead of possibly redeclaring
it
--> src/network.rs:4:5
  |
4 | mod server;
  | ^^^^^^
```

Listing 7-4: `server` 서브모듈을 `src/server.rs`로 추출을 시도했을 때 발생하는 에러

에러는 **이 위치에 새로운 모듈을 선언할 수 없다**고 말해주며 `src/network.rs`의 `mod server;` 라인을 지적하고 있습니다. `src/network.rs`는 `src/lib.rs`와는 다소 다릅니다: 왜 그런지 이해하려면 계속 읽어주세요.

Listing 7-4의 중간의 노트는 실질적으로 매우 도움이 되는데, 그 이유는 우리가 아직 설명하지 않은 무언가를 지적하고 있기 때문입니다:

```
note: maybe move this module `network` to its own directory via
`network/mod.rs`
```

전에 사용했던 똑같은 파일 이름 쓰기 패턴을 계속해서 따르는 대신, 아래 노트에서 제안하는 것을 해볼 수 있습니다:

1. 부모 모듈의 이름에 해당하는, `network`라는 이름의 새로운 디렉토리를 만드세요.
2. `src/network.rs` 파일을 이 새로운 `network` 디렉토리 안으로 옮기고, 파일 이름을 `src/network/mod.rs`로 고치세요.
3. 서브모듈 파일 `src/server.rs`를 `network` 디렉토리 안으로 옮기세요.

위의 단계들을 실행하기 위한 명령들입니다:

```
$ mkdir src/network
$ mv src/network.rs src/network/mod.rs
$ mv src/server.rs src/network
```

이제 `cargo build`를 다시 실행하면, 컴파일은 작동할 것입니다 (여전히 경고는 좀 있지만요). 우리의 모듈 레이아웃은 여전히 아래와 같이 되는데, 이는 Listing 7-3의 `src/lib.rs` 내의 코드에서 만든 것과 정확하게 동일합니다:

```
communicator
└── client
    └── network
        └── server
```

이에 대응하는 파일 레이아웃은 아래와 같이 생겼습니다:

```
src
├── client.rs
├── lib.rs
└── network
    └── mod.rs
        └── server.rs
```

그러니까 우리가 `network::server` 모듈을 추출하고자 할 때, 왜 `network::server` 모듈을 `src/server.rs`로 추출하는 대신, `src/network.rs` 파일을 `src/network/mod.rs`로 바꾸고 `network::server` 코드를 `network` 디렉토리 안에 있는 `src/network/server.rs`에 넣었을까요? 그 이유는 `src` 디렉토리 안에 `server.rs` 파일이 있으면, 러스트는 `server`가 `network`의 서브모듈이라고 인식할 수 없기 때문입니다. 러스트가 동작하는 방식을 명확하게 알기 위해서, 아래와 같은 모듈 계층 구조를 가진, `src/lib.rs` 내에 모든 정의가 다 들어있는 다른 예제를 봅시다:

```
communicator
└── client
    └── network
        └── client
```

이 예제에는 또다시 `client`, `network`, 그리고 `network::client`라는 세 개의 모듈이 있습니다. 모듈을 파일로 추출하기 위해 앞서 했던 단계를 따르면, `client` 모듈을 위한 `src/client.rs`를 만들게 될 것입니다. `network` 모듈을 위해서는 `src/network.rs` 파일을 만들게 될 것입니다. 하지만 `network::client` 모듈을 `src/client.rs`로 추출하는 것은 불가능한데, 그 이유는 최상위 층에 `client` 모듈이 이미 있기 때문이죠! 만일 `client`와 `network::client` 모듈 둘다 `src/client.rs` 파일에 집어넣는다면, 러스트는 이 코드가 `client`를 위한 것인지, 아니면 `network::client`를 위한 것인지 알아낼 방법이 없을 것입니다.

따라서, `network` 모듈의 `network::client` 서브모듈을 위한 파일을 추출하기 위해서는 `src/network.rs` 파일 대신 `network` 모듈을 위한 디렉토리를 만들 필요가 있습니다. `network` 모듈 내의 코드는 그후 `src/network/mod.rs` 파일로 가고, 서브모듈 `network::client`은 `src/network/client.rs` 파일을 갖게할 수 있습니다. 이제 최상위 층의 `src/client.rs`는 모호하지 않게 `client` 모듈이 소유한 코드

가 됩니다.

모듈 파일 시스템의 규칙

파일에 관한 모듈의 규칙을 정리해봅시다:

- 만일 `foo`라는 이름의 모듈이 서브모듈을 가지고 있지 않다면, `foo.rs`라는 이름의 파일 내에 `foo`에 대한 선언을 집어넣어야 합니다.
- 만일 `foo`가 서브모듈을 가지고 있다면, `foo/mod.rs`라는 이름의 파일에 `foo`에 대한 선언을 집어넣어야 합니다.

이 규칙들은 재귀적으로 적용되므로, `foo`라는 이름의 모듈이 `bar`라는 이름의 서브모듈을 갖고 있고 `bar`는 서브모듈이 없다면, 여러분의 `src` 디렉토리 안에는 아래와 같은 파일들이 있어야 합니다:

```
└── foo
    └── bar.rs (contains the declarations in `foo::bar`)
        └── mod.rs (contains the declarations in `foo`, including `mod bar`)
```

이 모듈들은 부모 모듈의 파일에 `mod` 키워드를 사용하여 선언되어 있어야 합니다.

다음으로, `pub` 키워드에 대해 알아보고 앞의 그 경고들을 없애봅시다!

pub으로 가시성(visibility) 제어하기

우리는 `network` 와 `network::server` 코드를 각각 `src/network/mod.rs`와 `src/network/server.rs` 파일 안으로 이동시켜서 Listing 7-4에 나온 에러 메세지를 해결했습니다. 이 지점에서 `cargo build`로 프로젝트를 빌드할 수 있긴 했지만, 사용하지 않고 있는 `client::connect`, `network::connect`, 그리고 `network::server::connect` 함수에 대한 경고 메세지를 보게 됩니다:

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default
src/client.rs:1:1
1 | fn connect() {
  | ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/mod.rs:1:1
1 | fn connect() {
  | ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/server.rs:1:1
1 | fn connect() {
  | ^
```

그럼 이런 경고들은 왜 나오는 걸까요? 결국, 우리는 우리 자신의 프로젝트 내에서 사용할 필요가 있는 것이 아닌, 사용자가 사용할 수 있도록 만들어진 함수들의 라이브러리를 만드는 중이므로, 이런 `connect` 함수 등이 사용되지 않는 것은 큰 문제가 아닙니다. 이 함수들을 만든 의도는 함수들이 우리의 지금 이 프로젝트가 아닌 다른 프로젝트에 사용될 것이란 점입니다.

이 프로그램이 이러한 경고들을 들먹이는 이유를 이해하기 위해, `connect` 라이브러리 를 다른 프로젝트에서 사용하기를 시도해 봅시다. 이를 위해서, 아래의 코드를 담은 `src/main.rs` 파일을 만듦으로서 같은 디렉토리에 라이브러리 크레이트와 마찬가지로 바이너리 크레이트를 만들겠습니다:

Filename: `src/main.rs`

```
extern crate communicator;

fn main() {
    communicator::client::connect();
}
```

`communicator` 라이브러리 크레이트를 가져오기 위해 `extern crate` 명령어를 사용합니다. 우리의 패키지는 이제 두 개의 크레이트를 담고 있습니다. 카고는 `src/main.rs`를 바이너리 크레이트의 루트 파일로 취

급하는데, 이 바이너리 크레이트는 `src/lib.rs`가 루트 파일인 이미 있던 라이브러리 크레이트는 별개입니다. 이러한 패턴은 실행 가능한 프로젝트에서 꽤 흔합니다: 대부분의 기능은 라이브러리 크레이트 안에 있고, 바이너리 크레이트는 이 라이브러리 크레이트를 이용합니다. 결과적으로, 다른 프로그램 또한 그 라이브러리 크레이트를 이용할 수 있고, 이는 멋지게 근심을 덜어줍니다.

`communicator` 라이브러리 밖의 크레이트가 안을 들여다 보는 시점에서, 우리가 만들어왔던 모든 모듈들은 `communicator`라는 이름을 갖는 모듈 내에 있습니다. 크레이트의 최상위 모듈을 **루트 모듈 (root module)** 이라 부릅니다.

또한. 비록 우리의 프로젝트의 서브모듈 내에서 외부 크레이트를 이용하고 있을지라도, `extern crate`이 루트 모듈에 와 있어야 한다는 점(즉 `src/main.rs` 혹은 `src/lib.rs`)을 기억하세요. 그러면 서브모듈 안에서 마치 최상위 모듈의 아이템을 참조하듯 외부 크레이트로부터 아이템들을 참조할 수 있습니다.

현시점에서 우리의 바이너리 크레이트는 고작 라이브러리의 `client` 모듈로부터 `connect` 함수를 호출할 뿐입니다. 하지만 `cargo build`을 실행하면 경고들 이후에 에러를 표시할 것입니다:

```
error: module `client` is private
--> src/main.rs:4:5
   |
4 |     communicator::client::connect();
   | ^^^^^^^^^^
```

아하! 이 에러는 `client` 모듈이 비공개(private)임을 알려주고 있는데, 이는 그 경고들의 요점입니다. 또한 러스트의 내용 중에서 공개(public) 그리고 비공개(private)에 대한 개념에 대해 알아보게 될 첫번째 시간입니다. 러스트의 모든 코드의 기본 상태는 비공개입니다: 즉, 다른 사람은 이 코드를 사용할 수 없습니다. 만일 여러분의 프로그램 내에서 비공개 함수를 이용하지 않는다면, 여러분의 프로그램이 그 함수를 이용할 수 있는 유일한 곳이기 때문에, 러스트는 그 함수가 사용된 적이 없다며 경고해줄 것입니다.

`client::connect`과 같은 함수를 공개로 지정한 뒤에는 우리의 바이너리 크레이트 상에서 이 함수를 호출하는 것이 가능해질 뿐만 아니라, 그 함수가 사용된 적이 없다는 경고 또한 사라질 것입니다. 함수를 공개로 표시하는 것은 러스트로 하여금 그 함수가 우리 프로그램 외부의 코드에 의해 사용될 것이라는 점을 알게끔 해줍니다. 러스트는 이제부터 가능하게 된 이론적인 외부 사용에 대해 이 함수가 “사용되었다”라고 간주합니다. 따라서, 어떤 것이 공개로 표시될 때, 러스트는 그것이 우리 프로그램 내에서 이용되는 것을 요구하지 않으며 해당 아이템이 미사용에 대한 경고를 멈출 것입니다.

함수를 공개로 만들기

러스트에게 어떤 것을 공개하도록 말하기 위해서는, 공개하길 원하는 아이템의 선언 시작 부분에 `pub` 키워드를 추가합니다. 지금은 `client::connect`가 사용된 적 없음을 알리는 경고와 바이너리 크레이트에서 나온 `module `client` is private` 에러를 제거하는데 집중하겠습니다. 아래와 같이 `src/lib.rs`를 수

정하여 `client` 모듈을 공개로 만드세요:

Filename: src/lib.rs

```
pub mod client;
mod network;
```

`pub` 키워드는 `mod` 바로 전에 위치합니다. 다시 빌드를 시도해봅시다:

```
error: function `connect` is private
--> src/main.rs:4:5
   |
4 |     communicator::client::connect();
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

만세! 다른 에러가 나왔습니다! 네, 다른 에러 메세지라는건 축하할만한 이유죠. 새로운 에러는 `function `connect` is private`라고 하고 있으므로, `src/client.rs`를 수정해서 `client::connect`도 공개로 만듭시다:

Filename: src/client.rs

```
pub fn connect() {
```

이제 `cargo build`를 다시 실행하면:

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/mod.rs:1:1
   |
1 | fn connect() {
   | ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/server.rs:1:1
   |
1 | fn connect() {
   | ^
```

코드가 컴파일되었고, `client::connect`가 사용된 적 없다는 것에 대한 경고도 사라집니다!

미사용 코드 경고가 항상 여러분의 코드에 있는 아이템이 공개로 만들 필요가 있음을 나타내는 것은 아닙니다: 이 함수들이 여러분의 공개 API의 일부분으로서 들어가길 원하지 않는다면, 미사용 코드 경고는 여러분에게 해당 코드가 더이상 필요 없고 안전하게 지울 수 있음을 알려줄 수 있습니다. 또한 이 경고는 여러분의 라

이브러리 내에서 해당 함수가 호출된 모든 곳을 실수로 지웠을 경우 발생할 수 있는 버그를 알려줄 수도 있습니다.

하지만 지금의 경우, 우리는 다른 두 함수들이 우리 크레이트의 공개 API의 일부분이 되길 원하고 있으므로, 이들에게 **pub**를 표시해줘서 남은 경고들을 제거합시다. *src/network/mod.rs*를 아래와 같이 수정하세요:

Filename: *src/network/mod.rs*

```
pub fn connect() {
}

mod server;
```

그리고 컴파일하면:

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/mod.rs:1:1
1 | pub fn connect() {
  | ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/server.rs:1:1
1 | fn connect() {
  | ^
```

흠, `network::connect` 가 **pub**으로 설정되어 있음에도, 여전히 미사용 함수 경고가 나옵니다. 그 이유는 함수가 모듈 내에서 공개지만, 함수가 상주해 있는 `network` 모듈은 공개가 아니기 때문입니다. 이번에는 모듈의 안쪽에서 작업하고 있지만, `client::connect` 에서는 바깥쪽에서 작업을 했었죠. *src/lib.rs*를 수정하여 `network` 가 공개가 되도록 할 필요가 있습니다. 이렇게요:

Filename: *src/lib.rs*

```
pub mod client;

pub mod network;
```

이제 컴파일하면, 그 경고는 사라집니다:

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/server.rs:1:1
1 | fn connect() {
| ^
```

경고 딱 하나 남았네요! 여러분이 직접 고쳐보세요!

비공개 규칙(Privacy Rules)

종합해보면, 아이템 가시성에 관한 규칙은 다음과 같습니다:

1. 만일 어떤 아이템이 공개라면, 이는 부모 모듈의 어디에서건 접근 가능합니다.
2. 만일 어떤 아이템이 비공개라면, 같은 파일 내에 있는 부모 모듈 및 이 부모의 자식 모듈에서만 접근 가능합니다.

비공개 예제(Privacy Examples)

연습을 위해 몇 가지 비공개에 관한 예제를 봅시다. 새로운 라이브러리 프로젝트를 만들고 이 새로운 프로젝트의 `src/lib.rs`에 Listing 7-5와 같이 코드를 넣으세요:

Filename: `src/lib.rs`

```
mod outermost {
    pub fn middle_function() {}

    fn middle_secret_function() {}

    mod inside {
        pub fn inner_function() {}

        fn secret_function() {}
    }
}

fn try_me() {
    outermost::middle_function();
    outermost::middle_secret_function();
    outermost::inside::inner_function();
    outermost::inside::secret_function();
}
```

Listing 7-5: 비공개 및 공개 함수 예제. 몇 가지는 잘못되었음.

이 코드를 컴파일하기 전에, `try_me` 함수의 어떤 라인이 에러를 발생시킬지 추측해보세요. 그리고나서 컴파일을 하여 여러분이 맞았는지 확인하고, 에러에 대한 논의를 위해 계속 읽어주세요!

에러 보기

`try_me` 함수는 우리 프로젝트의 루트 모듈 내에 있습니다. `outermost` 모듈은 비공개지만, 두 번째 비공개 규칙은 `try_me` 함수가 `outermost` 모듈에 접근하는 것이 허용됨을 알려주는데, 이는 `outermost` 가 `try_me` 함수와 마찬가지로 현재의 (루트) 모듈 내에 있기 때문입니다.

`middle_function`이 공개이므로 `outermost::middle_function` 호출은 작동할 것이며, `try_me` 는 `middle_function`의 부모 모듈인 `outermost`를 통해 `middle_function`에 접근하고 있습니다. 이 모듈에 접근 가능하하는 것은 이전 문단에서 알아냈죠.

`outermost::middle_secret_function` 호출은 컴파일 에러를 일으킬 것입니다.

`middle_secret_function`는 비공개이므로, 두번째 규칙이 적용됩니다. 루트 모듈은 `middle_secret_function`의 현재 모듈도 아니고 (`outermost` 가 현재 모듈입니다), `middle_secret_function`의 현재 모듈의 자식 모듈도 아닙니다.

`inside` 모듈은 비공개고 자식 모듈이 없으므로, 이것의 현재 모듈인 `outermost`에 의해서만 접근될 수 있습니다. 이는 즉 `try_me` 함수는 `outermost::inside::inner_function`나 `outermost::inside::secret_function`를 호출할 수 없음을 의미합니다.

에러 고치기

여기 이 에러들을 고치기 위해 코드를 수정하는것에 관한 몇 가지 제안이 있습니다. 각각을 시도해보기 전에, 이 시도가 에러를 고칠지 그렇지 않을지 추측해 보고, 컴파일을 해서 여러분이 맞췄는지 그렇지 않은지 확인하고, 왜 그랬는지 이해하기 위해 비공개 규칙을 이용해보세요.

- `inside` 모듈이 공개라면 어떨까요?
- `outermost` 가 공개고 `inside` 가 비공개면 어떨까요?
- `inner_function`의 내부에서 `::outermost::middle_secret_function()` 을 호출한다면 어떨까요? (시작 부분의 콜론 두개는 루트 모듈로부터 시작하여 모듈을 참조하고 싶음을 나타냅니다)

자유롭게 더 많은 실험을 설계하고 시도해 보세요!

다음으로, `use` 키워드를 사용하여 아이템을 스코프 내로 가져오는 것에 대해 이야기해 복시다.

이름 가져오기 (Importing Names)

우리는 Listing 7-6에서 보시는 것과 같이 `nested_modules` 함수를 호출하는 것처럼, 모듈 이름을 호출 구문의 일부분으로 사용하여 해당 모듈 내에 정의된 함수를 호출하는 방법을 다룬바 있습니다:

Filename: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

fn main() {
    a::series::of::nested_modules();
}
```

Listing 7-6: 함수에 인접한 모듈 경로를 완전히 특정한 함수 호출하기

보시다시피 완전하게 경로를 지정한 이름을 참조하는 것은 너무 길어질 수 있습니다. 다행히도 러스트는 이러한 호출을 더 간결하게 만들어주는 키워드를 가지고 있습니다.

use를 이용한 간결한 가져오기

러스트의 `use` 키워드는 여러분이 스코프 내에서 호출하고 싶어하는 함수의 모듈을 가져옴으로써 긴 함수 호출을 줄여줍니다. `a::series::of` 모듈을 바이너리 크레이트의 루트 스코프로 가져온 예제입니다:

Filename: src/main.rs

```

pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of;

fn main() {
    of::nested_modules();
}

```

`use a::series::of;` 줄은 `of` 모듈을 참조하고 싶은 곳마다 `a::series::of` 전부를 사용하기 보다는 `of`를 사용할 수 있다는 뜻입니다.

`use` 키워드는 우리가 명시한 것만 스코프 내로 가져옵니다: 즉 모듈의 자식들을 스코프 내로 가져오지는 않습니다. 이는 `nested_modules` 함수를 호출하고자 할 때 여전히 `of::nested_modules`를 사용해야 하는 이유입니다.

다음과 같이 `use` 구문 안에서 모듈 대신 함수를 명시하여 스코프 내에서 함수를 가져올 수도 있습니다:

```

pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of::nested_modules;

fn main() {
    nested_modules();
}

```

이렇게 하면 모든 모듈을 안 써주고 함수를 직접 참조하도록 해줍니다.

열거형 또한 모듈과 비슷한 일종의 이름공간을 형성하고 있기 때문에, 열거형의 variant 또한 `use`를 이용하여 가져올 수 있습니다. 어떠한 `use` 구문이건 하나의 이름공간으로부터 여러 개의 아이템을 가져오려 한다면, 여러분은 아래와 같이 중괄호와 쉼표를 구문의 마지막 위치에 사용하여 이 아이템들을 나열할 수 있습니다:

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::{Red, Yellow};

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = TrafficLight::Green;
}
```

`Green` variant에 대해서는 여전히 `TrafficLight` 이름공간을 명시하고 있는데, 이는 `use` 구문 내에 `Green`을 포함하지 않았기 때문입니다.

*를 이용한 모두(glob) 가져오기

이름공간 내의 모든 아이템을 가져오기 위해서는 `*` 문법을 이용할 수 있습니다. 예를 들면:

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::*;

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = Green;
}
```

`*`는 글롭(glob)이라고 부르며, 이는 이름공간 내에 공개된 모든 아이템을 가져올 것입니다. 여러분은 글롭을 아껴가며 써야 합니다: 글롭은 편리하지만, 여러분이 예상한 것보다 더 많은 아이템을 끌어와서 이름 간의 충돌(naming conflict)의 원인이 될 수도 있습니다.

super를 사용하여 부모 모듈에 접근하기

이 장의 시작 부분에서 보셨듯이, 여러분이 라이브러리 크레이트를 만들 때, 카고는 여러분들을 위해 `tests`

모듈을 만들어줍니다. 지금부터 이에 대한 구체적인 부분들을 봅시다. 여러분의 **communicator** 프로젝트 내에 있는 `src/lib.rs`를 여세요:

Filename: `src/lib.rs`

```
pub mod client;

pub mod network;

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

11장에서 테스트에 관한 더 많은걸 설명하고 있습니다만, 이 예제는 지금도 이해가 되시리라 봅니다: `tests`라는 이름의 모듈이 우리의 다른 모듈들 옆에 있고, `it_works`라는 이름의 함수 하나를 담고 있지요. 좀 특별한 주석(annotation)이 있지만, `tests` 모듈은 그냥 또다른 모듈일 뿐입니다! 따라서 우리의 모듈 계층 구조는 아래와 같이 생겼습니다:

```
communicator
├── client
└── network
    └── client
        └── tests
```

테스트는 우리 라이브러리 내에 있는 코드를 연습하기 위한 것이므로, 현재로서는 어떠한 기능도 확인할 게 없긴 하지만, `it_works` 함수 안에서 우리의 `client::connect` 함수를 호출해 봅시다:

Filename: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        client::connect();
    }
}
```

`cargo test` 명령을 써서 테스트를 실행하면:

```
$ cargo test
Compiling communicator v0.1.0 (file:///projects/communicator)
error[E0433]: failed to resolve. Use of undeclared type or module `client`
--> src/lib.rs:9:9
   |
9 |     client::connect();
   |     ^^^^^^^^^^^^^^ Use of undeclared type or module `client`
```

컴파일이 실패했습니다, 하지만 대체 왜일까요? 우리는 `src/main.rs`에서 했었던 것과 마찬가지로 함수 앞에 `communicator::`를 붙일 필요가 없는데, 왜냐하면 이 코드가 분명히 `communicator` 라이브러리 크레이트 안에 있기 때문입니다. 원인은 경로가 항상 현재 모듈을 기준으로 상대적인데, 여기는 `test`이기 때문입니다. 딱 하나의 예외는 `use` 구문인데, 이는 기본적으로 크레이트 루트에 대한 상대적인 경로로 인식됩니다. 우리의 `tests` 모듈은 이 스코프 내에서 `client` 모듈이 필요합니다!

그러면 어떻게 모듈 계층 구조 내에서 한 모듈 위로 거슬러 올라가 `tests` 모듈 안에서 `client::connect` 함수를 호출할 수 있을까요? 아래와 같이 앞에 콜론 두개를 사용하여 러스트에게 우리가 루트부터 시작하여 전체 경로를 나열하겠다고 알려주는 방법이 있습니다:

```
::client::connect();
```

혹은, 아래와 같이 `super`를 사용하여 계층 구조 상에서 현재 모듈로부터 한 모듈 거슬러 올라갈 수도 있습니다:

```
super::client::connect();
```

이 두 가지 옵션은 이번 예제에서는 차이가 없는 것처럼 보이지만, 여러분의 모듈 계층 구조가 깊어진다면, 매번 루트에서부터 경로를 시작하는 것은 여러분의 코드를 길게 만들 것입니다. 그런 경우에는 `super`를 이용하여 현재 모듈에서 형제 모듈을 가져오는 것이 좋은 지름길이 됩니다. 여기에 더해서, 만약 여러분이 여러 군데에 루트로부터 시작되는 경로를 명시한 뒤에 서브트리를 다른 곳으로 옮기는 식으로 여러분의 모듈을 재정리한다면, 여러분은 여러 군데의 경로를 간단하도록 요구받는 처지가 될 것이고, 이는 지루한 작업이 될 것입니다.

각각의 테스트에 `super::`를 타이핑해야 하는 것이 짜증날수 있겠지만, 여러분은 이미 여기에 대한 해답이 될 도구를 보셨습니다: `use` 말이죠! `super::`의 기능은 `use`에 제공한 경로를 변경시켜서 이제 루트 모듈 대신 부모 모듈에 상대적인 경로가 되게 해줍니다.

이러한 이유로, 특히 `tests` 모듈 내에서는 보통 `use super::something`이 가장 좋은 해결책이 됩니다. 따라서 이제 우리의 테스트는 이렇게 됩니다:

Filename: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    use super::client;

    #[test]
    fn it_works() {
        client::connect();
    }
}
```

`cargo test`를 다시 실행시키면, 테스트가 통과되고 테스트 결과 출력의 첫번째 부분이 아래와 같이 나타날 것입니다:

```
$ cargo test
Compiling communicator v0.1.0 (file:///projects/communicator)
Running target/debug/communicator-92007ddb5330fa5a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

정리

이제 여러분은 코드를 조직화하기 위한 몇가지 새로운 기술을 알게 되었습니다! 관련된 기능들을 함께 묶여주는 이 기술들을 사용하고, 파일들이 너무 길어지지 않게 하고, 여러분의 라이브러리 사용자들에게 깔끔한 공개 API를 제공해 보세요.

다음으로 여러분의 멋지고 깔끔한 코드에 사용할 수 있는 표준 라이브러리 내의 몇가지 컬렉션 데이터 구조를 보겠습니다!

일반적인 컬렉션

러스트의 표준 라이브러리에는 컬렉션이라 불리는 여러 개의 매우 유용한 데이터 구조들이 포함되어 있습니다. 대부분의 다른 데이터 타입들은 하나의 특정한 값을 나타내지만, 컬렉션은 다수의 값을 담을 수 있습니다. 내장된 배열(build-in array)와 튜플 타입과는 달리, 이 컬렉션들이 가리키고 있는 데이터들은 힙에 저장되는데, 이는 즉 데이터량이 컴파일 타임에 결정되지 않아도 되며 프로그램이 실행될 때 늘어나거나 줄어들 수 있다는 의미입니다. 각각의 컬렉션 종류는 서로 다른 용량과 비용을 가지고 있으며, 여러분의 현재 상황에 따라 적절한 컬렉션을 선택하는 것은 시간이 지남에 따라 발전시켜야 할 기술입니다. 이번 장에서는 러스트 프로그램에서 굉장히 자주 사용되는 세 가지 컬렉션에 대해 논의해 보겠습니다:

- **벡터(vector)** 는 여러 개의 값을 서로 붙여 있게 저장할 수 있도록 해줍니다.
- **스트링(string)** 은 문자(character)의 모음입니다. **String** 타입은 이전에 다루었지만, 이번 장에서는 더 깊이 있게 이야기해 보겠습니다.
- **해쉬맵(hash map)** 은 어떤 값을 특정한 키와 연관지어 주도록 해줍니다. 이는 **맵(map)** 이라 일컫는 좀더 일반적인 데이터 구조의 특정한 구현 형태입니다.

표준 라이브러리가 제공해주는 다른 종류의 컬렉션에 대해 알고 싶으시면, [the documentation](#)를 봐 주세요.

이제부터 어떻게 벡터, 스트링, 해쉬맵을 만들고 업데이트하는지 뿐만 아니라 어떤 것이 각각의 컬렉션을 특별하게 해주는지에 대해 논의해 보겠습니다.

벡터

우리가 보게될 첫번째 콜렉션은 벡터라고도 알려진 `Vec<T>`입니다. 벡터는 메모리 상에 서로 이웃하도록 모든 값을 집어넣는 단일 데이터 구조 안에 하나 이상의 값을 저장하도록 해줍니다. 벡터는 같은 타입의 값만을 저장할 수 있습니다. 이는 여러분이 파일 내의 텍스트의 라인들이라던가 장바구니의 아이템 가격들 같은 아이템 리스트를 저장하는 상황일 경우 유용합니다.

새 벡터 만들기

비어있는 새 벡터를 만들기 위해서는, 아래의 Listing 8-1과 같이 `Vec::new` 함수를 호출해 줍니다:

```
let v: Vec<i32> = Vec::new();
```

Listing 8-1: `i32` 타입의 값을 가질 수 있는 비어있는 새 벡터 생성

여기에 타입 명시(type annotation)를 추가한 것을 주목하세요. 이 벡터에 어떠한 값도 집어넣지 않았기 때문에, 러스트는 우리가 저장하고자 하는 요소의 종류가 어떤 것인지 알지 못합니다. 이는 중요한 지점입니다. 벡터는 제네릭(generic)을 이용하여 구현되었습니다; 제네릭을 이용하여 여러분만의 타입을 만드는 방법은 10장에서 다룰 것입니다. 지금 당장은, 표준 라이브러리가 제공하는 `Vec` 타입은 어떠한 종류의 값이라도 저장할 수 있다는 것, 그리고 특정한 벡터는 특정한 타입의 값을 저장할 때, 이 타입은 꺼쇠 괄호(<>) 안에 적는다는 것만 알아두세요. Listing 8-1에서는 러스트에게 `v` 안의 `Vec`가 `i32` 타입의 요소를 가질 것이고 알려주었습니다.

일단 우리가 값을 집어넣으면 러스트는 우리가 저장하고자 하는 값의 타입을 대부분 유추할 수 있으므로, 좀 더 현실적인 코드에서는 이러한 타입 명시를 할 필요가 거의 없습니다. 초기값들을 갖고 있는 `Vec<T>`을 생성하는 것이 더 일반적이며, 러스트는 편의를 위해 `vec!` 매크로를 제공합니다. 이 매크로는 우리가 준 값들을 저장하고 있는 새로운 `Vec`를 생성합니다. Listing 8-2는 `1, 2, 3`을 저장하고 있는 새로운 `Vec<i32>`을 생성할 것입니다:

```
let v = vec![1, 2, 3];
```

Listing 8-2: 값을 저장하고 있는 새로운 벡터 생성하기

초기 `i32` 값을 제공했기 때문에, 러스트는 `v` 가 `Vec 타입이라는 것을 유추할 수 있으며, 그래서 타입 명시는 필요치 않습니다. 다음은, 벡터를 어떻게 수정하는지를 살펴보겠습니다.

벡터 갱신하기

벡터를 만들고 여기에 요소들을 추가하기 위해서는 아래 Listing 8-3과 같이 `push` 메소드를 사용할 수 있습니다:

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

Listing 8-3: `push` 메소드를 사용하여 벡터에 값을 추가하기

3장에서 설명한 바와 같이, 어떤 변수에 대해 그 변수가 담고 있는 값이 변경될 수 있도록 하려면, `mut` 키워드를 사용하여 해당 변수를 가변으로 만들어 줄 필요가 있습니다. 우리가 집어넣는 숫자는 모두 `i32` 타입이며, 러스트는 데이터로부터 이 타입을 추론하므로, 우리는 `Vec<i32>` 명시를 붙일 필요가 없습니다.

벡터를 드롭하는 것은 벡터의 요소들을 드롭시킵니다

`struct` 와 마찬가지로, Listing 8-4에 달려있는 주석처럼 벡터도 스코프 밖으로 벗어났을 때 해제됩니다:

```
{
    let v = vec![1, 2, 3, 4];
    // v를 가지고 뭔가 합니다
} // <- v가 스코프 밖으로 벗어났고, 여기서 해제됩니다
```

Listing 8-4: 벡터와 벡터의 요소들이 드롭되는 곳을 보여주기

벡터가 드롭될 때 벡터의 내용물 또한 전부 드롭되는데, 이는 벡터가 가지고 있는 정수들이 모두 제거된다는 의미입니다. 이는 직관적인 것처럼 보일 수도 있겠지만 벡터의 요소들에 대한 참조자를 만들때는 좀 더 복잡해 질 수 있습니다. 다음으로 이런 상황을 파헤쳐 봅시다!

벡터의 요소들 읽기

지금까지 벡터를 만들고, 간신히, 없애는 방법에 대해 알아보았으니, 벡터의 내용물을 읽어들이는 방법을 알아보는 것이 다음 단계로 좋아보입니다. 벡터 내에 저장된 값을 참조하는 두 가지 방법이 있습니다. 예제에서는 특별히 더 명료하게 하기 위해 함수들이 반환하는 값의 타입을 명시했습니다.

Listing 8-5는 인덱스 문법이나 `get` 메소드를 가지고 벡터의 값에 접근하는 두 방법 모두를 보여주고 있습니다.

니다:

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
let third: Option<&i32> = v.get(2);
```

Listing 8-5: 인덱스 문법 혹은 `get` 메소드를 사용하여 벡터 내의 아이템에 접근하기

두가지 세부사항을 주목하세요. 첫번째로, 인덱스값 `2`를 사용하면 세번째 값이 얻어집니다: 벡터는 0부터 시작하는 숫자로 인덱스됩니다. 두번째로, 세번째 요소를 얻기 위해 두 가지 다른 방법이 사용되었습니다: `&` 와 `[]`를 이용하여 참조자를 얻은 것과, `get` 함수에 인덱스를 파라미터로 넘겨서 `Option<&T>`를 얻은 것입니다.

러스트가 벡터 요소를 참조하는 두가지 방법을 제공하는 이유는 여러분이 벡터가 가지고 있지 않은 인덱스값을 사용하고자 했을 때 프로그램이 어떻게 동작할 것인지 여러분이 선택할 수 있도록 하기 위해서입니다. 예를 들어, 아래의 Listing 8-6과 같이 5개의 요소를 가지고 있는 벡터가 있고 100 인덱스에 있는 요소에 접근하려고 시도한 경우 프로그램은 어떻게 동작해야 할까요:

```
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

Listing 8-6: 5개의 요소를 가진 벡터에 100 인덱스에 있는 요소에 접근하기

이 프로그램을 실행하면, 첫번째의 `[]` 메소드는 `panic!`을 일으키는데, 이는 존재하지 않는 요소를 참조하기 때문입니다. 이 방법은 여러분의 프로그램이 벡터의 끝을 넘어서는 요소에 접근하는 시도를 하면 프로그램이 죽게끔 하는 치명적 에러를 발생하도록 하기를 고려하는 경우 가장 좋습니다.

`get` 함수에 벡터 범위를 벗어난 인덱스가 주어졌을 때는 패닉 없이 `None`이 반환됩니다. 보통의 환경에서 벡터의 범위 밖에 있는 요소에 접근하는 것이 종종 발생한다면 이 방법을 사용할만 합니다. 여러분의 코드는 우리가 6장에서 본 것과 같이 `Some(&element)` 혹은 `None`에 대해 다루는 로직을 갖추어야 합니다. 예를 들어 인덱스는 사람이 직접 번호를 입력하는 것으로 들어올 수도 있습니다. 만일 사용자가 잘못하여 너무 큰 숫자를 입력하여 프로그램이 `None` 값을 받았을 경우, 여러분은 사용자에게 현재 `Vec`에 몇개의 아이템이 있으며 유효한 값을 입력할 또한번의 기회를 줄 수도 있습니다. 이런 편이 오타 때문에 프로그램이 죽는 것 보다는 더 사용자 친화적이겠죠!

유효하지 않은 참조자

프로그램이 유효한 참조자를 얻을 때, 빌림 검사기(borrow checker)가 (4장에서 다루었던) 소유권 및 빌림

규칙을 집행하여 이 참조자와 벡터의 내용물로부터 얻은 다른 참조자들이 계속 유효하게 남아있도록 확실히 해줍니다. 같은 스코프 내에서 가변 참조자와 불변 참조자를 가질 수 없다는 규칙을 상기하세요. 이 규칙은 아래 예제에서도 적용되는데, Listing 8-7에서는 벡터의 첫번째 요소에 대한 불변 참조자를 얻은 뒤 벡터의 끝에 요소를 추가하고자 했습니다:

```
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];
v.push(6);
```

Listing 8-7: 아이템에 대한 참조자를 가지는 동안 벡터에 요소 추가 시도하기

이 예제를 컴파일하면 아래와 같은 에러가 발생합니다:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
4 | let first = &v[0];
   |         - immutable borrow occurs here
5 |
6 | v.push(6);
   | ^ mutable borrow occurs here
7 | }
   | - immutable borrow ends here
```

Listing 8-7의 코드는 동작을 해야만 할것처럼 보일 수도 있습니다: 왜 첫번째 요소에 대한 참조자가 벡터 끝에 대한 변경을 걱정해야 하죠? 이 에러에 대한 내막은 벡터가 동작하는 방법 때문입니다: 새로운 요소를 벡터의 끝에 추가하는 것은 새로 메모리를 할당하여 예전 요소를 새 공간에 복사하는 일을 필요로 할 수 있는데, 이는 벡터가 모든 요소들을 붙여서 저장할 공간이 충분치 않는 환경에서 일어날 수 있습니다. 이러한 경우, 첫 번째 요소에 대한 참조자는 할당이 해제된 메모리를 가리키게 될 것입니다. 빌림 규칙은 프로그램이 이러한 상황에 빠지지 않도록 해줍니다.

노트: `Vec<T>` 타입의 구현 세부사항에 대한 그밖의 것에 대해서는 <https://doc.rust-lang.org/stable/nomicon/vec.html>에 있는 노미콘(The Nomicon)을 보세요:

벡터 내의 값들에 대한 반복처리

만일 벡터 내의 각 요소들을 차례대로 접근하고 싶다면, 하나의 값에 접근하기 위해 인덱스를 사용하는것 보다는, 모든 요소들에 대해 반복처리를 할 수 있습니다. Listing 8-8은 `for` 루프를 사용하여 `i32`의 벡터 내

에 있는 각 요소들에 대한 불변 참조자를 얻어서 이를 출력하는 방법을 보여줍니다:

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

Listing 8-8: `for` 루프를 이용한 요소들에 대한 반복작업을 통해 각 요소들을 출력하기

만일 모든 요소들을 변형시키길 원한다면 가변 벡터 내의 각 요소에 대한 가변 참조자로 반복작업을 할 수도 있습니다. Listing 8-9의 `for` 루프는 각 요소에 `50`을 더할 것입니다:

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Listing 8-9: 벡터 내의 요소에 대한 가변 참조자로 반복하기

가변 참조자가 참고하고 있는 값을 바꾸기 위해서, `i`에 `+=` 연산자를 이용하기 전에 역참조 연산자 (`*`)를 사용하여 값을 얻어야 합니다.

열거형을 사용하여 여러 타입을 저장하기

이 장의 시작 부분에서, 벡터는 같은 타입을 가진 값들만 저장할 수 있다고 이야기했습니다. 이는 불편할 수 있습니다; 다른 타입의 값들에 대한 리스트를 저장할 필요가 있는 상황이 분명히 있지요. 다행히도, 열거형의 variant는 같은 열거형 타입 내에 정의가 되므로, 벡터 내에 다른 타입의 값들을 저장할 필요가 있다면 열거형을 정의하여 사용할 수 있습니다!

예를 들어, 스프레드시트의 행으로부터 값들을 가져오고 싶은데, 여기서 어떤 열은 정수를, 어떤 열은 실수를, 어떤 열은 스트링을 갖고 있다고 해봅시다. 우리는 다른 타입의 값을 가지는 variant가 포함된 열거형을 정의할 수 있고, 모든 열거형 variant들은 해당 열거형 타입, 즉 같은 타입으로 취급될 것입니다. 따라서 우리는 궁극적으로 다른 타입을 담은 열거형 값에 대한 벡터를 생성할 수 있습니다. Listing 8-10에서 이를 보여주고 있습니다:

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

Listing 8-10: 열거형을 정의하여 벡터 내에 다른 타입의 데이터를 담을 수 있도록 하기

러스트가 컴파일 타임에 벡터 내에 저장될 타입이 어떤 것인지 알아야 할 필요가 있는 이유는 각 요소를 저장하기 위해 얼만큼의 힙 메모리가 필요한지 알기 위함입니다. 부차적인 이점은 이 벡터에 허용되는 타입에 대해 명시적일 수 있다는 점입니다. 만일 러스트가 어떠한 타입이든 담을 수 있는 벡터를 허용한다면, 벡터 내의 각 요소마다 수행되는 연산에 대해 하나 혹은 그 이상의 타입이 에러를 야기할 수도 있습니다. 열거형과 **match** 표현식을 사용한다는 것은 6장에서 설명한 바와 같이 러스트가 컴파일 타임에 모든 가능한 경우에 대해 처리한다는 것을 보장해준다는 의미입니다.

만약 프로그램을 작성할 때 여러분의 프로그램이 런타임에 벡터에 저장하게 될 타입의 모든 경우를 알지 못한다면, 열거형을 이용한 방식은 사용할 수 없을 것입니다. 대신 트레이트 객체(trait object)를 이용할 수 있는데, 이건 17장에서 다루게 될 것입니다.

지금까지 벡터를 이용하는 가장 일반적인 방식 중 몇 가지에 대해 논의했는데, 표준 라이브러리의 **Vec**에 정의된 수많은 유용한 메소드들이 있으니 API 문서를 꼭 살펴봐 주시기 바랍니다. 예를 들면, **push**에 더해서, **pop** 메소드는 제일 마지막 요소를 반환하고 지워줍니다. 다음 콜렉션 타입인 **String**으로 넘어갑시다!

스트링

4장에서 스트링에 관한 이야기를 했습니다만, 지금은 좀 더 깊이 살펴보겠습니다. 새로운 러스트인들은 흔히들 스트링 부분에서 막히는데 이는 세 가지 개념의 조합으로 인한 것입니다: 가능한 에러를 꼭 노출하도록 하는 러스트의 성향, 많은 프로그래머의 예상보다 더 복잡한 데이터 구조인 스트링, 그리고 UTF-8입니다. 다른 언어들을 사용하다 왔을 때 이 개념들의 조합이 러스트의 스트링을 어려운 것처럼 보이게 합니다.

스트링이 컬렉션 장에 있는 이유는 스트링이 바이트의 컬렉션 및 이 바이트들을 텍스트로 통역할 때 유용한 기능을 제공하는 몇몇 메소드로 구현되어 있기 때문입니다. 이번 절에서는 생성, 갱신, 값 읽기와 같은 모든 컬렉션 타입이 가지고 있는, `String`에서의 연산에 대해 이야기 해보겠습니다. 또한 `String`을 다른 컬렉션들과 다르게 만드는 부분, 즉 사람과 컴퓨터가 `String` 데이터를 통역하는 방식 간의 차이로 인해 생기는 `String` 인덱싱의 복잡함을 논의해보겠습니다.

스트링이 뭔가요?

먼저 스트링이라는 용어가 정확히 무엇을 뜻하는 것인지 정의해보겠습니다. 러스트는 핵심 언어 기능 내에서 딱 한 가지 스트링 타입만 제공하는데, 이는 바로 스트링 슬라이스인 `str`이고, 이것의 참조자 형태인 `&str`을 많이 봤죠. 4장에서는 스트링 슬라이스에 대해 얘기했고, 이는 다른 어딘가에 저장된 UTF-8로 인코딩된 스트링 데이터의 참조자입니다. 예를 들어, 스트링 리터럴은 프로그램의 바이너리 출력물 내에 저장되어 있으며, 그러므로 스트링 슬라이스입니다.

`String` 타입은 핵심 언어 기능 내에 구현된 것이 아니고 러스트의 표준 라이브러리를 통해 제공되며, 커질 수 있고, 가변적이며, 소유권을 갖고 있고, UTF-8로 인코딩된 스트링 타입입니다. 러스트인들이 “스트링”에 대해 이야기할 때, 그들은 보통 `String`과 스트링 슬라이스 `&str` 타입 둘 모두를 이야기한 것이지, 이들 중 하나를 뜻한 것은 아닙니다. 이번 절은 대부분 `String`에 관한 것이지만, 두 타입 모두 러스트 표준 라이브러리에서 매우 많이 사용되며 `String`과 스트링 슬라이스 모두 UTF-8로 인코딩되어 있습니다.

또한 러스트 표준 라이브러리는 `OsString`, `OsStr`, `CString`, 그리고 `cstr`과 같은 몇 가지 다른 스트링 타입도 제공합니다. 심지어 어떤 라이브러리 크레이트들은 스트링 데이터를 저장하기 위해 더 많은 옵션을 제공할 수 있습니다. `*String`/`*Str`이라는 작명과 유사하게, 이들은 종종 소유권이 있는 타입과 이를 빌린 변형 타입을 제공하는데, 이는 `String`/`&str`과 비슷합니다. 이러한 스트링 타입들은, 예를 들면 다른 종류의 인코딩이 된 텍스트를 저장하거나 다른 방식으로 메모리에 저장될 수 있습니다. 여기서는 이러한 다른 스트링 타입은 다루지 않겠습니다; 이것들을 어떻게 쓰고 어떤 경우에 적합한지에 대해 알고 싶다면 각각의 API 문서를 확인하시기 바랍니다.

새로운 스트링 생성하기

`Vec`에서 쓸 수 있는 많은 연산들이 `String`에서도 마찬가지로 똑같이 쓰일 수 있는데, `new` 함수를 이용

하여 스트링을 생성하는 것으로 아래의 Listing 8-11과 같이 시작해봅시다:

```
let mut s = String::new();
```

Listing 8-11: 비어있는 새로운 `String` 생성하기

이 라인은 우리가 어떤 데이터를 담아둘 수 있는 `s`라는 빈 스트링을 만들어 줍니다. 종종 우리는 스트링에 담아두고 시작할 초기값을 가지고 있을 것입니다. 그런 경우, `to_string` 메소드를 이용하는데, 이는 `Display` 트레이트가 구현된 어떤 타입이든 사용 가능하며, 스트링 리터럴도 이 트레이트를 구현하고 있습니다. Listing 8-12에서 두 가지 예제를 보여주고 있습니다:

```
let data = "initial contents";  
  
let s = data.to_string();  
  
// the method also works on a literal directly:  
let s = "initial contents".to_string();
```

Listing 8-12: `to_string` 메소드를 사용하여 스트링 리터럴로부터 `String` 생성하기

이 코드는 `initial contents`를 담고 있는 스트링을 생성합니다.

또한 스트링 리터럴로부터 `String`을 생성하기 위해서 `String::from` 함수를 이용할 수도 있습니다. Listing 8-13의 코드는 `to_string`을 사용하는 Listing 8-12의 코드와 동일합니다:

```
let s = String::from("initial contents");
```

Listing 8-13: `String::from` 함수를 사용하여 스트링 리터럴로부터 `String` 생성하기

스트링이 너무나 많은 것들에 사용되기 때문에, 스트링을 위해 다양한 제네릭 API들을 사용할 수 있으며, 다양한 옵션들을 제공합니다. 몇몇은 쓸모없는 것처럼 느껴질 수도 있지만, 다 사용할 곳이 있습니다! 지금의 경우, `String::from`과 `.to_string`은 정확히 똑같은 일을 하며, 따라서 어떤 것을 사용하는가는 여러분의 스타일에 따라 달린 문제입니다.

스트링이 UTF-8로 인코딩되었음을 기억하세요. 즉, 아래의 Listing 8-14에서 보는 것처럼 우리는 인코딩된 어떤 데이터라도 포함시킬 수 있습니다:

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("ହୋଲ୍ଟୋ");
let hello = String::from("নমস্তে");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

Listing 8-14: 스트링에 다양한 언어로 인삿말 저장하기

위의 모두가 유효한 `String` 값입니다.

스트링 갱신하기

`String`은 크기가 커질 수 있으며 이것이 담고 있는 내용물은 `Vec`의 내용물과 마찬가지로 더 많은 데이터를 집어넣음으로써 변경될 수 있습니다. 추가적으로, `+` 연산자나 `format!` 매크로를 사용하여 편리하게 `String` 값들을 서로 접합(concatenation)할 수 있습니다.

`push_str`과 `push`를 이용하여 스트링 추가하기

Listing 8-15와 같이 스트링 슬라이스를 추가하기 위해 `push_str` 메소드를 이용하여 `String`을 키울 수 있습니다:

```
let mut s = String::from("foo");
s.push_str("bar");
```

Listing 8-15: `push_str` 메소드를 사용하여 `String`에 스트링 슬라이스 추가하기

`s`는 위의 두 라인 뒤에 “foobar”를 담게 될 것입니다. `push_str` 메소드는 스트링 슬라이스를 파라미터로 갖는데 이는 파라미터의 소유권을 가져올 필요가 없기 때문입니다. 예를 들어, Listing 8-16의 코드는 `s1`에 `s2`의 내용물을 추가한 뒤 `s2`를 더 이상 쓸 수 없게 되었다면 불행했을 경우를 보여주고 있습니다:

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(&s2);
println!("s2 is {}", s2);
```

Listing 8-16: 스트링 슬라이스를 `String`에 붙인 이후에 스트링 슬라이스를 사용하기

만일 `push_str` 함수가 `s2`의 소유권을 가져갔다면, 마지막 줄에서 그 값을 출력할 수 없었을 것입니다. 하지만, 이 코드는 우리가 기대했던 대로 작동합니다!

`push` 메소드는 한 개의 글자를 파라미터로 받아서 `String`에 추가합니다. Listing 8-17은 `push` 메소드를 사용하여 `String`에 를 추가하는 코드를 보여주고 있습니다:

```
let mut s = String::from("lo");
s.push('l');
```

Listing 8-17: `push`를 사용하여 `String` 값에 한 글자 추가하기

위의 코드를 실행한 결과로 `s`는 `lol`을 담고 있을 것입니다.

+ 연산자나 `format!` 매크로를 이용한 접합

종종 우리는 가지고 있는 두 개의 스트링을 조합하고 싶어합니다. 한 가지 방법은 아래 Listing 8-18와 같이

+ 연산자를 사용하는 것입니다:

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // s1은 여기서 이동되어 더이상 쓸 수 없음을 유의하세요
```

Listing 8-18: + 연산자를 사용하여 두 `String` 값을 하나의 새로운 `String` 값으로 조합하기

위의 코드 실행 결과로서, 스트링 `s3`는 `Hello, world!`를 담게 될 것입니다. `s1`이 더하기 연산 이후에 더이상 유효하지 않은 이유와 `s2`의 참조자가 사용되는 이유는 + 연산자를 사용했을 때 호출되는 함수의 시그니처와 맞춰야 하기 때문입니다. + 연산자는 `add` 메소드를 사용하는데, 이 메소드의 시그니처는 아래처럼 생겼습니다:

```
fn add(self, s: &str) -> String {
```

이는 표준 라이브러리에 있는 정확한 시그니처는 아닙니다: 표준 라이브러리 내에서 `add`는 제네릭을 이용하여 정의되어 있습니다. 여기서는 제네릭에 구체 타입(concrete type)을 대입한 `add`의 시그니처를 보는 중인데, 이는 우리가 `String` 값으로 이 메소드를 호출했을 때 생깁니다. 제네릭에 대한 내용은 10장에서 다룰 것입니다. 이 시그니처는 교묘한 + 연산자를 이해하는데 필요한 단서를 줍니다.

첫번째로, `s2`는 &를 가지고 있는데, 이는 `add` 함수의 `s` 파라미터 때문에 첫번째 스트링에 두번째 스트링의 참조자를 더하고 있음을 뜻합니다: 우리는 `String`에 `&str`만 더할 수 있고, 두 `String`을 더하지는

못합니다. 하지만, 잠깐만요 - `&s2`의 타입은 `&String`이지, `add`의 두번째 파라미터에 명시한것처럼 `&str`은 아니죠. 왜 Listing 8-18의 예제가 컴파일될까요? `&s2`를 `add` 호출에 사용할 수 있는 이유는 `&String` 인자가 `&str`로 강제될 수 있기 때문입니다 - `add` 함수가 호출되면, 러스트는 역참조 강제 (*deref coercion*) 라 불리는 무언가를 사용하는데, 이는 `add` 함수내에서 사용되는 `&s2`가 `&s2[...]`로 바뀌는 것으로 생각할 수 있도록 해줍니다. 역참조 강제에 대한 것은 15장에서 다룰 것입니다. `add`가 파라미터의 소유권을 가져가지는 않으므로, `s2`는 이 연산 이후에도 여전히 유효한 `String`일 것입니다.

두번째로, 시그니처에서 `add`가 `self`의 소유권을 가져가는 것을 볼 수 있는데, 이는 `self`가 `&`를 안 가지고 있기 때문입니다. 즉 Listing 8-18의 예제에서 `s1`이 `add` 호출로 이동되어 이후에는 더 이상 유효하지 않을 것이라는 의미입니다. 따라서 `let s3 = s1 + &s2;` 가 마치 두 스트링을 복사하여 새로운 스트링을 만들 것처럼 보일지라도, 실제로 이 구문은 `s1`의 소유권을 가져다가 `s2`의 내용물의 복사본을 추가한 다음, 결과물의 소유권을 반환합니다. 달리 말하면, 이 구문은 여러 복사본을 만드는 것처럼 보여도 그렇지 않습니다: 이러한 구현은 복사보다 더 효율적입니다.

만일 여러 스트링을 접하고자 한다면, `+`의 동작은 다루기 불편해 집니다.:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

이 지점에서 `s`는 `tic-tac-toe`가 될 것입니다. 모든 `+`와 `"` 문자들과 함께 보면 어떤 결과가 나올지 알기 어렵습니다. 더 복잡한 스트링 조합을 위해서는 `format!` 매크로를 사용할 수 있습니다:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}-{}-{}", s1, s2, s3);
```

이 코드 또한 `s`에 `tic-tac-toe`을 설정합니다. `format!` 매크로는 `println!`과 똑같은 방식으로 작동하지만, 스크린에 결과를 출력하는 대신 결과를 담은 `String`을 반환해줍니다. `format!`을 이용한 버전이 훨씬 읽기 쉽고, 또한 어떠한 파라미터들의 소유권도 가져가지 않습니다.

스트링 내부의 인덱싱

다른 많은 프로그래밍 언어들에서, 인덱스를 이용한 참조를 통해 스트링 내부의 개별 문자들에 접근하는 것은 유효하고 범용적인 연산에 속합니다. 그러나 러스트에서 인덱싱 문법을 이용하여 `String`의 부분에 접근하고자 하면 에러를 얻게 됩니다. 아래 Listing 8-19와 같은 코드를 생각해봅시다:

```
let s1 = String::from("hello");
let h = s1[0];
```

Listing 8-19: 스트링에 인덱싱 문법을 사용하는 시도

이 코드는 아래와 같은 에러를 출력합니다:

```
error: the trait bound `std::string::String: std::ops::Index<_>` is not
satisfied [--explain E0277]
|>
|>     let h = s1[0];
|>         ^^^^^^
note: the type `std::string::String` cannot be indexed by `_`
```

에러와 노트 부분이 이야기해 줍니다: 러스트 스트링은 인덱싱을 지원하지 않는다고. 그렇지만 왜 안되는 걸까요? 이 질문에 답하기 위해서는 러스트가 어떻게 스트링을 메모리에 저장하는지에 관하여 살짝 이야기해야 합니다.

내부적 표현

`String`은 `Vec<u8>`을 감싼 것입니다(wrapper). Listing 8-14에서 보았던 몇가지 적절히 인코딩된 UTF-8 예제 스트링을 살펴봅시다. 첫번째로, 이것입니다:

```
let len = String::from("Hola").len();
```

이 경우, `len`은 4가 되는데, 이는 스트링 “Hola”를 저장하고 있는 `Vec`이 4바이트 길이라는 뜻입니다. UTF-8로 인코딩되면 각각의 글자들이 1바이트씩 차지한다는 것이죠. 그런데 아래 예제는 어떨까요?

```
let len = String::from("Здравствуйте").len();
```

이 스트링의 길이가 얼마인지 묻는다면, 여러분은 12라고 답할런지도 모릅니다. 그러나 러스트의 대답은 24입니다. 이는 “Здравствуйте”를 UTF-8로 인코딩된 바이트들의 크기인데, 각각의 유니코드 스칼라 값이 저장소의 2바이트를 차지하기 때문입니다. 따라서, 스트링의 바이트들 안의 인덱스는 유효한 유니코드 스칼라 값과 항상 대응되지는 않을 것입니다.

이를 보여주기 위해, 다음과 같은 유효하지 않은 러스트 코드를 고려해 보세요:

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

`answer`의 값은 무엇이 되어야 할까요? 첫번째 글자인 3이 되어야 할까요? UTF-8로 인코딩될 때, 3의 첫

번짜 바이트는 208이고, 두번째는 151이므로, `answer`는 사실 208이 되어야 하지만, 208은 그 자체로는 유효한 문자가 아닙니다. 208을 반환하는 것은 사람들이 이 스트링의 첫번째 글자를 요청했을 경우 사람들이 기대하는 것이 아닙니다; 하지만 그게 러스트가 인덱스 0에 가지고 있는 유일한 데이터죠. 바이트 값을 반환하는 것은 아마도 유저들이 원하는 것이 아닐 것입니다. 심지어는 라틴 글자들만 있을 때도요: `&"hello"[0]`는 h가 아니라 104를 반환합니다. 기대치 않은 값을 반환하고 즉시 발견하기 힘들지도 모를 버그를 야기하는 것을 방지하기 위해, 러스트는 이러한 코드를 전혀 컴파일하지 않고 이러한 오해들을 개발 과정 내에서 일찌감치 방지합니다.

바이트와 스칼라 값과 문자소 클러스터(Grapheme cluster)! 이런!

UTF-8에 대한 또 다른 지점은, 실제로는 러스트의 관점에서 문자열을 보는 세 가지의 의미있는 방식이 있다는 것입니다: 바이트, 스칼라 값, 그리고 문자소 클러스터(우리가 글자라고 부르는 것과 가장 근접한 것)입니다.

데바가나리 글자로 쓰여진 힌디어 “नमस्ते”를 보면, 이것은 궁극적으로 아래와 같이 `u8` 값들의 `Vec`으로서 저장됩니다:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]
```

이건 18바이트이고 컴퓨터가 이 데이터를 궁극적으로 저장하는 방법입니다. 만일 우리가 이를 유니코드 스칼라 값, 즉 러스트의 `char` 타입인 형태로 본다면, 아래와 같이 보이게 됩니다:

```
['न', 'म', 'स', '्', 'त', 'े']
```

여섯개의 `char` 값이 있지만, 네번째와 여섯번째는 글자가 아닙니다: 그 자체로는 이해할 수 없는 발음 구별 부호입니다. 마지막으로, 만일 이를 문자소 클러스터로서 본다면, 사람들이 발음할 이 힌디 단어를 구성하는 네 글자를 얻습니다:

```
["न", "म", "स", "्ते"]
```

러스트는 컴퓨터가 저장하는 가공되지 않은(raw) 스트링을 번역하는 다른 방법을 제공하여, 데이터가 담고 있는 것이 어떤 인간의 언어든 상관없이 각각의 프로그램이 필요로 하는 통역방식을 선택할 수 있도록 합니다.

러스트가 `String`을 인덱스로 접근하여 문자를 얻지 못하도록 하는 마지막 이유는 인덱스 연산이 언제나 상수 시간($O(1)$)에 실행될 것으로 기대받기 때문입니다. 그러나 `String`을 가지고 그러한 성능을 보장하는 것은 불가능한데, 그 이유는 러스트가 스트링 내에 얼마나 많은 유효 문자가 있는지 알아내기 위해 내용물의 시작 지점부터 인덱스로 지정된 곳까지 훑어야 하기 때문입니다.

스트링 슬라이싱하기

스트링 인덱싱의 리턴 타입이 어떤 것이 (바이트 값인지, 캐릭터인지, 문자소 클러스터인지, 혹은 스트링 슬라이스인지) 되어야 하는지 명확하지 않기 때문에 스트링의 인덱싱은 종종 나쁜 아이디어가 됩니다. 따라서, 여러분이 스트링 슬라이스를 만들기 위해 정말로 인덱스를 사용하고자 한다면 러스트는 좀 더 구체적으로 지정하도록 요청합니다. 여러분의 인덱싱을 더 구체적으로 하고 스트링 슬라이스를 원한다는 것을 가리키기 위해서, `[]`에 숫자 하나를 사용하는 인덱싱보다, `[]`와 범위를 사용하여 특정 바이트들이 담고 있는 스트링 슬라이스를 만들 수 있습니다:

```
let hello = "Здравствуйте";
let s = &hello[0..4];
```

여기서 `s`는 스트링의 첫 4바이트를 담고 있는 `&str`가 될 것입니다. 앞서 우리는 이 글자들이 각각 2바이트를 차지한다고 언급했으므로, 이는 `s`가 “Зд”이 될 것이라 뜻입니다.

만약에 `&hello[0..1]`라고 했다면 어떻게 될까요? 답은 다음과 같습니다: 러스트는 벡터 내에 유효하지 않은 인덱스에 접근했을 때와 동일한 방식으로 런타임에 패닉을 발생시킬 것입니다.

```
thread 'main' panicked at 'index 0 and/or 1 in `Здравствуйте` do not lie on character boundary', ../../src/libcore/str/mod.rs:1694
```

여러분은 스트링 슬라이스를 만들기 위하여 범위를 이용하는 방법을 조심스럽게 사용해야 하는데, 이는 여러분의 프로그램을 죽게 만들 수도 있기 때문입니다.

스트링 내에서 반복적으로 실행되는 메소드

다행히도, 스트링의 요소에 접근하는 다른 방법이 있습니다.

만일 개별적인 유니코드 스칼라 값에 대한 연산을 수행하길 원한다면, 가장 좋은 방법은 `chars` 메소드를 이용하는 것입니다. `chars`를 “나누기”에 대해 호출하면 `char` 타입의 6개의 값으로 나누어 반환하며, 여러분은 각각의 요소에 접근하기 위해 이 결과값에 대해 반복(iterate)할 수 있습니다:

```
for c in "नमस्ते".chars() {
    println!("{}", c);
}
```

이 코드는 다음을 출력할 것입니다:

나
마
사
다
라
트
트
트

bytes 메소드는 가공되지 않은 각각의 바이트를 반환하는데, 여러분의 문제 범위에 따라 적절할 수도 있습니다:

```
for b in "नमस्ते".bytes() {  
    println!("{}", b);  
}
```

이 코드는 이 **String**을 구성하는 아래처럼 시작되는 18 바이트를 출력합니다:

```
224  
164  
168  
224  
// ... etc
```

하지만 유효한 유니코드 스칼라 값이 하나 이상의 바이트로 구성될지도 모른다는 것을 확실히 기억해주세요.

스트링으로부터 문자소 클러스터를 얻는 방법은 복잡해서, 이 기능은 표준 라이브러리를 통해 제공되지 않습니다. 여러분이 원하는 기능이 이것이라면 [crates.io](#)에서 사용 가능한 크레이트가 있습니다.

스트링은 그렇게 단순하지 않습니다

종합하면, 스트링은 복잡합니다. 다른 프로그래밍 언어들은 이러한 복잡성을 프로그래머에게 어떻게 보여줄지에 대해 각기 다른 선택을 합니다. 러스트는 **String** 데이터의 올바른 처리가 모든 러스트 프로그램에 대한 기본적인 동작이 되도록 선택했는데, 이는 솔직히 프로그래머들이 UTF-8 데이터를 처리하는데 있어 더 많은 생각을 해야한다는 의미입니다. 이러한 거래는 다른 프로그래밍 언어들에 비해 더 복잡한 스트링을 노출시키지만, 한편으로는 여러분의 개발 생활 주기 후반에 비 ASCII 캐릭터를 포함하는 에러를 처리해야 하는 것을 막아줄 것입니다.

이것보다 살짝 덜 복잡한 것으로 옮겨 갑시다: 해쉬맵이요!

해쉬맵(hash map)

마지막으로 볼 일반적인 컬렉션은 해쉬맵입니다. `HashMap<K, V>` 타입은 `K` 타입의 키에 `V` 타입의 값을 매핑한 것을 저장합니다. 이 매핑은 해쉬 함수(*hashing function*)을 통해 동작하는데, 해쉬 함수는 이 키와 값을 메모리 어디에 저장할지 결정합니다. 많은 다른 프로그래밍 언어들도 이러한 종류의 데이터 구조를 지원 하지만, 종종 해쉬, 맵, 오브젝트, 해쉬 테이블, 혹은 연관 배열 (associative) 등과 같은 그저 몇몇 다른 이름으로 이용됩니다.

해쉬맵은 여러분이 벡터를 이용하듯 인덱스를 이용하는 것이 아니라 임의의 타입으로 된 키를 이용하여 데이터를 찾기를 원할때 유용합니다. 예를 들면, 게임 상에서는 각 팀의 점수를 해쉬맵에 유지할 수 있는데, 여기서 키는 팀의 이름이고 값은 팀의 점수가 될 수 있습니다. 팀의 이름을 주면, 여러분은 그 팀의 점수를 찾을 수 있습니다.

이 장에서는 해쉬맵의 기본 API를 다룰 것이지만, 표준 라이브러리의 `HashMap`에 정의되어 있는 함수 중에는 더 좋은 것들이 숨어있습니다. 항상 말했듯이, 더 많은 정보를 원하신다면 표준 라이브러리 문서를 확인하세요.

새로운 해쉬맵 생성하기

우리는 빈 해쉬맵을 `new`로 생성할 수 있고, `insert`를 이용하여 요소를 추가할 수 있습니다. Listing 8-20에서, 우리는 팀 이름이 각각 블루(Blue)와 옐로우(Yellow)인 두 팀의 점수를 유지하고 있습니다. 블루 팀은 10점, 옐로우 팀은 50점으로 시작할 것입니다:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

Listing 8-20: 새로운 해쉬맵을 생성하여 몇 개의 키와 값을 집어넣기

먼저 표준 라이브러리의 컬렉션 부분으로부터 `HashMap`을 `use`로 가져와야 할 필요가 있음을 주목하세요. 우리가 보고 있는 세 가지 일반적인 컬렉션 중에서 이 해쉬맵이 제일 덜 자주 사용되는 것이기 때문에, 프렐루드(prelude) 내에 자동으로 가져와지는 기능에 포함되어 있지 않습니다. 또한 해쉬맵은 표준 라이브러리로부터 덜 지원을 받습니다; 예를 들면 해쉬맵을 생성하는 빌트인 매크로가 없습니다.

벡터와 마찬가지로, 해쉬맵도 데이터를 힙에 저장합니다. 이 `HashMap`은 `String` 타입의 키와 `i32` 타입의 값을 갖습니다. 벡터와 비슷하게 해쉬맵도 동질적입니다: 모든 키는 같은 타입이어야 하고, 모든 값도 같은 타입이어야 합니다.

해쉬맵을 생성하는 또 다른 방법은 튜플의 벡터에 대해 `collect` 메소드를 사용하는 것인데, 이 벡터의 각 튜플은 키와 값으로 구성되어 있습니다. `collect` 메소드는 데이터를 모아서 `HashMap`을 포함한 여러 컬렉션 타입으로 만들어줍니다. 예를 들면, 만약 두 개의 분리된 벡터에 각각 팀 이름과 초기 점수를 갖고 있다면, 우리는 `zip` 메소드를 이용하여 “Blue”와 10이 한 쌍이 되는 식으로 튜플의 벡터를 생성할 수 있습니다. 그 다음 Listing 8-21과 같이 `collect` 메소드를 사용하여 튜플의 벡터를 `HashMap`으로 바꿀 수 있습니다:

```
use std::collections::HashMap;

let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> =
    teams.iter().zip(initial_scores.iter()).collect();
```

Listing 8-21: 팀의 리스트와 점수의 리스트로부터 해쉬맵 생성하기

타입 명시 `HashMap<_, _>`이 필요한데 이는 `collect` 가 다른 많은 데이터 구조로 바뀔 수 있고, 러스트는 여러분이 특정하지 않으면 어떤 것을 원하는지 모르기 때문입니다. 그러나 키와 값의 타입에 대한 타입 파라미터에 대해서는 밑줄을 쓸 수 있으며 러스트는 벡터에 담긴 데이터의 타입에 기초하여 해쉬에 담길 타입을 추론할 수 있습니다.

해쉬맵과 소유권

`i32` 와 같이 `Copy` 트레이트를 구현한 타입에 대하여, 그 값들은 해쉬맵 안으로 복사됩니다. `String`과 같이 소유된 값들에 대해서는, 아래의 Listing 8-22와 같이 값들이 이동되어 해쉬맵이 그 값들에 대한 소유자가 될 것입니다:

```
use std::collections::HashMap;

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name과 field_value은 이 지점부터 유효하지 않습니다.
// 이들을 이용하는 시도를 해보고 어떤 컴파일러 에러가 나오는지 보세요!
```

Listing 8-22: 키와 값이 삽입되는 순간 이들이 해쉬맵의 소유가 되는 것을 보여주는 예

`insert` 를 호출하여 `field_name` 과 `field_value` 를 해쉬맵으로 이동시킨 후에는 더 이상 이 둘을 사용할 수 없습니다.

만일 우리가 해쉬맵에 값들의 참조자들을 삽입한다면, 이 값들은 해쉬맵으로 이동되지 않을 것입니다. 하지만 참조자가 가리키고 있는 값은 해쉬맵이 유효할 때까지 계속 유효해야합니다. 이것과 관련하여 10장의 “라이프타임을 이용한 참조자 유효화”절에서 더 자세히 이야기할 것입니다.

해쉬맵 내의 값 접근하기

Listing 8-23과 같이 해쉬맵의 `get` 메소드에 키를 제공하여 해쉬맵으로부터 값을 얻어올 수 있습니다:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

Listing 8-23: 해쉬맵 내에 저장된 블루 팀의 점수 접근하기

여기서 `score`는 블루 팀과 연관된 값을 가지고 있을 것이고, 결과값은 `Some(&10)`일 것입니다. 결과값은 `Some`으로 감싸져 있는데 왜냐하면 `get`이 `Option<&V>`를 반환하기 때문입니다; 만일 해쉬맵 내에 해당 키에 대한 값이 없다면 `get`은 `None`을 반환합니다. 프로그램은 우리가 6장에서 다루었던 방법 중 하나로 `Option`을 처리해야 할 것입니다.

우리는 벡터에서 했던 방법과 유사한 식으로 `for` 루프를 이용하여 해쉬맵에서도 각각의 키/값 쌍에 대한 반복작업을 할 수 있습니다:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

이 코드는 각각의 쌍을 임의의 순서로 출력할 것입니다:

```
Yellow: 50
Blue: 10
```

해쉬맵 갱신하기

키와 값의 개수가 증가할 수 있을지라도, 각각의 개별적인 키는 한번에 연관된 값 하나만을 가질 수 있습니다. 해쉬맵 내의 데이터를 변경하길 원한다면, 키에 이미 값이 할당되어 있을 경우에 대한 처리를 어떻게 할지 결정해야 합니다. 예전 값을 완전히 무시하면서 예전 값을 새 값으로 대신할 수도 있습니다. 혹은 예전 값을 계속 유지하면서 새 값은 무시하고, 해당 키에 값이 할당되지 않을 경우에만 새 값을 추가하는 방법을 선택할 수도 있습니다. 또는 예전 값과 새 값을 조합할 수도 있습니다. 각각의 경우를 어떻게 할지 살펴봅시다!

값을 덮어쓰기

만일 해쉬맵에 키와 값을 삽입하고, 그 후 똑같은 키에 다른 값을 삽입하면, 키에 연관지어진 값은 새 값으로 대신될 것입니다. 아래 Listing 8-24의 코드가 `insert`를 두 번 호출함에도, 해쉬맵은 딱 하나의 키/값 쌍을 담게 될 것인데 그 이유는 두 번 모두 블루 팀의 키에 대한 값을 삽입하고 있기 때문입니다:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

println!("{:?}", scores);
```

Listing 8-24: 특정한 키로 저장된 값을 덮어쓰기

이 코드는 `{"Blue": 25}`를 출력할 것입니다. 원래의 값 10은 덮어써졌습니다.

키에 할당된 값이 없을 경우에만 삽입하기

특정 키가 값을 가지고 있는지 검사하고, 만일 가지고 있지 않다면 이 키에 대한 값을 삽입하고자 하는 경우는 흔히 발생합니다. 해쉬맵은 이를 위하여 `entry`라고 하는 특별한 API를 가지고 있는데, 이는 우리가 검사하고자 하는 키를 파라미터로 받습니다. `entry` 함수의 리턴값은 열거형 `Entry`인데, 해당 키가 있는지 혹은 없는지를 나타냅니다. 우리가 엘로우 팀에 대한 키가 연관된 값을 가지고 있는지 검사하고 싶어한다고 해봅시다. 만일 없다면, 값 50을 삽입하고, 블루팀에 대해서도 똑같이 하고 싶습니다. 엔트리 API를 사용한 코드는 아래의 Listing 8-25와 같습니다:

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{}: {}", "scores");
```

Listing 8-25: `entry` 메소드를 이용하여 어떤 키가 값을 이미 갖고 있지 않을 경우에만 추가하기

`Entry`에 대한 `or_insert` 메소드는 해당 키가 존재할 경우 관련된 `Entry` 키에 대한 값을 반환하도록 정의되어 있고, 그렇지 않을 경우에는 파라미터로 주어진 값을 해당 키에 대한 새 값을 삽입하고 수정된 `Entry`에 대한 값을 반환합니다. 이 방법은 우리가 직접 로직을 작성하는 것보다 훨씬 깔끔하고, 게다가 빌림 검사기와 잘 어울려 동작합니다.

Listing 8-25의 코드를 실행하면 `{"Yellow": 50, "Blue": 10}`를 출력할 것입니다. 첫번째 `entry` 호출은 옐로우 팀에 대한 키에 대하여 값 50을 삽입하는데, 이는 옐로우 팀이 값을 가지고 있지 않기 때문입니다. 두번째 `entry` 호출은 해쉬맵을 변경하지 않는데, 왜냐하면 블루 팀은 이미 값 10을 가지고 있기 때문입니다.

예전 값을 기초로 값을 갱신하기

해쉬맵에 대한 또 다른 흔한 사용 방식은 키에 대한 값을 찾아서 예전 값에 기초하여 값을 갱신하는 것입니다. 예를 들어, Listing 8-26은 어떤 텍스트 내에 각 단어가 몇번이나 나왔는지를 세는 코드를 보여줍니다. 단어를 키로 사용하는 해쉬맵을 이용하여 해당 단어가 몇번이나 나왔는지를 유지하기 위해 값을 증가시켜 줍니다. 만일 어떤 단어를 처음 본 것이라면, 값 `0`을 삽입할 것입니다.

```
use std::collections::HashMap;

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{}: {}", "map");
```

Listing 8-26: 단어와 횟수를 저장하는 해쉬맵을 사용하여 단어의 등장 횟수 세기

이 코드는 `{"world": 2, "hello": 1, "wonderful": 1}` 를 출력할 것입니다. `or_insert` 메소드는 실제로는 해당 키에 대한 값의 가변 참조자 (`&mut V`)를 반환합니다. 여기서는 `count` 변수에 가변 참조자를 저장하였고, 여기에 값을 할당하기 위해 먼저 애스터리스크 (`*`)를 사용하여 `count`를 역참조해야 합니다. 가변 참조자는 `for` 루프의 끝에서 스코프 밖으로 벗어나고, 따라서 모든 값들의 변경은 안전하며 빌림 규칙에 위배되지 않습니다.

해쉬 함수

기본적으로, `HashMap` 은 서비스 거부 공격(Denial of Service(DoS) attack)에 저항 기능을 제공할 수 있는 암호학적으로 보안되는 해쉬 함수를 사용합니다. 이는 사용 가능한 가장 빠른 해쉬 알고리즘은 아니지만, 성능을 떨어트리면서 더 나은 보안을 취하는 거래는 가치가 있습니다. 만일 여러분이 여러분의 코드를 프로파일하여 기본 해쉬 함수가 여러분의 목표에 관해서는 너무 느리다면, 다른 해쉬어(hasher)를 특정하여 다른 함수로 바꿀 수 있습니다. 해쉬어는 `BuildHasher` 트레이트를 구현한 타입을 말합니다. 트레이트와 이를 어떻게 구현하는지에 대해서는 10장에서 다를 것입니다. 여러분의 해쉬어를 바닥부터 새로 구현해야 할 필요는 없습니다; [crates.io](#)에서는 많은 수의 범용적인 해쉬 알고리즘을 구현한 해쉬어를 제공하는 공유 라이브러리를 제공합니다.

정리

벡터, 스트링, 그리고 해쉬맵은 프로그램 내에서 여러분이 데이터를 저장하고, 접근하고, 수정하고 싶어하는 곳마다 필요한 수많은 기능들을 제공해줄 것입니다. 이제 여러분이 풀 준비가 되어있어야 할만한 몇 가지 연습 문제를 소개합니다:

- 정수 리스트가 주어졌을 때, 벡터를 이용하여 이 리스트의 평균값(mean, average), 중간값(median, 정렬했을 때 가운데 위치한 값), 그리고 최빈값(mode, 가장 많이 발생한 값; 해쉬맵이 여기서 도움이 될 것입니다)를 반환해보세요.
- 스트링을 피그 라틴(pig Latin)으로 변경해보세요. 각 단어의 첫번째 자음은 단어의 끝으로 이동하고 “ay”를 붙이므로, “first”는 “irst-fay”가 됩니다. 모음으로 시작하는 단어는 대신 끝에 “hay”를 붙입니다. (“apple”은 “apple-hay”가 됩니다.) UTF-8 인코딩에 대해 기억하세요!
- 해쉬맵과 벡터를 이용하여, 사용자가 회사 내의 부서에 대한 피고용인 이름을 추가할 수 있도록 하는 텍스트 인터페이스를 만들어보세요. 예를들어 “Add Sally to Engineering”이나 “Add Amir to Sales” 같은 식으로요. 그후 사용자가 각 부서의 모든 사람들에 대한 리스트나 알파벳 순으로 정렬된 부서별 모든 사람에 대한 리스트를 조회할 수 있도록 해보세요.

표준 라이브러리 API 문서는 이 연습문제들에 대해 도움이 될만한 벡터, 스트링, 그리고 해쉬맵의 메소드들을 설명해줍니다!

우리는 연산이 실패할 수 있는 더 복잡한 프로그램으로 진입하고 있는 상황입니다; 따라서, 다음은 에러 처리에 대해 다룰 완벽한 시간이란 뜻이죠!

에러 처리

러스트의 신뢰성에 대한 약속은 에러 처리에도 확장되어 있습니다. 에러는 소프트웨어에서 피할 수 없는 현실이며, 따라서 러스트는 무언가 잘못되었을 경우에 대한 처리를 위한 몇 가지 기능을 갖추고 있습니다. 많은 경우, 러스트는 여러분이 에러가 발생할 가능성을 인정하고 여러분의 코드가 컴파일되기 전에 어떤 행동을 취하기를 요구할 것입니다. 이러한 요구사항은 여러분의 코드를 제품으로서 배포하기 전에 에러를 발견하고 적절히 조치할 것이라고 보장함으로써 여러분의 프로그램을 더 강건하게 해줍니다!

러스트는 에러를 두 가지 범주로 묶습니다: **복구 가능한(recoverable)** 에러와 **복구 불가능한(unrecoverable)** 에러입니다. 복구 가능한 에러는 사용자에게 문제를 보고하고 연산을 재시도하는 것이 보통 합리적인 경우인데, 이를테면 파일을 찾지 못하는 에러가 그렇습니다. 복구 불가능한 에러는 언제나 버그의 증상이 나타나는데, 예를 들면 배열의 끝을 넘어선 위치의 값에 접근하려고 시도하는 경우가 그렇습니다.

대부분의 언어들은 이 두 종류의 에러를 분간하지 않으며 예외 처리(exception)와 같은 메카니즘을 이용하여 같은 방식으로 둘 다 처리합니다. 러스트는 예외 처리 기능이 없습니다. 대신, 복구 가능한 에러를 위한 `Result<T, E>` 값과 복구 불가능한 에러가 발생했을 때 실행을 멈추는 `panic!` 매크로를 가지고 있습니다. 이번 장에서는 `panic!`을 호출하는 것을 먼저 다룬 뒤, `Result<T, E>` 값을 반환하는 것에 대해 이야기하겠습니다. 추가로, 에러로부터 복구를 시도할지 아니면 실행을 멈출지를 결정할 때 고려할 것에 대해 탐구해 보겠습니다.

panic!과 함께하는 복구 불가능한 에러

가끔씩 여러분의 코드에서 나쁜 일이 일어나고, 이에 대해 여러분이 할 수 있는 것이 없을 수도 있습니다. 이러한 경우를 위하여 러스트는 `panic!` 매크로를 가지고 있습니다. 이 매크로가 실행되면, 여러분의 프로그램은 실패 메세지를 출력하고, 스택을 되감고 청소하고, 그 후 종료됩니다. 이런 일이 발생하는 가장 흔한 상황은 어떤 종류의 버그가 발견되었고 프로그래머가 이 에러를 어떻게 처리할지가 명확하지 않을 때입니다.

panic!에 응하여 스택을 되감거나 그만두기

기본적으로, `panic!`이 발생하면, 프로그램은 *되감기(unwinding)*를 시작하는데, 이는 러스트가 패닉을 마주친 각 함수로부터 스택을 거꾸로 훑어가면서 데이터를 제거한다는 뜻이지만, 이 훑어가기 및 제거는 일이 많습니다. 다른 대안으로는 즉시 *그만두기(abort)*가 있는데, 이는 데이터 제거 없이 프로그램을 끝내는 것입니다. 프로그램이 사용하고 있던 메모리는 운영체제에 의해 청소될 필요가 있을 것입니다. 여러분의 프로젝트 내에서 결과 바이너리가 가능한 작아지기를 원한다면, 여러분의 `Cargo.toml` 내에서 적합한 `[profile]` 섹션에 `panic = 'abort'`를 추가함으로써 되감기를 그만두기로 바꿀 수 있습니다. 예를 들면, 여러분이 릴리즈 모드 내에서는 패닉 상에서 그만두기를 쓰고 싶다면, 다음을 추가하세요:

```
[profile.release]
panic = 'abort'
```

단순한 프로그램 내에서 `panic!` 호출을 시도해 봅시다:

Filename: src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

이 프로그램을 실행하면, 다음과 같은 것을 보게 될 것입니다:

```
$ cargo run
Compiling panic v0.1.0 (file:///projects/panic)
  Finished dev [unoptimized + debuginfo] target(s) in 0.25 secs
    Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/panic` (exit code: 101)
```

panic!의 호출이 마지막 세 줄의 에러 메세지를 야기합니다. 첫 번째 줄은 우리의 패닉 메세지와 소스 코드에서 패닉이 발생한 지점을 보여줍니다: *src/main.rs:2*는 *src/main.rs* 파일의 두 번째 줄을 가리킵니다.

위 예제의 경우, 가리키고 있는 줄은 우리 코드 부분이고, 해당 줄로 가면 **panic!** 매크로 호출을 보게 됩니다. 그 외의 경우들에서는, **panic!** 호출이 우리가 호출한 코드 내에 있을 수도 있습니다. 에러 메세지에 의해 보고되는 파일 이름과 라인 번호는 **panic!** 매크로가 호출된 다른 누군가의 코드일 것이며, 궁극적으로 **panic!**을 이끌어낸 것이 우리 코드 라인이 아닐 것입니다. 문제를 일으킨 코드 부분을 발견하기 위해서 **panic!** 호출이 발생된 함수에 대한 백트레이스(backtrace)를 사용할 수 있습니다. 백트레이스가 무엇인가에 대해서는 뒤에 더 자세히 다를 것입니다.

panic! 백트레이스 사용하기

다른 예를 통해서, 우리 코드가 직접 매크로를 호출하는 대신 우리 코드의 버그 때문에 **panic!** 호출이 라이브러리로부터 발생될 때는 어떻게 되는지 살펴봅시다. Listing 9-1은 벡터 내의 요소를 인덱스로 접근 시도하는 코드입니다:

Filename: *src/main.rs*

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];
}
```

Listing 9-1: **panic!**을 일으키는 벡터의 끝을 넘어선 요소에 대한 접근 시도

여기서 우리는 벡터의 100번째 요소(0부터 시작하여 100번째)에 접근하기를 시도하고 있지만, 벡터는 오직 3개의 요소만 가지고 있습니다. 이러한 상황이면 러스트는 패닉을 일으킬 것입니다. **[]**를 사용하는 것은 어떤 요소를 반환하기를 가정하지만, 유효하지 않은 인덱스를 넘기게 되면 러스트가 반환할 올바른 요소는 없습니다.

이러한 상황에서 C와 같은 다른 언어들은 여러분이 원하는 것이 아닐지라도, 여러분이 요청한 것을 정확히 주

려고 시도할 것입니다: 여러분은 벡터 내에 해당 요소와 상응하는 위치의 메모리에 들어 있는 무언가를 얻을 것입니다. 설령 그 메모리 영역이 벡터 소유가 아닐지라도 말이죠. 이러한 것을 **버퍼 오버리드(buffer overread)**라고 부르며, 만일 어떤 공격자가 읽도록 허용되어선 안 되지만 배열 뒤에 저장된 데이터를 읽어 낼 방법으로서 인덱스를 다룰 수 있게 된다면, 이는 보안 취약점을 발생시킬 수 있습니다.

여러분의 프로그램을 이러한 종류의 취약점으로부터 보호하기 위해서, 여러분이 존재하지 않는 인덱스 상의 요소를 읽으려 시도한다면, 러스트는 실행을 멈추고 계속하기를 거부할 것입니다. 한번 시도해 봅시다:

```
$ cargo run
Compiling panic v0.1.0 (file:///projects/panic)
Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
100', /stable-dist-rustc/build/src/libcollections/vec.rs:1362
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/panic` (exit code:
101)
```

위 에러는 우리가 작성하지 않은 파일인 *libcollections/vec.rs*를 가리키고 있습니다. 이는 표준 라이브러리 내에 있는 **Vec<T>**의 구현 부분입니다. 우리가 벡터 **v**에 **[]**를 사용할 때 실행되는 코드는 *libcollections/vec.rs* 안에 있으며, 그곳이 바로 **panic!**이 실제 발생한 곳입니다.

그 다음 노트는 **RUST_BACKTRACE** 환경 변수를 설정하여 에러의 원인이 된 것이 무엇인지 정확하게 백트레이스할 수 있다고 말해주고 있습니다. 백트레이스 (*backtrace*)란 어떤 지점에 도달하기까지 호출해온 모든 함수의 리스트를 말합니다. 러스트의 백트레이스는 다른 언어들에서와 마찬가지로 동작합니다: 백트레이스를 읽는 요령은 위에서부터 시작하여 여러분이 작성한 파일이 보일 때까지 읽는 것입니다. 그곳이 바로 문제를 일으킨 지점입니다. 여러분의 파일을 언급한 줄보다 위에 있는 줄들은 여러분의 코드가 호출한 코드입니다; 밑의 코드는 여러분의 코드를 호출한 코드입니다. 이 줄들은 핵심(core) 러스트 코드, 표준 라이브러리, 혹은 여러분이 이용하고 있는 크레이트를 포함하고 있을지도 모릅니다. 백트레이스를 얻어내는 시도를 해봅시다: Listing 9-2는 여러분이 보게 될 것과 유사한 출력을 보여줍니다:

```
$ RUST_BACKTRACE=1 cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
100', /stable-dist-rustc/build/src/libcollections/vec.rs:1392
stack backtrace:
1:      0x560ed90ec04c -
std::sys::imp::backtrace::tracing::imp::write::hf33ae72d0baa11ed
    at /stable-dist-
rustc/build/src/libstd/sys/unix/backtrace/tracing/gcc_s.rs:42
2:      0x560ed90ee03e - std::panicking::default_hook::{{closure}}::h59672b733cc6a455
    at /stable-dist-
```

```

rustc/build/src/libstd/panicking.rs:351
 3:      0x560ed90edc44 - std::panicking::default_hook::h1670459d2f3f8843
          at /stable-dist-
rustc/build/src/libstd/panicking.rs:367
 4:      0x560ed90ee41b -
std::panicking::rust_panic_with_hook::hcf0ddb069e7abcd7
          at /stable-dist-
rustc/build/src/libstd/panicking.rs:555
 5:      0x560ed90ee2b4 - std::panicking::begin_panic::hd6eb68e27bdf6140
          at /stable-dist-
rustc/build/src/libstd/panicking.rs:517
 6:      0x560ed90ee1d9 - std::panicking::begin_panic_fmt::abcd5965948b877f8
          at /stable-dist-
rustc/build/src/libstd/panicking.rs:501
 7:      0x560ed90ee167 - rust_begin_unwind
          at /stable-dist-
rustc/build/src/libstd/panicking.rs:477
 8:      0x560ed911401d - core::panicking::panic_fmt::hc0f6d7b2c300cdd9
          at /stable-dist-
rustc/build/src/libcore/panicking.rs:69
 9:      0x560ed9113fc8 -
core::panicking::panic_bounds_check::h02a4af86d01b3e96
          at /stable-dist-
rustc/build/src/libcore/panicking.rs:56
10:      0x560ed90e71c5 - <collections::vec::Vec<T> as
core::ops::Index<usize>>::index::h98abcd4e2a74c41
          at /stable-dist-
rustc/build/src/libcollections/vec.rs:1392
11:      0x560ed90e727a - panic::main::h5d6b77c20526bc35
          at /home/you/projects/panic/src/main.rs:4
12:      0x560ed90f5d6a - __rust_maybe_catch_panic
          at /stable-dist-
rustc/build/src/libpanic_unwind/lib.rs:98
13:      0x560ed90ee926 - std::rt::lang_start::hd7c880a37a646e81
          at /stable-dist-
rustc/build/src/libstd/panicking.rs:436
          at /stable-dist-rustc/build/src/libstd/panic.rs:361
          at /stable-dist-rustc/build/src/libstd/rt.rs:57
14:      0x560ed90e7302 - main
15:      0x7f0d53f16400 - __libc_start_main
16:      0x560ed90e6659 - _start
17:      0x0 - <unknown>

```

Listing 9-2: 환경 변수 `RUST_BACKTRACE` 가 설정되었을 때 `panic!` 의 호출에 의해 발생되는 백트레이스 출력

출력이 엄청 많군요! 여러분이 보는 실제 출력값은 여러분의 운영 체제 및 러스트 버전에 따라 다를 수 있습니다. 이러한 정보들과 함께 백트레이스를 얻기 위해서는 디버그 심볼이 활성화되어 있어야 합니다. 디버그 심볼은 여기서와 마찬가지로 여러분이 `cargo build` 나 `cargo run` 을 `--release` 플래그 없이 실행했을

때 기본적으로 활성화됩니다.

Listing 9-2의 출력값 내에서, 백트레이스의 11번 라인이 문제를 일으킨 우리 프로젝트의 라인을 가리키고 있습니다: 바로 `src/main.rs`, 4번 라인입니다. 만일 프로그램이 패닉에 빠지지 않도록 하고 싶다면, 우리가 작성한 파일이 언급된 첫 라인으로 지적된 위치가 바로 패닉을 일으킨 값을 가지고 있는 위치를 찾아내기 위해 수사하기 시작할 지점입니다. 백트레이스를 어떻게 사용하는지 시범을 보이기 위해 고의로 패닉을 일으키는 코드를 작성한 우리의 예제에서, 패닉을 고칠 방법은 고작 3개의 아이템을 가진 벡터로부터 인덱스 100에서의 요소를 요청하지 않도록 하는 것입니다. 여러분의 코드가 추후 패닉에 빠졌을 때, 여러분의 특정한 경우에 대하여 어떤 코드가 패닉을 일으키는 값을 만드는지와 코드는 대신 어떻게 되어야 할지를 알아낼 필요가 있을 것입니다.

우리는 `panic!`으로 다시 돌아올 것이며 언제 `panic!`을 써야 하는지, 혹은 쓰지 말아야 하는지에 대해 이 장의 뒷부분에서 알아보겠습니다. 다음으로 `Result`를 이용하여 에러로부터 어떻게 복구하는지를 보겠습니다.

Result와 함께하는 복구 가능한 에러

대부분의 에러는 프로그램을 전부 멈추도록 요구될 정도로 심각하지는 않습니다. 종종 어떤 함수가 실패할 때는, 우리가 쉽게 해석하고 대응할 수 있는 이유에 대한 것입니다. 예를 들어, 만일 우리가 어떤 파일을 여는데 해당 파일이 존재하지 않아서 연산에 실패했다면, 프로세스를 멈추는 대신 파일을 새로 만드는 것을 원할지도 모릅니다.

2장의 “[Result 타입으로 잠재된 실패 다루기](#)” 절에서 `Result` 열거형은 다음과 같이 `Ok` 와 `Err` 라는 두 개의 variant를 갖도록 정의되어 있음을 상기하세요:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`T`와 `E`는 제네릭 타입 파라미터입니다; 10장에서 제네릭에 대해 더 자세히 다룰 것입니다. 지금으로서 여러분이 알아둘 필요가 있는 것은, `T`는 성공한 경우에 `ok` variant 내에 반환될 값의 타입을 나타내고 `E`는 실패한 경우에 `Err` variant 내에 반환될 에러의 타입을 나타내는 것이라는 점입니다. `Result`가 이러한 제네릭 타입 파라미터를 갖기 때문에, 우리가 반환하고자 하는 성공적인 값과 에러 값이 다를 수 있는 다양한 상황 내에서 표준 라이브러리에 정의된 `Result` 타입과 함수들을 사용할 수 있습니다.

실패할 수도 있기 때문에 `Result` 값을 반환하는 함수를 호출해 봅시다: Listing 9-3에서는 파일 열기를 시도합니다:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
}
```

Listing 9-3: 파일 열기

`File::open`이 `Result`를 반환하는지 어떻게 알까요? 표준 라이브러리 API 문서를 찾아보거나, 컴파일러에게 물어볼 수 있습니다! 만일 `f`에게 우리가 알고 있고 그 함수의 반환 타입은 아닐 어떤 타입에 대한 타입 명시를 주고 그 코드의 컴파일을 시도한다면, 컴파일러는 우리에게 타입이 맞지 않는다고 알려줄 것입니다. 그 후 에러 메세지는 `f`의 타입이 무엇인지 알려줄 것입니다. 한번 해봅시다: 우리는 `File::open`의 반환 타입이 `u32`는 아니라는 것을 알고 있으니, `let f` 구문을 이렇게 바꿔봅시다:

```
let f: u32 = File::open("hello.txt");
```

이제 컴파일을 시도하면 다음 메세지가 나타납니다:

```
error[E0308]: mismatched types
--> src/main.rs:4:18
 |
4 |     let f: u32 = File::open("hello.txt");
|           ^^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
`std::result::Result`
 |
= note: expected type `u32`
= note:    found type `std::result::Result<std::fs::File, std::io::Error>`
```

이 메세지는 `File::open` 함수의 반환 타입이 `Result<T, E>`라는 것을 알려줍니다. 여기서 제네릭 파라미터 `T`는 성공값의 타입인 `std::fs::File`로 채워져 있는데, 이것은 파일 핸들입니다. 에러에 사용되는 `E`의 타입은 `std::io::Error`입니다.

이 반환 타입은 `File::open`을 호출하는 것이 성공하여 우리가 읽거나 쓸 수 있는 파일 핸들을 반환해 줄 수도 있다는 뜻입니다. 함수 호출은 또한 실패할 수도 있습니다: 예를 들면 파일이 존재하지 않거나 파일에 접근할 권한이 없을지도 모릅니다. `File::open` 함수는 우리에게 성공했는지 혹은 실패했는지를 알려주면서 동시에 파일 핸들이나 에러 정보 둘 중 하나를 우리에게 제공할 방법을 가질 필요가 있습니다. 바로 이러한 정보가 `Result` 열거형이 전달하는 것과 정확히 일치합니다.

`File::open`이 성공한 경우, 변수 `f`가 가지게 될 값은 파일 핸들을 담고 있는 `Ok` 인스턴스가 될 것입니다. 실패한 경우, `f`의 값은 발생한 에러의 종류에 대한 더 많은 정보를 가지고 있는 `Err`의 인스턴스가 될 것입니다.

우리는 Listing 9-3의 코드에 `File::open`이 반환하는 값에 따라 다른 행동을 취하는 코드를 추가할 필요가 있습니다. Listing 9-4은 우리가 6장에서 다뤘던 기초 도구 `match` 표현식을 이용하여 `Result`를 처리하는 한 가지 방법을 보여줍니다:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```

Listing 9-4: `match` 표현식을 사용하여 발생 가능한 `Result` variant들을 처리하기

`Option` 열거형과 같이 `Result` 열거형과 variant들은 프렐루드(prelude)로부터 가져와진다는 점을 기억하세요. 따라서 `match`의 각 경우에 대해서 `Ok`와 `Err` 앞에 `Result::`를 특정하지 않아도 됩니다.

여기서 우리는 러스트에게 결과가 `Ok`일 때에는 `Ok` variant로부터 내부의 `file` 값을 반환하고, 이 파일 핸들 값을 변수 `f`에 대입한다고 말해주고 있습니다. `match` 이후에는 읽거나 쓰기 위해 이 파일 핸들을 사용할 수 있습니다.

`match`의 다른 경우는 `File::open`으로부터 `Err`를 얻은 경우를 처리합니다. 이 예제에서는 `panic!` 매크로를 호출하는 방법을 택했습니다. 우리의 현재 디렉토리 내에 `hello.txt`라는 이름의 파일이 없는데 이 코드를 실행하게 되면, `panic!` 매크로로부터 다음과 같은 출력을 보게 될 것입니다:

```
thread 'main' panicked at 'There was a problem opening the file: Error {  
    repr:  
        Os { code: 2, message: "No such file or directory" } }', src/main.rs:9:12
```

늘 그렇듯이, 이 출력은 어떤 것이 잘못되었는지 정확히 알려줍니다.

서로 다른 에러에 대해 매칭하기

Listing 9-3의 코드는 `File::open`이 실패한 이유가 무엇이든 간에 `panic!`을 일으킬 것입니다. 대신 우리가 원하는 것은 실패 이유에 따라 다른 행동을 취하는 것입니다: 파일이 없어서 `File::open`이 실패한 것이라면, 새로운 파일을 만들어서 핸들을 반환하고 싶습니다. 만일 그 밖의 이유로 `File::open`이 실패한 거라면, 예를 들어 파일을 열 권한이 없어서라면, Listing 9-4에서 했던 것과 마찬가지로 `panic!`을 일으키고 싶습니다. `match`에 새로운 경우를 추가한 Listing 9-5를 봅시다:

Filename: src/main.rs

```

use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(ref error) if error.kind() == ErrorKind::NotFound => {
            match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => {
                    panic!(
                        "Tried to create file but there was a problem: {:?}",
                        e
                    )
                },
            }
        },
        Err(error) => {
            panic!(
                "There was a problem opening the file: {:?}",
                error
            )
        },
    };
}

```

Listing 9-5: 다른 종류의 에러를 다른 방식으로 처리하기

`Err` variant 내에 있는 `File::open`이 반환하는 값의 타입은 `io::Error`인데, 이는 표준 라이브러리에서 제공하는 구조체입니다. 이 구조체는 `kind` 메소드를 제공하는데 이를 호출하여 `io::ErrorKind` 값을 얻을 수 있습니다. `io::ErrorKind`는 `io` 연산으로부터 발생할 수 있는 여러 종류의 에러를 표현하는 variant를 가진, 표준 라이브러리에서 제공하는 열거형입니다. 우리가 사용하고자 하는 variant는 `ErrorKind::NotFound`인데, 이는 열고자 하는 파일이 아직 존재하지 않음을 나타냅니다.

조건문 `if error.kind() == ErrorKind::NotFound`는 매치 가드(match guard)라고 부릅니다: 이는 `match` 줄기 상에서 줄기의 패턴을 좀 더 정제해주는 추가 조건문입니다. 그 줄기의 코드가 실행되기 위해서는 이 조건문이 참이어야 합니다; 그렇지 않다면, 패턴 매칭은 `match`의 다음 줄기에 맞춰보기 위해 이동할 것입니다. 패턴에는 `ref`가 필요하며 그럼으로써 `error`가 가드 조건문으로 소유권 이동이 되지 않고 그저 참조만 됩니다. 패턴 내에서 참조자를 얻기 위해 &대신 `ref`가 사용되는 이유는 18장에서 자세히 다룰 것입니다. 짧게 설명하면, &는 참조자를 매치하고 그 값을 제공하지만, `ref`는 값을 매치하여 그 참조자를 제공합니다.

매치 가드 내에서 확인하고자 하는 조건문은 `error.kind()`에 의해 반환된 값이 `ErrorKind` 열거형의

`NotFound` variant인가 하는 것입니다. 만일 그렇다면, `File::create`로 파일 생성을 시도합니다. 그러나, `File::create` 또한 실패할 수 있기 때문에, 안쪽에 `match` 구문을 바깥쪽과 마찬가지로 추가할 필요가 있습니다. 파일이 열 수 없을 때, 다른 에러 메세지가 출력될 것입니다. 바깥쪽 `match`의 마지막 갈래는 똑같이 남아서, 파일을 못 찾는 에러 외에 다른 어떤 에러에 대해서도 패닉을 일으킵니다.

에러가 났을 때 패닉을 위한 속컷: `unwrap`과 `expect`

`match`의 사용은 충분히 잘 동작하지만, 살짝 장황하기도 하고 의도를 항상 잘 전달하는 것도 아닙니다. `Result<T, E>` 타입은 다양한 작업을 하기 위해 정의된 수많은 헬퍼 메소드를 가지고 있습니다. 그 중 하나인 `unwrap` 이라 부르는 메소드는 Listing 9-4에서 작성한 `match` 구문과 비슷한 구현을 한 속컷 메소드입니다. 만일 `Result` 값이 `Ok` variant라면, `unwrap`은 `Ok` 내의 값을 반환할 것입니다. 만일 `Result` 가 `Err` variant라면, `unwrap`은 우리를 위해 `panic!` 매크로를 호출할 것입니다. 아래에 `unwrap`이 작동하는 예가 있습니다:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

`hello.txt` 파일이 없는 상태에서 이 코드를 실행시키면, `unwrap` 메소드에 의한 `panic!` 호출로부터의 에러 메세지를 보게 될 것입니다:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error
{
  repr: Os { code: 2, message: "No such file or directory" } }',
/stable-dist-rustc/build/src/libcore/result.rs:868
```

또 다른 메소드인 `expect`는 `unwrap`과 유사한데, 우리가 `panic!` 에러 메세지를 선택할 수 있게 해줍니다. `unwrap` 대신 `expect`를 이용하고 좋은 에러 메세지를 제공하는 것은 여러분의 의도를 전달해주고 패닉의 근원을 추적하는 걸 쉽게 해 줄 수 있습니다. `expect`의 문법은 아래와 같이 생겼습니다:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

`expect`는 `unwrap`과 같은 식으로 사용됩니다: 파일 핸들을 리턴하거나 `panic!` 매크로를 호출하는 것 이죠. `expect`가 `panic!` 호출에 사용하는 에러 메세지는 `unwrap`이 사용하는 기본 `panic!` 메세지보다는 `expect`에 넘기는 파라미터로 설정될 것입니다. 아래에 어떻게 생겼는지에 대한 예가 있습니다:

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code: 2, message: "No such file or directory" } }',
/stable-dist-rustc/build/src/libcore/result.rs:868
```

이 에러 메세지는 우리가 특정한 텍스트인 `Failed to open hello.txt`로 시작하기 때문에, 이 에러 메세지가 어디서부터 왔는지를 코드 내에서 찾기가 더 수월해질 것입니다. 만일 우리가 여러 군데에 `unwrap`을 사용하면, 정확히 어떤 `unwrap`이 패닉을 일으켰는지 찾기에 좀 더 많은 시간이 걸릴 수 있는데, 그 이유는 패닉을 호출하는 모든 `unwrap`이 동일한 메세지를 출력하기 때문입니다.

에러 전파하기

실패할지도 모르는 무언가를 호출하는 구현을 가진 함수를 작성할 때, 이 함수 내에서 에러를 처리하는 대신, 에러를 호출하는 코드 쪽으로 반환하여 그쪽에서 어떻게 할지 결정하도록 할 수 있습니다. 이는 에러 전파하기로 알려져 있으며, 에러가 어떻게 처리해야 좋을지 좌우해야 할 상황에서, 여러분의 코드 내용 내에서 이용 가능한 것들보다 더 많은 정보와 로직을 가지고 있을 수도 있는 호출하는 코드 쪽에 더 많은 제어권을 줍니다.

예를 들면, Listing 9-6은 파일로부터 사용자 이름을 읽는 함수를 작성한 것입니다. 만일 파일이 존재하지 않거나 읽을 수 없다면, 이 함수는 호출하는 코드 쪽으로 해당 에러를 반환할 것입니다:

Filename: src/main.rs

```

use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}

```

Listing 9-6: `match`를 이용하여 호출 코드 쪽으로 에러를 반환하는 함수

함수의 반환 타입부터 먼저 살펴봅시다: `Result<String, io::Error>`. 이는 함수가 `Result<T, E>` 타입의 값을 반환하는데 제네릭 파라미터 `T`는 구체적 타입(concrete type)인 `String`로 채워져 있고, 제네릭 타입 `E`는 구체적 타입인 `io::Error`로 채워져 있습니다. 만일 이 함수가 어떤 문제 없이 성공하면, 함수를 호출한 코드는 `String`을 담은 값을 받을 것입니다 - 이 함수가 파일로부터 읽어들인 사용자 이름이겠지요. 만일 어떤 문제가 발생한다면, 이 함수를 호출한 코드는 문제가 무엇이었는지에 대한 더 많은 정보를 담고 있는 `io::Error`의 인스턴스를 담은 `Err` 값을 받을 것입니다. 이 함수의 반환 타입으로서 `io::Error`를 선택했는데, 그 이유는 우리가 이 함수 내부에서 호출하고 있는 실패 가능한 연산 두 가지가 모두 이 타입의 에러 값을 반환하기 때문입니다: `File::open` 함수와 `read_to_string` 메소드 말이죠.

함수의 본체는 `File::open` 함수를 호출하면서 시작합니다. 그다음에는 Listing 9-4에서 본 `match`와 유사한 식으로 `match`을 이용해서 `Result` 값을 처리하는데, `Err` 경우에 `panic!`을 호출하는 대신 이 함수를 일찍 끝내고 `File::open`으로부터의 에러 값을 마치 이 함수의 에러 값인 것처럼 호출하는 쪽의 코드에게 전달합니다. 만일 `File::open`이 성공하면, 파일 핸들을 `f`에 저장하고 계속합니다.

그 뒤 변수 `s`에 새로운 `String`을 생성하고 파일의 콘텐츠를 읽어 `s`에 넣기 위해 `f`에 있는 파일 핸들의 `read_to_string` 메소드를 호출합니다. `File::open`가 성공하더라도 `read_to_string` 메소드가 실패할 수 있기 때문에 이 함수 또한 `Result`를 반환합니다. 따라서 이 `Result`를 처리하기 위해서 또 다른 `match`가 필요합니다: 만일 `read_to_string`이 성공하면, 우리의 함수가 성공한 것이고, 이제 `s` 안에 있는 파일로부터 읽어들인 사용자 이름을 `Ok`에 싸서 반환합니다. 만일 `read_to_string`이 실패하면, `File::open`의 반환값을 처리했던 `match`에서 에러값을 반환하는 것과 같은 방식으로 에러 값을 반환합니다. 하지만 여기서는 명시적으로 `return`이라 말할 필요는 없는데, 그 이유는 이 함수의 마지막 표현식이

기 때문입니다.

그러면 이 코드를 호출하는 코드는 사용자 이름을 담은 `Ok` 값 혹은 `io::Error`를 담은 `Err` 값을 얻는 처리를하게 될 것입니다. 호출하는 코드가 이 값을 가지고 어떤 일을 할 것인지 우리는 알지 못합니다. 만일 그쪽에서 `Err` 값을 얻었다면, 예를 들면 `panic!`을 호출하여 프로그램을 종료시키는 선택을 할 수도 있고, 기본 사용자 이름을 사용할 수도 있으며, 혹은 파일이 아닌 다른 어딘가에서 사용자 이름을 찾을 수도 있습니다. 호출하는 코드가 정확히 어떤 것을 시도하려 하는지에 대한 충분한 정보가 없기 때문에, 우리는 모든 성공 혹은 에러 정보를 위로 전파하여 호출하는 코드가 적절하게 처리를 하도록 합니다.

러스트에서 에러를 전파하는 패턴은 너무 흔하여 러스트에서는 이를 더 쉽게 해주는 물음표 연산자 `?`를 제공합니다.

에러를 전파하기 위한 속컷: `?`

Listing 9-7은 Listing 9-6과 같은 기능을 가진 `read_username_from_file`의 구현을 보여주는데, 다만 이 구현은 물음표 연산자를 이용하고 있습니다:

Filename: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

Listing 9-7: `?`를 이용하여 에러를 호출하는 코드 쪽으로 반환하는 함수

`Result` 값 뒤의 `?`는 Listing 9-6에서 `Result` 값을 다루기 위해 정의했던 `match` 표현식과 거의 같은 방식으로 동작하게끔 정의되어 있습니다. 만일 `Result`의 값이 `Ok`라면, `Ok` 내의 값이 이 표현식으로부터 얻어지고 프로그램이 계속됩니다. 만일 값이 `Err`라면, 우리가 `return` 키워드를 사용하여 에러 값을 호출하는 코드에게 전파하는 것과 같이 전체 함수로부터 `Err` 내의 값이 반환될 것입니다.

Listing 9-6에 있는 `match` 표현식과 물음표 연산자가 수행하는 한 가지 차이점은 물음표 연산자를 사용할 때 에러 값들이 표준 라이브러리 내에 있는 `From` 트레이트에 정의된 `from` 함수를 친다는 것입니다. 많은 에러 타입들이 어떤 타입의 에러를 다음 타입의 에러로 변환하기 위해 `from` 함수를 구현하였습니다. 물음표 연산자가 사용되면, `from` 함수의 호출이 물음표 연산자가 얻게 되는 에러 타입을 `?`이 사용되고 있는 현재

함수의 반환 타입에 정의된 에러 타입으로 변환합니다. 이는 어떤 함수의 부분들이 수많은 다른 이유로 인해 실패할 수 있지만 이 함수는 실패하는 모든 방식을 하나의 에러 타입으로 반환할 때 유용합니다. 각각의 에러 타입이 그 자신을 반환되는 에러 타입으로 변경할 방법을 정의하기 위해 `from` 함수를 구현하기만 한다면, 물음표 연산자는 이 변환을 자동적으로 다룹니다.

Listing 9-7의 내용에서, `File::open` 호출 부분의 끝에 있는 `?`는 `Ok` 내의 값을 변수 `f`에게 반환해줄 것입니다. 만일 에러가 발생하면 `?`는 전체 함수로부터 일찍 빠져나와 호출하는 코드에게 어떤 `Err` 값을 줄 것입니다. `read_to_string` 호출의 끝부분에 있는 `?`도 같은 것이 적용되어 있습니다.

`?`는 많은 수의 보일러플레이트(boilerplate)를 제거해주고 이 함수의 구현을 더 단순하게 만들어 줍니다. 심지어는 Listing 9-8과 같이 `?` 뒤에 바로 메소드 호출을 연결하는 식으로 (chaining) 이 코드를 더 줄일 수도 있습니다:

Filename: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

Listing 9-8: 물음표 연산자 뒤에 메소드 호출을 연결하기

새로운 `String`을 만들어 `s`에 넣는 부분을 함수의 시작 부분으로 옮겼습니다; 이 부분은 달라진 것이 없습니다. `f` 변수를 만드는 대신, `File::open("hello.txt")?`의 결과 바로 뒤에 `read_to_string`의 호출을 연결시켰습니다. `read_to_string` 호출의 끝에는 여전히 `?`가 남아있고, `File::open`과 `read_to_string`이 모두 에러를 반환하지 않고 성공할 때 `s` 안의 사용자 이름을 담은 `Ok`를 여전히 반환합니다. 함수의 기능 또한 Listing 9-6과 Listing 9-7의 것과 동일하고, 다만 작성하기에 더 인체공학적인 방법이라는 차이만 있을 뿐입니다.

?는 Result를 반환하는 함수에서만 사용될 수 있습니다

`?`는 `Result` 타입을 반환하는 함수에서만 사용이 가능한데, 이것이 Listing 9-6에 정의된 `match` 표현식과 동일한 방식으로 동작하도록 정의되어 있기 때문입니다. `Result` 반환 타입을 요구하는 `match` 부분은 `return Err(e)`이며, 따라서 함수의 반환 타입은 반드시 이 `return`과 호환 가능한 `Result`가 되어야

합니다.

`main`의 반환 타입이 `()`라는 것을 상기하면서, 만약 `main` 함수 내에서 `?`를 사용하면 어떤일이 생길지 살펴봅시다:

```
use std::fs::File;  
  
fn main() {  
    let f = File::open("hello.txt")?;  
}
```

이걸 컴파일하면, 아래와 같은 에러 메세지가 뜹니다:

```
error[E0277]: the `?` operator can only be used in a function that returns
`Result` (or another type that implements `std::ops::Try`)
--> src/main.rs:4:13
|
4 |     let f = File::open("hello.txt")?;
|     -----|
|         |
|             cannot use the `?` operator in a function that returns `()``in
|             this macro invocation
= help: the trait `std::ops::Try` is not implemented for `()``in
= note: required by `std::ops::Try::from_error`
```

이 에러는 오직 **Result** 를 반환하는 함수 내에서만 물음표 연산자를 사용할 수 있음을 지적합니다.

Result을 반환하지 않는 함수 내에서, 여러분이 **Result**을 반환하는 다른 함수를 호출했을 때, 여러분은 **?**을 사용하여 호출하는 코드에게 잠재적으로 에러를 전파하는 대신 **match**나 **Result**에서 제공하는 메소드들 중 하나를 사용하여 이를 처리할 필요가 있을 것입니다.

panic!을 호출하거나 **Result**를 반환하는 것의 자세한 부분을 논의했으니, 어떤 경우에 어떤 방법을 사용하는 것이 적합할지를 어떻게 결정하는가에 대한 주제로 돌아갑시다.

panic!이냐, panic!이 아니냐, 그것이 문제로다

그러면 언제 `panic!`을 써야 하고 언제 `Result`를 반환할지 어떻게 결정해야 할까요? 코드가 패닉을 일으킬 때는 복구할 방법이 없습니다. 복구 가능한 방법이 있든 혹은 그렇지 않든 여러분은 어떤 에러 상황에 대해 `panic!`을 호출할 수 있지만, 그렇다면 여러분은 여러분의 코드를 호출하는 코드를 대신하여 현 상황은 복구 불가능한 것이라고 결정을 내리는 겁니다. 여러분이 `Result` 값을 반환하는 선택을 한다면, 호출하는 코드에게 결단을 내려주기보다는 옵션을 제공하는 것입니다. 그들은 그들의 상황에 적합한 방식으로 복구를 시도 할 수도 있고, 혹은 현재 상황의 `Err`은 복구 불가능하다고 사실상 결론을 내려서 `panic!`을 호출하여 여러분이 만든 복구 가능한 에러를 복구 불가능한 것으로 바꿔놓을 수도 있습니다. 그러므로, 여러분이 실패할지도 모르는 함수를 정의할 때는 `Result`을 반환하는 것이 기본적으로 좋은 선택입니다.

몇 가지 상황에서는 `Result`를 반환하는 대신 패닉을 일으키는 코드를 작성하는 것이 더 적합하지만, 덜 일반적입니다. 예제, 프로토타입 코드 및 테스트의 경우에는 왜 패닉이 더 좋은지를 탐구합시다; 그다음, 사람으로서의 여러분이라면 실패할 리 없는 메소드라는 것을 알 수 있지만 컴파일러는 이유를 파악할 수 없는 경우도 봅시다; 그리고 라이브러리 코드에 패닉을 추가해야 할지 말지를 어떻게 결정할까에 대한 일반적인 가이드 라인을 내림으로서 결론지어 봅시다.

예제, 프로토타입 코드, 그리고 테스트는 전부 패닉을 일으켜도 완전 괜찮은 곳입니다

여러분이 어떤 개념을 그려내기 위한 예제를 작성 중이라면, 강건한 에러 처리 코드를 예제 안에 넣는 것은 또한 예제를 덜 깨끗하게 만들 수 있습니다. 예제 코드 내에서는 `panic!`을 일으킬 수 있는 `unwrap` 같은 메소드를 호출하는 것이 여러분의 어플리케이션이 에러를 처리하고자 하는 방법에 대한 플레이스홀더로서의 의미를 갖는데, 이는 여러분의 코드의 나머지 부분이 어떤 것을 하는지에 따라 달라질 수 있습니다.

비슷한 상황에서, 여러분이 에러를 어떻게 처리할지 결정할 준비가 되기 전에는, `unwrap`과 `expect` 메소드가 프로토타이핑을 할 때 매우 편리합니다. 이 함수들은 여러분의 코드를 더 강건하게 만들 준비가 되었을 때를 위해서 명확한 표시를 남겨 둡니다.

만일 테스트 내에서 메소드 호출이 실패한다면, 해당 메소드가 테스트 중인 기능이 아니더라도 전체 테스트가 실패하는 게 좋을 것입니다. `panic!`이 테스트를 실패시키는 방법이기 때문에, `unwrap`이나 `expect`를 호출하는 것은 정확하게 하고자 하는 일과 일치합니다.

컴파일러보다 여러분이 더 많은 정보를 가지고 있을 때

`Result` 가 `Ok` 값을 가지고 있을 거라 확신할 다른 논리를 가지고 있지만, 그 논리가 컴파일러에 의해 이해 할 수 있는 것이 아닐 때라면, `unwrap`을 호출하는 것이 또한 적절할 수 있습니다. 여러분은 여전히 처리할 필요가 있는 `Result` 값을 가지고 있습니다: 여러분의 특정한 상황에서 논리적으로 불가능할지라도, 여러분이 호출하고 있는 연산이 무엇이든 간에 일반적으로는 여전히 실패할 가능성이 있습니다. 만일 여러분이 수동

적으로 `Err` variant를 결코 발생시키지 않는 코드를 조사하여 확신할 수 있다면, `unwrap`을 호출하는 것이 완벽히 허용됩니다. 여기 예제가 있습니다:

```
use std::net::IpAddr;

let home = "127.0.0.1".parse::<IpAddr>().unwrap();
```

여기서는 하드코딩된 스트링을 파싱하여 `IpAddr` 인스턴스를 만드는 중입니다. 우리는 `127.0.0.1`이 유효한 IP 주소임을 볼 수 있으므로, 여기서 `unwrap`을 사용하는 것은 허용됩니다. 그러나, 하드코딩된 유효한 스트링을 갖고 있다는 것이 `parse` 메소드의 반환 타입을 변경해주지는 않습니다: 우리는 여전히 `Result` 값을 갖게 되고, 컴파일러는 마치 `Err` variant가 나올 가능성이 여전히 있는 것처럼 우리가 `Result`를 처리하도록 할 것인데, 그 이유는 이 스트링이 항상 유효한 IP 주소라는 것을 알 수 있을 만큼 컴파일러가 똑똑하지는 않기 때문입니다. 만일 IP 주소 스트링이 프로그램 내에 하드코딩된 것이 아니라 사용자로부터 입력되었다면, 그래서 실패할 가능성이 생겼다면, 우리는 대신 더 강건한 방식으로 `Result`를 처리할 필요가 분명히 있습니다.

에러 처리를 위한 가이드라인

여러분의 코드가 결국 나쁜 상태에 처하게 될 가능성이 있을 때는 여러분의 코드에 `panic!`을 넣는 것이 바람직합니다. 이 글에서 말하는 나쁜 상태란 어떤 가정, 보장, 계약, 혹은 불변성이 깨질 때를 뜻하는 것으로, 이를테면 유효하지 않은 값이나 모순되는 값, 혹은 찾을 수 없는 값이 여러분의 코드를 통과할 경우를 말합니다 - 아래에 쓰여진 상황 중 하나 혹은 그 이상일 경우라면 말이죠:

- 이 나쁜 상태란 것이 가끔 벌어질 것으로 예상되는 무언가가 아닙니다.
- 그 시점 이후의 코드는 이 나쁜 상태에 있지 않아야만 할 필요가 있습니다.
- 여러분이 사용하고 있는 타입 내에 이 정보를 집어 넣을만한 뾰족한 수가 없습니다.

만일 어떤 사람이 여러분의 코드를 호출하고 타당하지 않은 값을 집어넣었다면, `panic!`을 써서 여러분의 라이브러리를 사용하고 있는 사람에게 그들의 코드 내의 버그를 알려서 개발하는 동안 이를 고칠 수 있게끔 하는 것이 최선책일 수도 있습니다. 비슷한 식으로, 만일 여러분의 제어권을 벗어난 외부 코드를 호출하고 있고, 이것이 여러분이 고칠 방법이 없는 유효하지 않은 상태를 반환한다면, `panic!`이 종종 적합합니다.

나쁜 상태에 도달했지만, 여러분이 얼마나 코드를 잘 작성했든 간에 일어날 것으로 예상될 때라면 `panic!`을 호출하는 것보다 `Result`를 반환하는 것이 여전히 더 적합합니다. 이에 대한 예는 기형적인 데이터가 주어지는 파서나, 속도 제한에 달했음을 나타내는 상태를 반환하는 HTTP 요청 등을 포함합니다. 이러한 경우, 여러분은 이러한 나쁜 상태를 위로 전파하기 위해 호출자가 그 문제를 어떻게 처리할지를 결정할 수 있도록 하기 위해서 `Result`를 반환하는 방식으로 실패가 예상 가능한 것임을 알려줘야 합니다. `panic!`에 빠지는 것은 이러한 경우를 처리하는 최선의 방식이 아닐 것입니다.

여러분의 코드가 어떤 값에 대해 연산을 수행할 때, 여러분의 코드는 해당 값이 유효한지를 먼저 검사하고, 만일 그렇지 않다면 **panic!**을 호출해야 합니다. 이는 주로 안전상의 이유를 위한 것입니다: 유효하지 않은 데 이터 상에서 어떤 연산을 시도하는 것은 여러분의 코드를 취약점에 노출시킬 수 있습니다. 이는 여러분이 범위를 벗어난 메모리 접근을 시도했을 경우 표준 라이브러리가 **panic!**을 호출하는 주된 이유입니다: 현재의 데이터 구조가 소유하지 않은 메모리를 접근 시도하는 것은 흔한 보안 문제입니다. 함수는 종종 계약을 갖고 있습니다: 입력이 특정 요구사항을 만족시킬 경우에만 함수의 행동이 보장됩니다. 이 계약을 위반했을 때 패닉에 빠지는 것은 사리에 맞는데, 그 이유는 계약 위반이 언제나 호출자 쪽의 버그임을 나타내고, 이는 호출하는 코드가 명시적으로 처리하도록 하는 종류의 버그가 아니기 때문입니다. 사실, 호출하는 쪽의 코드가 복구 시킬 합리적인 방법은 없습니다: 호출하는 프로그래머는 그 코드를 고칠 필요가 있습니다. 함수에 대한 계약은, 특히 계약 위반이 패닉의 원인이 될 때는, 그 함수에 대한 API 문서에 설명되어야 합니다.

하지만 여러분의 모든 함수 내에서 수많은 에러 검사를 한다는 것은 장황하고 짜증 날 것입니다. 다행스럽게도, 러스트의 타입 시스템이 (그리고 컴파일러가 하는 타입 검사 기능이) 여러분을 위해 수많은 검사를 해줄 수 있습니다. 여러분의 함수가 특정한 타입을 파라미터로 갖고 있다면, 여러분이 유효한 값을 갖는다는 것을 컴파일러가 이미 보장했음을 아는 상태로 여러분의 코드 로직을 진행할 수 있습니다. 예를 들면, 만약 여러분이 **Option**이 아닌 어떤 타입을 갖고 있다면, 여러분의 프로그램은 아무것도 아닌 것이 아닌 무언가를 갖고 있음을 예측합니다. 그러면 여러분의 코드는 **Some**과 **None** variant에 대한 두 경우를 처리하지 않아도 됩니다: 이는 분명히 값을 가지고 있는 하나의 경우만 있을 것입니다. 여러분의 함수에 아무것도 넘기지 않는 시도를 하는 코드는 컴파일조차 되지 않을 것이고, 따라서 여러분의 함수는 그러한 경우에 대해서 런타임에 검사하지 않아도 됩니다. 또 다른 예로는 **u32**와 같은 부호 없는 정수를 이용하는 것이 있는데, 이는 파라미터가 절대 음수가 아님을 보장합니다.

유효성을 위한 커스텀 타입 생성하기

러스트의 타입 시스템을 이용하여 유효한 값을 보장하는 아이디어에서 한 발 더 나가서, 유효성을 위한 커스텀 타입을 생성하는 것을 살펴봅시다. 2장의 추리 게임을 상기해 보시면, 우리의 코드는 사용자에게 1부터 100 사이의 숫자를 추측하도록 요청했었죠. 우리는 실제로는 사용자의 추측 값이 우리의 비밀 숫자와 비교하기 전에 해당 값이 유효한지 결코 확인하지 않았습니다; 우리는 추측값이 양수인지 만을 확인했습니다. 이 경우, 결과는 매우 끔찍하지는 않았습니다: “Too high”나 “Too low”라고 표시했던 출력은 여전히 맞을 것입니다. 사용자에게 유효한 추측 값을 안내해주고, 사용자가 예를 들어 글자를 입력했을 때에 비해 사용자가 범위 밖의 값을 추측했을 때 다른 동작을 하는 것은 쓸모 있는 향상일 것입니다.

이를 위한 한 가지 방법은 **u32** 대신 **i32**로서 추측 값을 파싱하여 음수가 입력될 가능성을 허용하고, 그리고나서 아래와 같이 숫자가 범위 내에 있는지에 대한 검사를 추가하는 것입니다:

```
loop {
    // snip

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will be between 1 and 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
        // snip
    }
}
```

`if` 표현식은 우리의 값이 범위 밖에 있는지 혹은 그렇지 않은지 검사하고, 사용자에게 문제점을 말해주고, `continue`를 호출하여 루프의 다음 반복을 시작하고 다른 추측값을 요청해줍니다. `if` 표현식 이후에는, `guess`가 1과 100 사이의 값이라는 것을 아는 상태에서 `guess`와 비밀 숫자의 비교를 진행할 수 있습니다.

하지만, 이는 이상적인 해결책이 아닙니다: 만일 프로그램이 오직 1과 100 사이의 값에서만 동작하는 것이 전적으로 중요하고, 많은 함수가 이러한 요구사항을 가지고 있다면, 모든 함수 내에서 이렇게 검사를 하는 것은 지루할 것입니다. (그리고 잠재적으로 성능에 영향을 줄 것입니다.)

대신, 우리는 새로운 타입을 만들어서, 유효성 확인을 모든 곳에서 반복하는 것보다는 차라리 그 타입의 인스턴스를 생성하는 함수 내에 유효성 확인을 넣을 수 있습니다. 이 방식에서, 함수가 그 시그니처 내에서 새로운 타입을 이용하고 받은 값을 자신 있게 사용하는 것은 안전합니다. Listing 9-9는 `new` 함수가 1과 100 사이의 값을 받았을 때에만 인스턴스를 생성하는 `Guess` 타입을 정의하는 한 가지 방법을 보여줍니다:

```

pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }

    pub fn value(&self) -> u32 {
        self.value
    }
}

```

Listing 9-9: 1과 100 사이의 값일 때만 계속되는 `Guess` 타입

먼저 `u32`를 갖는 `value`라는 이름의 항목을 가진 `Guess`라는 이름의 구조체를 선언하였습니다. 이것이 숫자가 저장될 곳입니다.

그런 뒤 `Guess` 값의 인스턴스를 생성하는 `new`라는 이름의 연관 함수를 구현하였습니다. `new` 함수는 `u32` 타입의 값인 `value`를 파라미터를 갖고 `Guess`를 반환하도록 정의 되었습니다. `new` 함수의 본체에 있는 코드는 `value`가 1부터 100 사이의 값인지 확인하는 테스트를 합니다. 만일 `value`가 이 테스트에 통과하지 못하면 `panic!`을 호출하며, 이는 이 코드를 호출하는 프로그래머에게 고쳐야 할 버그가 있음을 알려주는데, 범위 밖의 `value`를 가지고 `Guess`를 생성하는 것은 `Guess::new`가 필요로 하는 계약을 위반하기 때문입니다. `Guess::new`가 패닉을 일으킬 수도 있는 조건은 공개된 API 문서 내에 다뤄져야 합니다; 여러분이 만드는 API 문서 내에서 `panic!`의 가능성을 가리키는 것에 대한 문서 관례는 14장에서 다룰 것입니다. 만일 `value`가 테스트를 통과한다면, `value` 항목을 `value` 파라미터로 설정한 새로운 `Guess`를 생성하여 이 `Guess`를 반환합니다.

다음으로, `self`를 빌리고, 파라미터를 갖지 않으며, `u32`를 반환하는 `value`라는 이름의 메소드를 구현했습니다. 이러한 종류 메소드를 종종 *게터(getter)*라고 부르는데, 그 이유는 이런 함수의 목적이 객체의 항목으로부터 어떤 데이터를 가져와서 이를 반환하는 것이기 때문입니다. 이 공개 메소드는 `Guess` 구조체의 `value` 항목이 비공개이기 때문에 필요합니다. `value` 항목이 비공개라서 `Guess` 구조체를 이용하는 코드가 `value`를 직접 설정하지 못하도록 하는 것은 중요합니다: 모듈 밖의 코드는 반드시 `Guess::new` 함수를 이용하여 새로운 `Guess`의 인스턴스를 만들어야 하는데, 이는 `Guess`가 `Guess::new` 함수의 조건들을 확인한 적이 없는 `value`를 갖는 방법이 없음을 보장합니다.

그러면 파라미터를 가지고 있거나 오직 1에서 100 사이의 숫자를 반환하는 함수는 `u32` 보다는 `Guess`를 얻거나 반환하는 시그니처로 선언되고 더 이상의 확인이 필요치 않을 것입니다.

정리

러스트의 에러 처리 기능은 여러분이 더 강건한 코드를 작성하는 데 도움을 주도록 설계되었습니다.

`panic!` 매크로는 여러분의 프로그램이 처리 불가능한 상태에 놓여 있음에 대한 신호를 주고 여러분이 유효하지 않거나 잘못된 값으로 계속 진행하는 시도를 하는 대신 실행을 멈추게끔 해줍니다. `Result` 열거형은 러스트의 타입 시스템을 이용하여 여러분의 코드가 복구할 수 있는 방법으로 연산이 실패할 수도 있음을 알려줍니다. 또한 `Result`를 이용하면 여러분의 코드를 호출하는 코드에게 잠재적인 성공이나 실패를 처리해야 할 필요가 있음을 알려줄 수 있습니다. `panic!` 과 `Result`를 적합한 상황에서 사용하는 것은 여러분의 코드가 불가피한 문제에 직면했을 때도 더 신뢰할 수 있도록 해줄 것입니다.

이제 표준 라이브러리가 `Option`과 `Result` 열거형을 가지고 제네릭을 사용하는 유용한 방식들을 보았으니, 제네릭이 어떤 식으로 동작하고 여러분의 코드에 어떻게 이용할 수 있는지에 대해 다음 장에서 이야기해보겠습니다.

제네릭 타입, 트레이트, 그리고 라이프타임

모든 프로그래밍 언어는 컨셉의 복제를 효율적으로 다루기 위한 도구를 가지고 있습니다; 러스트에서, 그러한 도구 중 하나가 바로 제네릭(generic)입니다. 제네릭은 구체화된 타입이나 다른 속성들에 대하여 추상화된 대리인입니다. 코드를 작성하고 컴파일할 때, 우리는 제네릭들이 실제로 어떻게 완성되는지 알 필요 없이, 제네릭의 동작 혹은 다른 제네릭과 어떻게 연관되는지와 같은 제네릭에 대한 속성을 표현할 수 있습니다.

여러 개의 구체화된 값들에 대해 실행될 코드를 작성하기 위해서 함수가 어떤 값을 담을지 알 수 없는 파라미터를 갖는 것과 동일한 방식으로, `i32`나 `String`과 같은 구체화된 타입 대신 몇몇 제네릭 타입의 파라미터를 갖는 함수를 작성할 수 있습니다. 우리는 6장의 `Option<T>`, 8장의 `Vec<T>`와 `HashMap<K, V>`, 그리고 9장의 `Result<T, E>`에서 이미 제네릭을 사용해 보았습니다. 이 장에서는, 어떤 식으로 우리만의 타입, 함수, 그리고 메소드를 제네릭으로 정의하는지 탐험해 볼 것입니다!

우선, 우리는 코드 중복을 제거하는 함수의 추출하는 원리에 대해 돌아볼 것입니다. 그리고 나서 두 함수가 오직 파라미터의 타입만 다른 경우에 대하여 이들을 하나의 제네릭 함수로 만들기 위해 동일한 원리를 사용할 것입니다. 또한 제네릭 타입을 구조체와 열거형의 정의에 사용하는 것을 살펴볼 것입니다.

그리고 난 후 트레이트(trait)에 대하여 논의할 것인데, 이는 동작을 제네릭 한 방식으로 정의하는 방법을 말합니다. 트레이트는 제네릭 타입과 결합되어 제네릭 타입에 대해 아무 타입이나 허용하지 않고, 특정 동작을 하는 타입으로 제한할 수 있습니다.

마지막으로, 우리는 라이프타임(lifetime)에 대해 다룰 것인데, 이는 제네릭의 일종으로서 우리가 컴파일러에게 참조자들이 서로에게 어떤 연관이 있는지에 대한 정보를 줄 수 있도록 해줍니다. 라이프타임은 수많은 상황에서 값을 빌릴 수 있도록 허용해 주고도 여전히 참조자들이 유효할지를 컴파일러가 검증하도록 해주는 러스트의 지능입니다.

함수를 추출하여 중복 없애기

제네릭 문법을 들어가기 전에, 먼저 제네릭 타입을 이용하지 않는 중복 코드 다루기 기술을 훑어봅시다: 바로 함수 추출하기죠. 이를 한번 우리 마음속에서 생생하게 상기시키고 나면, 우리는 제네릭 함수를 추출하기 위해 제네릭을 가지고 똑같은 수법을 이용할 것입니다! 여러분이 함수로 추출할 중복된 코드를 인식하는 것과 똑같은 방식으로, 여러분은 제네릭을 이용할 수 있는 중복된 코드를 인식하기 시작할 것입니다.

Listing 10-1과 같이 리스트에서 가장 큰 숫자를 찾아내는 작은 프로그램이 있다고 칩니다:

Filename: src/main.rs

```
fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let mut largest = numbers[0];

    for number in numbers {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

Listing 10-1: 숫자 리스트 중에서 가장 큰 수를 찾는 코드

이 코드는 정수의 리스트를 얻는데, 여기서는 변수 `numbers`에 저장되어 있습니다. 리스트의 첫 번째 아이템을 `largest`라는 이름의 변수에 우선 집어넣습니다. 그리고 나서 리스트 내의 모든 숫자들에 대해 반복 접근을 하는데, 만일 현재 숫자가 `largest` 내에 저장된 숫자보다 더 크다면, 이 숫자로 `largest` 내의 값을 변경합니다. 만일 현재 숫자가 여태까지 본 가장 큰 값보다 작다면, `largest`는 바뀌지 않습니다. 리스트 내의 모든 아이템을 다 처리했을 때, `largest`는 가장 큰 값을 가지고 있을 것인데, 위 코드의 경우에는 100이 될 것입니다.

만일 두 개의 서로 다른 숫자 리스트로부터 가장 큰 숫자를 찾기를 원한다면, Listing 10-1의 코드를 복사하여, Listing 10-2에서처럼 한 프로그램 내에 동일한 로직이 두 군데 있게 할 수도 있습니다:

Filename: src/main.rs

```

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let mut largest = numbers[0];

    for number in numbers {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let numbers = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = numbers[0];

    for number in numbers {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}

```

Listing 10-2: 두 개의 숫자 리스트에서 가장 큰 숫자를 찾는 코드

이 코드는 잘 동작하지만, 코드를 중복 적용하는 일은 지루하고 오류가 발생하기도 쉬우며, 또한 로직을 바꾸고 싶다면 이 로직을 갱신할 곳이 여러 군데가 된다는 의미이기도 합니다.

이러한 중복을 제거하기 위해서 우리는 추상화를 쓸 수 있는데, 이 경우에는 어떠한 정수 리스트가 함수의 파라미터로 주어졌을 때 동작하는 함수의 형태가 될 것입니다. 이는 우리 코드의 명료성을 증가시켜주고 리스트 내에서 가장 큰 수를 찾는 컨셉을 사용하는 특정한 위치와 상관없이 이러한 컨셉을 전달하고 추론하도록 해줍니다.

Listing 10-3의 프로그램에서는 가장 큰 수를 찾는 코드를 `largest`라는 이름의 함수로 추출했습니다. 이 프로그램은 두 개의 서로 다른 숫자 리스트에서 가장 큰 수를 찾을 수 있지만, Listing 10-1에서의 코드는 한 군데에서만 나타납니다:

Filename: src/main.rs

```

fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let numbers = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&numbers);
    println!("The largest number is {}", result);
}

```

Listing 10-3: 두 리스트에서 가장 큰 수를 찾는 추상화된 코드

이 함수는 `list`라는 파라미터를 갖고 있는데, 이것이 함수로 넘겨질 구체적인 임의의 `i32` 값들의 슬라이스를 나타냅니다. 함수 정의 내의 코드는 임의의 `&[i32]`의 `list` 표현에 대해 동작합니다. `largest` 함수를 호출할 때, 이 코드는 실제로 우리가 넘겨준 구체적인 값에 대해 실행됩니다.

Listing 10-2에서부터 Listing 10-3까지 우리가 살펴본 원리는 아래와 같은 단계로 진행되었습니다:

1. 중복된 코드가 있음을 알아했습니다.
2. 중복된 코드를 함수의 본체로 추출하고, 함수의 시그니처 내에 해당 코드의 입력값 및 반환 값을 명시했습니다.
3. 두 군데의 코드가 중복되었던 구체적인 지점에 함수 호출을 대신 집어넣었습니다.

우리는 다른 시나리오 상에서 다른 방식으로 제네릭을 가지고 중복된 코드를 제거하기 위해 같은 단계를 밟을 수 있습니다. 함수의 본체가 현재 구체적인 값 대신 추상화된 `list`에 대해 동작하고 있는 것과 같이, 제네릭을 이용한 코드는 추상화된 타입에 대해 작동할 것입니다. 제네릭으로 강화되는 컨셉은 여러분이 이미 알고 있는 함수로 강화되는 컨셉과 동일하며, 다만 다른 방식으로 적용될 뿐입니다.

만일 우리가 두 개의 함수를 가지고 있는데, 하나는 `i32`의 슬라이스에서 최댓값을 찾는 것이고 다른 하나는 `char` 값의 슬라이스에서 최댓값을 찾는 것이라면 어떨까요? 어떻게 하면 이런 중복을 제거할 수 있을까요? 한번 알아봅시다!

제네릭 데이터 타입

함수 시그니처나 구조체에서와 같은 방식으로, 우리가 일반적으로 타입을 쓰는 곳에다 제네릭을 이용하는 것은 여러 다른 종류의 구체적인 데이터 타입에 대해 사용할 수 있는 정의를 생성하도록 해줍니다. 제네릭을 이용하여 함수, 구조체, 열거형, 그리고 메소드를 정의하는 방법을 살펴본 뒤, 이 절의 끝에서 제네릭을 이용한 코드의 성능에 대해 논의하겠습니다.

함수 정의 내에서 제네릭 데이터 타입을 이용하기

우리는 함수의 시그니처 내에서 파라미터의 데이터 타입과 반환 값이 올 자리에 제네릭을 사용하는 함수를 정의할 수 있습니다. 이러한 방식으로 작성된 코드는 더 유연해지고 우리 함수를 호출하는 쪽에서 더 많은 기능을 제공할 수 있는 한편, 코드 중복을 야기하지도 않습니다.

우리의 `largest` 함수로 계속 진행하면, Listing 10-4는 슬라이스 내에서 가장 큰 값을 찾는 동일한 기능을 제공하는 두 함수를 보여주고 있습니다. 첫 번째 함수는 Listing 10-3에서 추출한 슬라이스에서 가장 큰 `i32`를 찾는 함수입니다. 두 번째 함수는 슬라이스에서 가장 큰 `char`를 찾습니다:

Filename: src/main.rs

```

fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&chars);
    println!("The largest char is {}", result);
}

```

Listing 10-4: 이름과 시그니처만 다른 두 함수들

여기서 함수 `largest_i32` 와 `largest_char` 는 정확히 똑같은 본체를 가지고 있으므로, 만일 우리가 이 두 함수를 하나로 바꿔서 중복을 제거할 수 있다면 좋을 것입니다. 운 좋게도, 제네릭 타입 파라미터를 도입해서 그렇게 할 수 있습니다!

우리가 정의하고자 하는 함수의 시그니처 내에 있는 타입들을 파라미터화 하기 위해서, 타입 파라미터를 위한 이름을 만들 필요가 있는데, 이는 값 파라미터들의 이름을 함수에 제공하는 방법과 유사합니다. 우리는 `T` 라는 이름을 선택할 겁니다. 어떤 식별자(identifier)든지 타입 파라미터의 이름으로 사용될 수 있지만, 러스트의 타입 이름에 대한 관례가 낙타 표기법(CamelCase)이기 때문에 `T`를 사용하려고 합니다. 제네릭 타입 파라미터의 이름은 또한 관례상 짧은 경향이 있는데, 종종 그냥 한 글자로 되어 있습니다. "type"을 줄인 것으

로서, `T` 가 대부분의 러스트 프로그래머의 기본 선택입니다.

함수의 본체에 파라미터를 이용할 때는, 시그니처 내에 그 파라미터를 선언하여 해당 이름이 함수 본체 내에서 무엇을 의미하는지 컴파일러가 할 수 있도록 합니다. 비슷하게, 함수 시그니처 내에서 타입 파라미터 이름을 사용할 때는, 사용 전에 그 타입 파라미터 이름을 선언해야 합니다. 타입 이름 선언은 함수의 이름과 파라미터 리스트 사이에 꺼쇠괄호를 쓰고 그 안에 넣습니다.

우리가 정의하고자 하는 제네릭 `largest` 함수의 함수 시그니처는 아래와 같이 생겼습니다:

```
fn largest<T>(list: &[T]) -> T {
```

이를 다음과 같이 읽습니다: 함수 `largest`는 어떤 타입 `T`을 이용한 제네릭입니다. 이것은 `list`라는 이름을 가진 하나의 파라미터를 가지고 있고, `list`의 타입은 `T` 타입 값들의 슬라이스입니다. `largest` 함수는 동일한 타입 `T` 값을 반환할 것입니다.

Listing 10-5는 함수 시그니처 내에 제네릭 데이터 타입을 이용한 통합된 형태의 `largest` 함수 정의를 보여주며, 또한 `i32` 값들의 슬라이스 혹은 `char` 값들의 슬라이스를 가지고 어떻게 `largest`를 호출할 수 있을지를 보여줍니다. 이 코드가 아직 컴파일되지 않는다는 점을 주의하세요!

Filename: src/main.rs

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest(&chars);
    println!("The largest char is {}", result);
}
```

Listing 10-5: 제네릭 타입 파라미터를 이용하지만 아직 컴파일되지 않는 `largest` 함수의 정의

이 코드를 지금 컴파일하고자 시도하면, 다음과 같은 에러를 얻게 될 것입니다:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
5 |         if item > largest {
|         ^^^^
|
note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

위 노트는 `std::cmp::PartialOrd`를 언급하는데, 이는 트레이트(trait)입니다. 트레이트에 대해서는 다음 절에서 살펴볼 것이지만, 간략하게 설명하자면, 이 에러가 말하고 있는 것은 `T`가 될 수 있는 모든 가능한 타입에 대해서 동작하지 않으리라는 것입니다: 함수 본체 내에서 `T` 타입의 값을 비교하고자 하기 때문에, 어떻게 순서대로 정렬하는지 알고 있는 타입만 사용할 수 있는 것입니다. 표준 라이브러리는 어떤 타입에 대해 비교 연산이 가능하도록 구현할 수 있는 트레이트인 `std::cmp::PartialOrd`를 정의해뒀습니다. 다음 절에서 트레이트, 그리고 어떤 제네릭 타입이 특정 트레이트를 갖도록 명시하는 방법을 알아보기 위해 돌아올 것이지만, 이 예제는 잠시 옆으로 치워두고 제네릭 타입 파라미터를 이용할 수 있는 다른 곳을 먼저 돌아봅시다.

구조체 정의 내에서 제네릭 데이터 타입 사용하기

우리는 또한 하나 혹은 그 이상의 구조체 필드 내에 제네릭 타입 파라미터를 사용하여 구조체를 정의할 수 있습니다. Listing 10-6은 임의의 타입으로 된 `x`와 `y` 좌표값을 가질 수 있는 `Point` 구조체의 정의 및 사용법을 보여주고 있습니다:

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Listing 10-6: `T` 타입의 값 `x`와 `y`를 갖는 `Point` 구조체

문법은 함수 정의 내에서의 제네릭을 사용하는 것과 유사합니다. 먼저, 구조체 이름 바로 뒤에 꺽쇠괄호를 쓰고 그 안에 타입 파라미터의 이름을 선언해야 합니다. 그러면 구조체 정의부 내에서 구체적인 데이터 타입을 명시하는 곳에 제네릭 타입을 이용할 수 있습니다.

`Point`의 정의 내에서 단 하나의 제네릭 타입을 사용했기 때문에, `Point` 구조체는 어떤 타입 `T`를 이용한 제네릭이고 `x`와 `y`가 이게 결국 무엇이 되든 간에 둘 다 동일한 타입을 가지고 있다고 말할 수 있음을 주목하세요. 만일 Listing 10-7에서와 같이 다른 타입의 값을 갖는 `Point`의 인스턴스를 만들고자 한다면, 컴파일이 되지 않을 것입니다:

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

Listing 10-7: `x`와 `y` 필드는 둘 모두 동일한 제네릭 데이터 타입 `T`를 가지고 있기 때문에 동일한 타입이어야 합니다

이 코드를 컴파일하고자 하면, 다음과 같은 에러를 얻게 될 것입니다:

```
error[E0308]: mismatched types
-->
|
7 |     let wont_work = Point { x: 5, y: 4.0 };
|                         ^^^ expected integral variable,
found
floating-point variable
|
= note: expected type `'{integer}`
= note:     found type `'{float}'
```

`x`에 정수 5를 대입할 때, 컴파일러는 이 `Point`의 인스턴스에 대해 제네릭 타입 `T`가 정수일 것이고 알게 됩니다. 그다음 `y`에 대해 4.0을 지정했는데, 이 `y`는 `x`와 동일한 타입을 갖도록 정의되었으므로, 타입 불일치 에러를 얻게 됩니다.

만일 `x`와 `y`가 서로 다른 타입을 가지지만 해당 타입들이 여전히 제네릭인 `Point` 구조체를 정의하길 원한다면, 여러 개의 제네릭 타입 파라미터를 이용할 수 있습니다. Listing 10-8에서는 `Point`의 정의를 `T`와 `U`를 이용한 제네릭이 되도록 변경했습니다. 필드 `x`의 타입은 `T`이고, 필드 `y`의 타입은 `U`입니다:

Filename: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Listing 10-8: 두 타입을 이용한 제네릭이어서 `x`와 `y`가 다른 타입의 값일 수도 있는 `Point`

이제 위와 같은 모든 `Point` 인스턴스가 허용됩니다! 정의 부분에 여러분이 원하는 만큼 많은 수의 제네릭 타입 파라미터를 이용할 수 있지만, 몇몇 개보다 더 많이 이용하는 것은 읽고 이해하는 것을 어렵게 만듭니다. 여러분이 많은 수의 제네릭 타입을 필요로 하는 지점에 다다랐다면, 이는 아마도 여러분의 코드가 좀 더 작은 조각들로 나뉘는 재구조화가 필요할지도 모른다는 징조입니다.

열거형 정의 내에서 제네릭 데이터 타입 사용하기

구조체와 유사하게, 열거형도 그 variant 내에서 제네릭 데이터 타입을 갖도록 정의될 수 있습니다. 6장에서 표준 라이브러리가 제공하는 `Option<T>` 열거형을 이용해봤는데, 이제는 그 정의를 좀 더 잘 이해할 수 있겠지요. 다시 한번 봅시다:

```
enum Option<T> {
    Some(T),
    None,
}
```

달리 말하면, `Option<T>`는 `T` 타입에 제네릭인 열거형입니다. 이것은 두 개의 variant를 가지고 있습니다: 타입 `T` 값 하나를 들고 있는 `Some`, 그리고 어떠한 값도 들고 있지 않는 `None` variant입니다. 표준 라이브러리는 구체적인 타입을 가진 이 열거형에 대한 값의 생성을 지원하기 위해서 딱 이 한 가지 정의만 가지고 있으면 됩니다. "옵션 값"의 아이디어는 하나의 명시적인 타입에 비해 더 추상화된 개념이고, 러스트는 이 추상화 개념을 수많은 중복 없이 표현할 수 있도록 해줍니다.

열거형은 또한 여러 개의 제네릭 타입을 이용할 수 있습니다. 우리가 9장에서 사용해본 `Result` 열거형의 정의가 한 가지 예입니다:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Result 열거형은 **T** 와 **E**, 두 개의 타입을 이용한 제네릭입니다. **Result**는 두 개의 variant를 가지고 있습니다: 타입 **T**의 값을 들고 있는 **Ok**, 그리고 타입 **E**의 값을 들고 있는 **Err**입니다. 이 정의는 성공하거나 (그래서 어떤 **T** 값을 반환하거나) 혹은 실패하는 (그래서 **E** 타입으로 된 에러를 반환하는) 연산이 필요한 어디에서든 편리하게 **Result** 열거형을 이용하도록 해줍니다. Listing 9-2에 우리가 파일을 열 때를 상기해보세요: 이 경우, 파일이 성공적으로 열렸을 때는 **T**에 **std::fs::File** 타입의 값이 채워지고 파일을 여는데 문제가 생겼을 때는 **E**에 **std::io::Error** 타입으로 된 값이 채워졌습니다.

여러분의 코드에서 단지 들고 있는 값의 타입만 다른 여러 개의 구조체나 열거형이 있는 상황을 인지했다면, 우리가 함수 정의에서 제네릭 타입을 대신 도입하여 사용했던 것과 똑같은 절차를 통해 그러한 중복을 제거할 수 있습니다.

메소드 정의 내에서 제네릭 데이터 타입 사용하기

5장에서 했던 것과 유사하게, 정의부에 제네릭 타입을 갖는 구조체와 열거형 상의 메소드를 구현할 수도 있습니다. Listing 10-9는 우리가 Listing 10-6에서 정의했던 **Point<T>** 구조체를 보여주고 있습니다. 그리고 나서 필드 **x**의 값에 대한 참조자를 반환하는 **x**라는 이름의 메소드를 **Point<T>** 상에 정의했습니다:

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Listing 10-9: **T** 타입의 **x** 필드에 대한 참조자를 반환하는 **Point<T>** 구조체 상에 **x**라는 이름의 메소

드 정의

`impl` 바로 뒤에 `T`를 정의해야만 타입 `Point<T>` 메소드를 구현하는 중에 이를 사용할 수 있음을 주목하세요.

구조체 정의 내에서의 제네릭 타입 파라미터는 여러분이 구조체의 메소드 시그니처 내에서 사용하고 싶어하는 제네릭 타입 파라미터와 항상 같지 않습니다. Listing 10-10에서는 Listing 10-8에서의 `Point<T, U>` 구조체 상에 `mixup`이라는 메소드를 정의했습니다. 이 메소드는 또 다른 `Point`를 파라미터로 갖는데, 이는 우리가 호출하는 `mixup` 상의 `self`의 `Point`와 다른 타입을 가지고 있을 수도 있습니다. 이 메소드는 새로운 `Point`를 생성하는데 `self` `Point`로부터 (`T` 타입인) `x` 값을 가져오고, 파라미터로 넘겨받은 `Point`로부터 (`W` 타입인) `y` 값을 가져온 것입니다:

Filename: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Listing 10-10: 구조체 정의에서와는 다른 제네릭 타입을 사용하는 메소드

`main`에서, 우리는 (`5` 값을 갖는) `x`에 대해 `i32`를, (`10.4` 값을 갖는) `y`에 대해 `f64`를 사용하는 `Point`를 정의했습니다. `p2`는 (`"Hello"` 값을 갖는) `x`에 대해 스트링 슬라이스를, (`c` 값을 갖는) `y`에 대해 `char`를 사용하는 `Point`입니다. `p1`상에서 인자로 `p2`를 넘기는 `mixup` 호출은 `p3`을 반환하는데, 이는 `x`가 `p1`으로부터 오기 때문에 `x`는 `i32` 타입을 갖게 될 것입니다. 또한 `y`는 `p2`로부터 오기 때문에 `p3`은 `y`에 대해 `char` 타입을 가지게 될 것입니다. `println!`은 `p3.x = 5, p3.y = c`를 출력하겠지요.

제네릭 파라미터 `T`와 `U`는 `impl` 뒤에 선언되었는데, 이는 구조체 정의와 함께 사용되기 때문임을 주목하세요. 제네릭 파라미터 `V`와 `W`는 `fn mixup` 뒤에 선언되었는데, 이는 이들이 오직 해당 메소드에 대해서만 관련이 있기 때문입니다.

제네릭을 이용한 코드의 성능

여러분이 이 절을 읽으면서 제네릭 타입 파라미터를 이용한 런타임 비용이 있는지 궁금해하고 있을지도 모르겠습니다. 좋은 소식을 알려드리죠: 러스트가 제네릭을 구현한 방식이 의미하는 바는 여러분이 제네릭 파라미터 대신 구체적인 타입을 명시했을 때와 비교해 전혀 느려지지 않을 것이란 점입니다!

러스트는 컴파일 타임에 제네릭을 사용하는 코드에 대해 단형성화(*monomorphization*)를 수행함으로써 이러한 성능을 이루어 냈습니다. 단형성화란 제네릭 코드를 실제로 채워질 구체적인 타입으로 된 특정 코드로 바꾸는 과정을 말합니다.

컴파일러가 하는 일은 Listing 10-5에서 우리가 제네릭 함수를 만들 때 수행한 단계들을 반대로 한 것입니다. 컴파일러는 제네릭 코드가 호출되는 모든 곳을 살펴보고 제네릭 코드가 호출될 때 사용된 구체적인 타입에 대한 코드를 생성합니다.

표준 라이브러리의 `Option` 열거형을 사용하는 예제를 통해 알아봅시다:

```
let integer = Some(5);
let float = Some(5.0);
```

러스트가 이 코드를 컴파일할 때, 단형성화를 수행할 것입니다. 컴파일러는 `Option`에 넘겨진 값들을 읽고 두 종류의 `Option<T>`를 가지고 있다는 사실을 알게 됩니다: 하나는 `i32`이고 나머지 하나는 `f64` 이지요. 그리하여 컴파일러는 제네릭 정의를 명시적인 것들로 교체함으로써 `Option<T>`에 대한 제네릭 정의를 `Option_i32`와 `Option_f64`로 확장시킬 것입니다.

컴파일러가 생성한 우리의 단형성화된 버전의 코드는 아래와 같이 보이게 되는데, 컴파일러에 의해 생성된 구체화된 정의로 교체된 제네릭 `Option`이 사용되었습니다:

Filename: src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

우리는 제네릭을 사용하여 중복 없는 코드를 작성할 수 있고, 러스트는 이를 각 인스턴스에 대해 구체적인 타입을 갖는 코드로 컴파일할 것입니다. 이는 우리가 제네릭을 사용하는 데에 어떠한 런타임 비용도 없음을 의미합니다; 코드가 실행될 때, 손으로 각각 특정 정의를 중복시킨 것과 같이 실행될 것입니다. 단형성화의 과정은 러스트의 제네릭이 런타임에 극도로 효율적 이도록 만들어 주는 것입니다.

트레이트: 공유 동작을 정의하기

트레이트는 다른 종류의 추상화를 사용할 수 있도록 해줍니다: 이는 타입들이 공통적으로 갖는 동작에 대하여 추상화하도록 해줍니다. 트레이트(trait) 이란 러스트 컴파일러에게 특정한 타입이 갖고 다른 타입들과 함께 공유할 수도 있는 기능에 대해 말해줍니다. 우리가 제네릭 타입 파라미터를 사용하는 상황에서는, 컴파일 타임에 해당 제네릭 타입이 어떤 트레이트를 구현한 타입이어야 함을 명시하여, 그러한 상황에서 우리가 사용하기 원하는 동작을 갖도록 하기 위해 트레이트 바운드(trait bounds) 를 사용할 수 있습니다.

노트: 트레이트는 다른 언어들에서 '인터페이스(interface)'라고 부르는 기능과 유사하지만, 몇 가지 다른 점이 있습니다.

트레이트 정의하기

어떤 타입의 동작은 우리가 해당 타입 상에서 호출할 수 있는 메소드들로 구성되어 있습니다. 만일 우리가 서로 다른 타입에 대해 모두 동일한 메소드를 호출할 수 있다면 이 타입들은 동일한 동작을 공유하는 것입니다. 트레이트의 정의는 어떠한 목적을 달성하기 위해 필요한 동작의 집합을 정의하기 위해 메소드 시그니처들을 함께 묶는 방법입니다.

예를 들면, 다양한 종류와 양의 텍스트를 갖는 여러 가지의 구조체를 가지고 있다고 칩니다: `NewsArticle` 구조체는 세계의 특정한 곳에서 줄지어 들어오는 뉴스 이야기를 들고 있고, `Tweet` 은 최대 140글자의 콘텐츠와 함께 해당 트윗이 리트윗인지 혹은 다른 트윗에 대한 답변인지와 같은 메타데이터를 가지고 있습니다.

우리는 `NewsArticle` 혹은 `Tweet` 인스턴스에 저장되어 있을 데이터에 대한 종합 정리를 보여줄 수 있는 미디어 종합기 라이브러리를 만들고 싶어 합니다. 각각의 구조체들이 가질 필요가 있는 동작은 정리해주기가 되어야 하며, 그래서 각 인스턴스 상에서 `summary` 메소드를 호출함으로써 해당 정리를 얻어낼 수 있어야 한다는 것입니다. Listing 10-11은 이러한 개념을 표현한 `Summarizable` 트레이트의 정의를 나타냅니다:

Filename: lib.rs

```
pub trait Summarizable {
    fn summary(&self) -> String;
}
```

Listing 10-11: `summary` 메소드에 의해 제공되는 동작으로 구성된 `Summarizable` 트레이트의 정의

`trait` 키워드 다음 트레이트의 이름, 위의 경우 `Summarizable` 을 써서 트레이트를 선언했습니다. 중괄호 내에서는 이 트레이트를 구현하는 타입들이 가질 필요가 있는 동작들을 묘사한 메소드 시그니처들을 정의했는데, 위의 경우에는 `fn summary(&self) -> String`입니다. 메소드 시그니처 뒤에, 중괄호 내의 정의부를

제공하는 대신, 세미콜론을 집어넣었습니다. 그러면 이 트레이잇을 구현하는 각 타입은 이 메소드의 본체에 대한 해당 타입 고유의 커스텀 동작을 제공해야 하는데, 컴파일러는 **Summarizable** 트레이잇을 갖는 어떠한 타입이든 그에 대한 메소드 **summary**를 정확히 동일한 시그니처로 정의되도록 강제할 것입니다.

트레이잇은 한 줄 당 하나의 메소드 시그니처와 각 줄의 끝에 세미콜론을 갖도록 함으로써, 본체 내에 여러 개의 메소드를 가질 수 있습니다.

특정 타입에 대한 트레이잇 구현하기

Summarizable 트레이잇을 정의하였으니, 이제 우리의 미디어 종합기 내에서 이 동작을 갖길 원했던 타입들 상에 이 트레이잇을 구현할 수 있습니다. Listing 10-12는 **summary**의 반환 값을 만들기 위해 헤드라인, 저자, 위치를 사용하는 **NewsArticle** 구조체 상의 **Summarizable** 트레이잇 구현을 보여줍니다. **Tweet** 구조체에 대해서는, 트윗 내용이 이미 140자로 제한되어 있음을 가정하고, **summary**를 정의하는 데 있어 사용자 이름과 해당 트윗의 전체 텍스트를 가지고 오는 선택을 했습니다.

Filename: lib.rs

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summarizable for NewsArticle {
    fn summary(&self) -> String {
        format!("{} by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summarizable for Tweet {
    fn summary(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Listing 10-12: **NewsArticle**과 **Tweet** 타입 상에서의 **Summarizable** 트레이잇 구현

어떤 타입 상에서의 트레이트 구현은 트레이트과 관련이 없는 메소드를 구현하는 것과 유사합니다. 다른 점은 `impl` 뒤에 우리가 구현하고자 하는 트레이트 이름을 넣고, 그다음 `for`와 우리가 트레이트를 구현하고자 하는 타입의 이름을 쓴다는 것입니다. `impl` 블록 내에서는 트레이트 정의부가 정의한 바 있는 메소드 시그니처를 집어넣지만, 각 시그니처의 끝에 세미콜론을 집어넣는 대신 중괄호를 넣고 우리가 트레이트의 메소드가 특정한 타입에 대해서 갖기를 원하는 특정한 동작으로 메소드의 본체를 채웁니다.

트레이트를 한번 구현했다면, 트레이트의 일부가 아닌 메소드들을 호출했던 것과 동일한 방식으로 `NewsArticle`과 `Tweet`의 인스턴스 상에서 해당 메소드들을 호출할 수 있습니다:

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summary());
```

이 코드는 `1 new tweet: horse_ebooks: of course, as you probably already know, people`를 출력할 것입니다.

Listing 10-12에서 `Summarizable` 트레이트와 `NewsArticle` 및 `Tweet` 타입을 동일한 `lib.rs` 내에 정의했기 때문에, 이들이 모두 동일한 스코프 내에 있다는 점을 주목하세요. 만일 이 `lib.rs`가 `aggregator`라고 불리는 크레이트에 대한 것이고 누군가가 우리의 크레이트 기능에 더해 그들의 `WeatherForecast` 구조체에 대하여 `Summarizable`을 구현하기를 원한다면, 그들의 코드는 Listing 10-13과 같이 이를 구현하기 전에 먼저 `Summarizable` 트레이트를 그들의 스코프로 가져올 필요가 있습니다:

Filename: lib.rs

```

extern crate aggregator;

use aggregator::Summarizable;

struct WeatherForecast {
    high_temp: f64,
    low_temp: f64,
    chance_of_precipitation: f64,
}

impl Summarizable for WeatherForecast {
    fn summary(&self) -> String {
        format!("The high will be {}, and the low will be {}. The chance of
precipitation is {}%.", self.high_temp, self.low_temp,
self.chance_of_precipitation)
    }
}

```

Listing 10-13: 우리의 `aggregator` 크레이트로부터 다른 크레이트 내의 스코프로 `Summarizable` 트레잇을 가져오기

이 코드는 또한 `Summarizable`이 공개 트레잇임을 가정하는데, 이는 Listing 10-11에서 `trait` 전에 `pub` 키워드를 집어넣었기 때문입니다.

트레잇 구현과 함께 기억할 한 가지 제한사항이 있습니다: 트레잇 혹은 타입이 우리의 크레이트 내의 것일 경우에만 해당 타입에서의 트레잇을 정의할 수 있습니다. 바꿔 말하면, 외부의 타입에 대한 외부 트레잇을 구현하는 것은 허용되지 않습니다. 예를 들어, `Vec`에 대한 `Display` 트레잇은 구현이 불가능한데, `Display` 와 `Vec` 모두 표준 라이브러리 내에 정의되어 있기 때문입니다. 우리의 `aggregator` 크레이트 기능의 일부로서 `Tweet`과 같은 커스텀 타입에 대한 `Display`와 같은 표준 라이브러리 트레잇을 구현하는 것은 허용됩니다. 또한 우리의 `aggregator` 크레이트 내에서 `Vec`에 대한 `Summarizable`을 구현하는 것도 가능합니다, 이는 우리 크레이트 내에 `Summarizable`이 정의되어 있기 때문입니다. 이러한 제한은 고아 규칙 (*orphan rule*)이라고 불리는 것의 일부인데, 이는 타입 이론에 흥미가 있다면 찾아볼 수 있습니다. 간단하게 말하면, 부모 타입이 존재하지 않기 때문에 고아 규칙이라고 부릅니다. 이 규칙이 없다면, 두 크레이트는 동일한 타입에 대해 동일한 트레잇을 구현할 수 있게 되고, 이 두 구현체가 충돌을 일으킬 것입니다: 러스트는 어떤 구현을 이용할 것인지 알지 못할 것입니다. 러스트가 고아 규칙을 강제하기 때문에, 다른 사람의 코드는 여러분의 코드를 망가뜨리지 못하고 반대의 경우도 마찬가지입니다.

기본 구현

종종 모든 타입 상에서의 모든 구현체가 커스텀 동작을 정의하도록 하는 대신, 트레잇의 몇몇 혹은 모든 메소드들에 대한 기본 동작을 갖추는 것이 유용할 수 있습니다. 특정한 타입에 대한 트레잇을 구현할 때, 각 메소드의 기본 동작을 유지하거나 오버라이드(override)하도록 선택할 수 있습니다.

Listing 10-14는 우리가 Listing 10-11에서 한 것과 같이 메소드 시그니처를 정의만 하는 선택 대신 **Summarizable** 트레이트의 **summary** 메소드에 대한 기본 스트링을 명시하는 선택을 하는 방법을 보여줍니다:

Filename: lib.rs

```
pub trait Summarizable {
    fn summary(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Listing 10-14: **summary** 메소드의 기본 구현을 포함한 **Summarizable** 트레이트의 정의

만일 우리가 Listing 10-12에서 한 것과 같은 커스텀 구현을 정의하는 대신 **NewsArticle**의 인스턴스를 정리하기 위해 이 기본 구현을 사용하고자 한다면, 빈 **impl** 블록을 명시하면 됩니다:

```
impl Summarizable for NewsArticle {}
```

비록 **NewsArticle**에 대한 **summary** 메소드를 직접 정의하는 선택을 더 이상 하지 않았더라도, **summary** 메소드가 기본 구현을 갖고 있고 **NewsArticle**이 **Summarizable** 트레이트를 구현하도록 명시했기 때문에, 우리는 여전히 **newsArticle**의 인스턴스 상에서 **summary** 메소드를 호출할 수 있습니다:

```
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from("The Pittsburgh Penguins once again are the best
hockey team in the NHL."),
};

println!("New article available! {}", article.summary());
```

위의 코드는 **New article available! (Read more...)**를 출력합니다.

Summarizable 트레이트가 **summary**에 대한 기본 구현을 갖도록 변경하는 것은 Listing 10-12의 **Tweet**이나 Listing 10-13의 **WeatherForecast** 상에서의 **Summarizable** 구현에 대한 어떤 것도 바꾸도록 요구하지 않습니다: 기본 구현을 오버라이딩 하기 위한 문법은 기본 구현이 없는 트레이트 메소드를 구현하기 위한 문법과 정확히 동일합니다.

기본 구현은 동일한 트레이트 내의 다른 메소드들을 호출하는 것이 허용되어 있는데, 심지어 그 다른 메소드들이 기본 구현을 갖고 있지 않아도 됩니다. 이러한 방식으로, 트레이트는 수많은 유용한 기능을 제공하면서도 다른 구현자들이 해당 트레이트의 작은 일부분만 구현하도록 요구할 수 있습니다. 우리는 **Summarizable** 트레이트

잇이 구현이 필요한 `author_summary` 메소드도 갖도록 하여, `summary` 메소드가 `author_summary` 메소드를 호출하는 기본 구현을 갖는 형태를 선택할 수도 있습니다:

```
pub trait Summarizable {
    fn author_summary(&self) -> String;

    fn summary(&self) -> String {
        format!("(Read more from {}...)", self.author_summary())
    }
}
```

이 버전의 `Summarizable`을 사용하기 위해서는, 어떤 타입에 대한 이 트레이트를 구현할 때 `author_summary`만 정의하면 됩니다:

```
impl Summarizable for Tweet {
    fn author_summary(&self) -> String {
        format!("@{}", self.username)
    }
}
```

일단 `author_summary`를 정의하면, `Tweet` 구조체의 인스턴스 상에서 `summary`를 호출할 수 있으며, `summary`의 기본 구현이 우리가 제공한 `author_summary`의 정의부를 호출할 것입니다.

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summary());
```

위의 코드는 `1 new tweet: (Read more from @horse_ebooks...)`를 출력할 것입니다.

오버라이딩된 구현으로부터 기본 구현을 호출하는 것은 불가능하다는 점을 기억해주세요.

트레이트 바운드

이제 트레이트를 정의하고 어떤 타입들에 대해 이 트레이트를 구현해봤으니, 제네릭 타입 파라미터를 이용하는 트레이트를 사용할 수 있습니다. 우리는 제네릭 타입에 제약을 가하여 이 제네릭 타입이 어떠한 타입이든 되기보다는, 이 제네릭 타입이 특정한 트레이트를 구현하여 이 타입들이 가지고 있을 필요가 있는 동작을 갖고 있도록 타입들로 제한함을 컴파일러가 확신하도록 할 수 있습니다.

예를 들면, Listing 10-12에서는 `NewsArticle`과 `Tweet` 타입에 대하여 `Summarizable` 트레이트를 구현했습니다. 우리는 파라미터 `item` 상에서 `summary` 메소드를 호출하는 함수 `notify`를 정의할 수 있는데, 이 `item`은 제네릭 타입 `T`의 값입니다. 어려없이 `item` 상에서 `summary`를 호출하기 위해서는, `T`에 대한 트레이트 바운드를 사용하여 `item`이 `Summarizable` 트레이트를 반드시 구현한 타입이어야 함을 특정할 수 있습니다:

```
pub fn notify<T: Summarizable>(item: T) {
    println!("Breaking news! {}", item.summary());
}
```

트레이트 바운드는 제네릭 타입 파라미터의 선언부와 함께, 꺽쇠 괄호 내에 콜론 뒤에 옵니다. `T` 상에서의 트레이트 바운드이므로, 우리는 `notify`를 호출하여 `NewsArticle`이나 `Tweet`의 어떠한 인스턴스라도 넘길 수 있습니다. 우리의 `aggregator` 크레이트를 사용하는 Listing 10-13의 외부 코드도 우리의 `notify` 함수를 호출하여 `WeatherForecast`의 인스턴스를 넘길 수 있는데, 이는 `WeatherForecast` 또한 `Summarizable`을 구현하였기 때문입니다. `String`이나 `i32` 같은 어떠한 다른 타입을 가지고 `notify`를 호출하는 코드는 컴파일되지 않을 것인데, 그 이유는 그러한 타입들이 `Summarizable`을 구현하지 않았기 때문입니다.

`+`를 이용하면 하나의 제네릭 타입에 대해 여러 개의 트레이트 바운드를 특정할 수 있습니다. 만일 함수 내에서 타입 `T`에 대해 `summary` 메소드 뿐만 아니라 형식화된 출력력을 사용하길 원한다면, 트레이트 바운드 `T: Summarizable + Display`를 이용할 수 있습니다. 이는 `T`가 `Summarizable`과 `Display` 둘다 구현한 어떤 타입이어야 함을 의미합니다.

여러 개의 제네릭 타입 파라미터를 가진 함수들에 대하여, 각 제네릭은 고유의 트레이트 바운드를 가집니다. 함수 이름과 파라미터 리스트 사이의 꺽쇠 괄호 내에 많은 수의 트레이트 바운드 정보를 특정하는 것은 코드를 읽기 힘들게 만들 수 있으므로, 함수 시그니처 뒤에 `where` 절 뒤로 트레이트 바운드를 옮겨서 특정하도록 해주는 대안 문법이 있습니다. 따라서 아래와 같은 코드 대신:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

`where` 절을 이용하여 아래와 같이 작성할 수 있습니다:

```
fn some_function<T, U>(t: T, u: U) -> i32
where T: Display + Clone,
      U: Clone + Debug
{
```

함수 이름, 파라미터 리스트, 그리고 반환 타입이 서로 가까이 있도록 하여, 이쪽이 덜 어수선하고 이 함수의 시그니처를 많은 트레이트 바운드를 가지고 있지 않은 함수처럼 보이도록 만들어 줍니다.

트레이트 바운드를 사용하여 `largest` 함수 고치기

따라서 여러분이 어떤 제네릭 상에서 어떤 트레이트으로 정의된 동작을 이용하기를 원하는 어떤 경우이든, 여러분은 해당 제네릭 타입 파라미터의 타입내에 트레이트 바운드를 명시할 필요가 있습니다. 이제 우리는 Listing 10-5에서 제네릭 타입 파라미터를 사용하는 `largest` 함수의 정의를 고칠 수 있습니다! 우리가 그 코드를 치워뒀을 때, 아래와 같은 에러를 봤었지요:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
|
5 |         if item > largest {
|             ^^^^
|
note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

`largest`의 본체 내에서 큰 부등호 연산자를 사용하여 타입 `T`의 두 값을 비교할 수 있길 원했습니다. 이 연산자는 표준 라이브러리 트레이트인 `std::cmp::PartialOrd` 상에서 기본 메소드로 정의되어 있습니다. 따라서 큰 부등호 연산자를 사용할 수 있도록 하기 위해서는, `T`에 대한 트레이트 바운드 내에 `PartialOrd`를 특정하여 `largest` 함수가 비교 가능한 어떤 타입의 슬라이스에 대해 작동하도록 할 필요가 있습니다. `PartialOrd`는 프렐루드(prelude)에 포함되어 있기 때문에 따로 스코프 내로 가져올 필요는 없습니다.

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

이 코드를 컴파일하면, 다른 에러를 얻게 됩니다:

```
error[E0508]: cannot move out of type `<T>`, a non-copy array
--> src/main.rs:4:23
|
4 |     let mut largest = list[0];
|           ----- ^^^^^^^^ cannot move out of here
|           |
|           hint: to prevent move, use `ref largest` or `ref mut largest`  

error[E0507]: cannot move out of borrowed content
--> src/main.rs:6:9
|
6 |     for &item in list.iter() {
|           ^
|           |
|           |           hint: to prevent move, use `ref item` or `ref mut item`  

|           cannot move out of borrowed content
```

이 에러에 대한 열쇠는 `cannot move out of type [T], a non-copy array`에 있습니다.

`largest` 함수의 제네릭 없는 버전에서, 우리는 고작 가장 큰 `i32` 혹은 `char`를 찾는 시도만 했습니다. 4

장에서 논의한 바와 같이, 고정된 크기를 갖는 `i32`와 `char`와 같은 타입들은 스택에 저장될 수 있으며, 따라서 이 타입들은 `Copy` 트레이트를 구현하고 있습니다. 우리가 `largest` 함수를 제네릭으로 바꿨을 때, 이제는 `list` 파라미터가 `Copy` 트레이트를 구현하지 않은 타입을 가질 가능성도 생기는데, 이는 곧 `list[0]`의 값을 `largest` 변수로 소유권을 옮기지 못할 것이라는 의미입니다.

만약 이 코드를 오직 `Copy`가 구현된 타입들을 가지고 호출하도록 하는 것만 원한다면, `T`의 트레이트 바운드에 `Copy`를 추가할 수 있습니다! Listing 10-15는 `largest`로 넘겨지는 슬라이스 내의 값의 타입이 `i32`와 `char`처럼 `PartialOrd` 및 `Copy` 트레이트 모두를 구현했을 때에 한하여 컴파일되는 제네릭 `largest` 함수의 완전체 코드를 보여줍니다:

Filename: src/main.rs

```
use std::cmp::PartialOrd;

fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest(&chars);
    println!("The largest char is {}", result);
}
```

Listing 10-15: `PartialOrd`와 `Copy` 트레이트를 구현한 어떠한 제네릭 타입 상에서 동작하는 `largest` 함수의 동작 가능한 정의

만일 우리의 `largest` 함수를 `Copy` 트레이트를 구현한 타입에 대한 것으로만 제한하길 원치 않는다면, `T`가 `Copy` 대신 `clone` 트레이트 바운드를 갖도록 명시하여 `largest` 함수가 소유권을 갖길 원하는 경우 슬라이스의 각 값이 복제되도록 할 수도 있습니다. 그러나 `clone` 함수를 이용한다는 것은 더 많은 힙 할당을 할 수 있다는 것이고, 힙 할당은 많은 양의 데이터에 대해서 동작할 경우 느릴 수 있습니다. `largest`를 구현

하는 또다른 방법은 함수가 슬라이스 내의 `T` 값에 대한 참조자를 반환하도록 하는 것입니다. 만약 반환 타입을 `T` 대신 `&T`로 바꾸고 함수의 본체가 참조자를 반환하도록 바꾼다면, `Clone`이나 `Copy` 트레이트 바운드도 필요치 않으며 어떠한 힘 할당도 하지 않게 될 것입니다. 여러분이 직접 이 대안 해결책을 구현해보세요!

트레이트와 트레이트 바운드는 중복을 제거하기 위하여 제네릭 타입 파라미터를 사용하는 코드를 작성할 수 있도록 해주지만, 여전히 컴파일러에게 해당 제네릭 타입이 어떤 동작을 할 필요가 있는지를 정확히 명시하도록 해줍니다. 컴파일러에게 트레이트 바운드를 제공하기 때문에, 우리 코드와 함께 이용되는 모든 구체적인 타입들이 정확한 동작을 제공하는지를 확인할 수 있습니다. 동적 타입 언어에서는, 어떤 타입에 대해 어떤 메소드를 호출하는 시도를 했는데 해당 타입이 그 메소드를 구현하지 않았다면, 런타임에 에러를 얻게 됩니다. 러스트는 이러한 에러들을 컴파일 타임으로 옮겨서 우리의 코드가 실행 가능하기 전에 그 문제들을 해결하도록 우리를 강제합니다. 이에 더해서, 우리는 런타임에 해당 동작에 대한 검사를 하는 코드를 작성할 필요가 없는데, 우리는 이미 컴파일 타임에 이를 확인했기 때문이며, 이는 제네릭의 유연성을 포기하지 않고도 다른 언어들에 비해 성능을 향상시킵니다.

우리가 심지어 아직 알아채지도 못한 *라이프타임(lifetime)*이라 불리는 또다른 종류의 제네릭이 있습니다. 라이프타임은 어떤 타임이 우리가 원하는 동작을 갖도록 확신하는데 도움을 주기 보다는, 참조자들이 우리가 원하는 만큼 오랫동안 유효한지를 확신하도록 도와줍니다. 라이프타임이 어떤 식으로 그렇게 하는지를 배워봅시다.

라이프타임을 이용한 참조자 유효화

4장에서 참조자에 대한 이야기를 할 때, 중요한 디테일을 한 가지 남겨두었습니다: 러스트에서 모든 참조자는 **라이프타임(lifetime)**을 갖는데, 이는 해당 참조자가 유효한 스코프입니다. 대부분의 경우에서 타입들이 추론되는 것과 마찬가지로, 대부분의 경우에서 라이프타임 또한 암묵적이며 추론됩니다. 여러 가지 타입이 가능하기 때문에 우리가 타입을 명시해야 하는 때와 비슷하게, 참조자의 라이프타임이 몇몇 다른 방식으로 연관될 수 있는 경우들이 있으므로, 러스트는 우리에게 제네릭 라이프타임 파라미터를 이용하여 이 관계들을 명시하기 요구하여 런타임에 실제 참조자가 확실히 유효하도록 확신할 수 있도록 합니다.

네 그렇습니다. 이러한 개념은 다소 흔치 않으며, 여러분들이 다른 프로그래밍 언어에서 사용해온 도구들과는 다른 것입니다. 몇 가지 측면에서, 라이프타임은 러스트의 가장 독특한 기능입니다.

라이프타임은 이 장에서 전체를 다룰 수 없는 큰 주제이므로, 이 장에서는 여러분이 이 개념에 친숙해질 수 있도록 여러분이 라이프타임 문법을 맞닥뜨릴 흔한 경우에 대해 다룰 것입니다. 19장에서는 라이프타임이 할 수 있는 좀 더 상급 정보를 다룰 것입니다.

라이프타임은 맹글링 참조자를 방지합니다

라이프타임의 주목적은 맹글링 참조자(dangling reference)를 방지하는 것인데, 맹글링 참조자는 프로그램이 우리가 참조하기로 의도한 데이터가 아닌 다른 데이터를 참조하는 원인이 됩니다. Listing 10-16의 프로그램과 같이 외부 스코프와 내부 스코프를 가진 프로그램을 생각해봅니다. 외부 스코프는 `r`이라는 이름의 변수를 초기값 없이 선언하였고, 내부 스코프는 `x`라는 이름의 변수를 초기값 5와 함께 선언했습니다. 내부 스코프 내에서, `x`의 참조자를 `r`에 대입하도록 시도했습니다. 그 후 내부 스코프는 끝났고, `r`의 값을 출력하도록 시도했습니다:

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

Listing 10-16: 스코프 밖으로 벗어난 값에 대한 참조자를 사용하는 시도

초기화되지 않은 변수는 사용할 수 없습니다

다음에 나올 몇 가지 예제는 초기값을 주지 않고 변수를 선언하고 있으며, 따라서 해당 변수의 이름이 외부 스코프에 존재하고 있습니다. 이는 러스트가 널(null) 값을 갖지 않는다는 개념과 충돌을 일으키는 것처럼 보일지도 모릅니다. 그러나 우리가 값을 제공하기 전에 변수를 사용하고자 시도하면, 컴파일 에러가 나올 것입니다. 시도해 보세요!

이 코드를 컴파일하면, 다음과 같은 에러가 나타날 것입니다:

```
error: `x` does not live long enough
|
6 |         r = &x;
|             - borrow occurs here
7 |     }
|     ^ `x` dropped here while still borrowed
...
10 | }
| - borrowed value needs to live until here
```

변수 `x`는 "충분히 오래 살지 못한다(does not live long enough)"고 합니다. 왜 안될까요? `x`는 7번 라인의 닫는 중괄호 기호에 도달했을 때 내부 스코프가 끝나면서 스코프 밖으로 벗어날 것입니다. 그러나 `r`은 외부 스코프에 대해 유효합니다; 이쪽의 스코프가 더 크고 우리는 이쪽이 "더 오래 산다"라고 말합니다. 만일 러스트가 이 코드를 작동하도록 허용한다면, `r`은 `x`가 스코프 밖으로 벗어났을 때 할당이 해제되는 메모리를 참조하게 될 것이고, `r`을 가지고 시도하려 했던 어떤 것인든 정확히 동작하지 않게 될 것입니다. 그렇다면 러스트는 이 코드가 허용되어서는 안 된다는 것을 어떻게 결정할까요?

빌림 검사기(Borrow checker)

빌림 검사기(borrow checker)라고 불리는 컴파일러의 부분이 모든 빌림이 유효한지를 결정하기 위해 스코프를 비교합니다. Listing 10-17은 변수들의 라이프타임을 보여주는 주석과 함께 Listing 10-16과 동일한 예제를 보여줍니다:

```
{
    let r;          // -----+-- 'a
                    //      |
{
    //      |
    let x = 5;   // -+-----+-- 'b
    r = &x;       //      |      |
}
                    //      |
                    //      |
println!("r: {}", r); //      |
                    //      |
                    // -----+
```

Listing 10-17: 각각 'a'과 'b'로 명명된 r과 x의 라이프타임에 대한 주석

우리는 r의 라이프타임을 'a'라고 명명하였고, x의 라이프타임을 'b'라고 명명하였습니다. 보시다시피, 내부의 'b' 블록은 외부의 'a' 라이프타임 블록에 비해 훨씬 작습니다. 컴파일 타임에서, 러스트는 두 라이프타임의 크기를 비교하고 r이 'a' 라이프타임을 가지고 있지만, 'b' 라이프타임을 가지고 있는 어떤 오브젝트를 참조하고 있음을 보게 됩니다. 'b' 라이프타임이 'a' 라이프타임에 비해 작기 때문에 러스트 컴파일러는 이 프로그램을 거부합니다: 참조자의 주체가 참조자만큼 오래 살지 못하고 있으니까요.

프로그래밍 참조자를 만드는 시도가 없고 예러 없이 컴파일되는 Listing 10-18의 예제를 살펴봅시다:

```
{
    let x = 5;           // -----+-- 'b
                        //     |
    let r = &x;          // -+---+-- 'a
                        //     |   |
    println!("r: {}", r); //     |   |
                        // -+   |
}                   // -----+
```

Listing 10-18: 데이터가 참조자에 비해 더 긴 라이프타임을 갖고 있기 때문에 유효한 참조자

여기서 x는 라이프타임 'b'를 갖고 있는데, 위의 경우 'a'에 비해 더 큽니다. 이는 r이 x를 참고할 수 있음을 의미합니다: 러스트는 r의 참조자가 x가 유효한 동안 언제나 유효할 것이라는 점을 알고 있습니다.

지금까지 참조자의 라이프타임이 구체적인 예제 어디에 나오는지를 보았고 러스트가 어떻게 라이프타임을 분석하여 참조자가 항상 유효하도록 확인시키는지를 논의했으니, 이제 함수의 내용물 내에 있는 파라미터와 반환 값에 대한 제네릭 라이프타임에 대하여 이야기해 봅시다.

함수에서의 제네릭 라이프타임

두 스트링 슬라이스 중에서 긴 쪽을 반환하는 함수를 작성해 봅시다. 이 함수에 두 개의 스트링 슬라이스를 넘겨서 호출할 수 있기를 원하고, 스트링 슬라이스를 반환하기를 원합니다. Listing 10-19의 코드는

`longest` 함수를 구현하면 `The longest string is abcd`를 출력해야 합니다:

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

Listing 10-19: 두 스트링 슬라이스 중 긴 쪽을 찾기 위해 `longest` 함수를 호출하는 `main` 함수

`longest` 함수가 인자의 소유권을 얻는 것을 원치 않기 때문에 스트링 슬라이스들을 (4장에서 이야기했던 것처럼 이들은 참조자입니다) 파라미터로서 갖는 함수를 원한다는 점을 주목하세요. 우리는 함수가 `String`의 슬라이스 (이는 변수 `string1`의 타입입니다)는 물론 스트링 리터럴 (이는 변수 `string2`가 담고 있는 것이지요) 또한 받아들일 수 있기를 원하고 있습니다.

왜 이들이 우리가 원하는 인자 들인지에 대한 더 많은 논의에 대해서는 4장의 "인자로서의 스트링 슬라이스"를 참조하세요.

만일 Listing 10-20에서 보는 바와 같이 `longest` 함수를 구현하는 시도를 한다면, 이는 컴파일되지 않을 것입니다:

Filename: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-20: 두 스트링 슬라이스 중 긴 쪽을 반환하는 `longest` 함수의 구현체, 그러나 아직 컴파일되지 않음

대신 우리는 라이프타임에 대해 이야기하는 다음과 같은 에러를 얻습니다:

```
error[E0106]: missing lifetime specifier
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |           ^ expected lifetime parameter
  |
= help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`
```

이 도움말은 반환 타입에 대하여 제네릭 라이프타임 파라미터가 필요하다는 것을 말해주고 있는데, 왜냐하면 반환되는 참조자가 `x`를 참조하는지 혹은 `y`를 참조하는지를 러스트가 말할 수 없기 때문입니다. 사실, 우리 또한 모르는데, 이 함수의 본체 내의 `if` 블록은 `x`의 참조자를 반환하고 `else` 블록은 `y`의 참조자를 반환하기 때문입니다!

우리가 이 함수를 정의하고 있는 시점에서, 우리는 이 함수에 넘겨지게 될 구체적인 값을 모르므로, `if` 케이스가 실행될지 혹은 `else` 케이스가 실행될지는 알 수 없습니다. 또한 함수에 넘겨지게 될 참조자의 구체적인 라이프타임을 알지 못하므로, 우리가 반환하는 참조자가 항상 유효한지를 결정하기 위해서 Listing 10-17과 10-18에서 했던 것과 같이 스코프를 살펴볼 수도 없습니다. 빌림 검사기 또한 이를 결정할 수 없는데, 그 이유는 `x`와 `y`의 라이프타임이 반환 값의 라이프타임과 어떻게 연관되어 있는지 알지 못하기 때문입니다. 우리는 참조자들 간의 관계를 정의하는 제네릭 라이프타임 파라미터를 추가하여 빌림 검사기가 분석을 수행할 수 있도록 할 것입니다.

라이프타임 명시 문법

라이프타임 명시는 연관된 참조자가 얼마나 오랫동안 살게 되는지를 바꾸지는 않습니다. 함수의 시그니처가 제네릭 타입 파라미터를 특정할 때 이 함수가 어떠한 타입이든 허용할 수 있는 것과 같은 방식으로, 함수의 시그니처가 제네릭 라이프타임 파라미터를 특정할 때라면 이 함수는 어떠한 라이프타임을 가진 참조자라도 허용할 수 있습니다. 라이프타임 명시가 하는 것은 여러 개의 참조자에 대한 라이프타임들을 서로 연관 짓도록 하는 것입니다.

라이프타임 명시는 약간 독특한 문법을 갖고 있습니다: 라이프타임 파라미터의 이름은 어퍼스트로피 `!`로 시작해야 합니다. 라이프타임 파라미터의 이름은 보통 모두 소문자이며, 제네릭 타입과 비슷하게 그들의 이름은 보통 매우 짧습니다. `'a`는 대부분의 사람들이 기본적으로 사용하는 이름입니다. 라이프타임 파라미터 명시는 참조자의 `&` 뒤에 오며, 공백 문자가 라이프타임 명시와 참조자의 타입을 구분해줍니다.

여기 몇 가지 예제가 있습니다: 라이프타임 파라미터가 없는 `i32`에 대한 참조자, `'a`라고 명명된 라이프타임 파라미터를 가지고 있는 `i32`에 대한 참조자, 그리고 역시 라이프타임 `'a`를 갖고 있는 `i32`에 대한 가변 참조자입니다:

```
&i32          // a reference
'a i32      // a reference with an explicit lifetime
'a mut i32 // a mutable reference with an explicit lifetime
```

스스로에 대한 하나의 라이프타임 명시는 큰 의미를 가지고 있지 않습니다: 라이프타임 명시는 러스트에게 여러 개의 참조자에 대한 제네릭 라이프타임 파라미터가 서로 어떻게 연관되는지를 말해줍니다. 만일 라이프타임 `'a`를 가지고 있는 `i32`에 대한 참조자인 `first`를 파라미터로, 그리고 또한 라이프타임 `'a`를 가지고 있는 `i32`에 대한 또 다른 참조자인 `second`를 또 다른 파라미터로 가진 함수가 있다면, 이 두 개의 같은 이름을 가진 라이프타임 명시는 참조자 `first`와 `second`가 둘다 동일한 제네릭 라이프타임만큼 살아야 한

다는 것을 가리킵니다.

함수 시그니처 내의 라이프타임 명시

우리가 작업하고 있던 `longest` 함수의 내용 중에서 라이프타임 명시 부분을 살펴봅시다. 제네릭 타입 파라미터와 마찬가지로, 제네릭 라이프타임 파라미터도 함수 이름과 파라미터 리스트 사이에 꺠쇠괄호를 쓰고 그 안에 정의가 되어야 합니다. 우리가 파라미터들과 반환 값에서의 참조자들에 대해 러스트에게 말해주고 싶은 제약사항은 그들이 모두 동일한 라이프타임을 갖고 있어야 한다는 것인데, 이는 Listing 10-21에서 보는 바와 같이 우리가 '`a`'라고 명명하여 각각의 참조자에 추가할 것입니다:

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-21: 시그니처 내의 모든 참조자들이 동일한 라이프타임 '`a`'를 가지고 있어야 함을 특정한 `longest` 함수 정의

이는 컴파일될 것이고 Listing 10-19에 있는 `main` 함수에서 사용되었을 때 우리가 원하는 결과를 만들어 줄 것입니다.

이 함수 시그니처는 이제 어떤 라이프타임 '`a`'에 대하여, 이 함수는 두 개의 파라미터를 갖게 될 것인데, 두 개 모두 적어도 라이프타임 '`a`'만큼 살아있는 스트링 슬라이스임을 말해줍니다. 이 함수는 또한 적어도 라이프타임 '`a`'만큼 살아있는 스트링 슬라이스를 반환할 것입니다. 이는 러스트에게 우리가 강제하고 싶은 것을 말해주는 계약입니다.

이 함수 시그니처 내에 라이프타임 파라미터를 특정함으로써, 우리는 함수에 넘겨지거나 반환되는 어떠한 값들의 라이프타임도 바꾸지 않지만, 이 계약에 부합하지 않는 어떠한 값들도 빌림 검사기에 의해 거부되어야 함을 말해주는 것입니다. 이 함수는 `x`와 `y`가 정확히 얼마나 오래 살게 될지 알지 못하지만 (혹은 알 필요가 없지만), 다만 이 시그니처를 만족시킬 '`a`'에 대입될 수 있는 어떤 스코프가 있음을 알아야 할 필요가 있을 뿐입니다.

함수 안에 라이프타임을 명시할 때, 이 명시는 함수 시그니처에 붙어 있으며, 함수의 본체 내에의 어떠한 코드에도 붙어있지 않습니다. 이는 러스트가 다른 도움 없이 함수 내의 코드를 분석할 수 있지만, 함수가 그 함수 밖의 코드에서의 참조자를 가지고 있을 때, 인자들 혹은 반환 값들의 라이프타임이 함수가 호출될 때마다 달라질 가능성이 있기 때문입니다. 이는 러스트가 발견해내기에는 너무나 비용이 크고 종종 불가능할 것입니다.

이 경우, 우리는 스스로 라이프타임을 명시할 필요가 있습니다.

구체적인 참조자들이 `longest`로 넘겨질 때, `'a`에 대입되게 되는 구체적인 라이프타임은 `y`의 스코프와 겹치는 `x` 스코프의 부분입니다. 스코프는 언제나 중첩되기 때문에, 이것이 제네릭 라이프타임 `'a`이다라고 말하는 또 다른 방법은 `x`와 `y`의 라이프타임 중에서 더 작은 쪽과 동일한 구체적인 라이프타임을 구하는 것 일 겁니다. 반환되는 참조자에 대해서도 같은 라이프타임 파라미터인 `'a`를 명시했으므로, 반환되는 참조자도 `x` 와 `y`의 라이프타임 중 짧은 쪽만큼은 길게 유효함을 보장할 것입니다.

서로 다른 구체적인 라이프타임을 가진 참조자들을 넘김으로써 이것이 `longest` 함수의 사용을 어떻게 제한하는지 봅시다. Listing 10-22는 아무 언어에서나 여러분의 직관에 부합될 간단한 예제입니다:

`string1`은 외부 스코프가 끝날 때까지 유효하고 `string2`는 내부 스코프가 끝날 때까지 유효하며, `result`는 내부 스코프가 끝날 때까지 유효한 무언가를 참조합니다. 빌림 검사기는 이 코드를 승인합니다; 이는 컴파일되며 실행했을 때 `The longest string is long string is long`를 출력합니다:

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

Listing 10-22: 서로 다른 구체적인 라이프타임을 가진 `String` 값의 참조자들을 이용한 `longest` 함수의 사용

다음으로, `result`의 참조자의 라이프타임이 두 인자들의 라이프타임보다 작아야 함을 보여줄 예제를 시도해봅시다. 우리는 `result`의 선언부를 내부 스코프 밖으로 옮길 것이지만, `result` 변수에 대한 값의 대입은 `string2`가 있는 스코프 내에 남겨둘 것입니다. 다음으로, `result`를 이용하는 `println!` 구문을 내부 스코프 바깥에, 내부 스코프가 끝나는 시점으로 옮기겠습니다. 이렇게 수정한 Listing 10-23의 코드는 컴파일되지 않을 것입니다:

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

Listing 10-23: `string2`가 스코프 밖으로 벗어난 후에 `result`를 사용하고자 하는 시도는 컴파일되지 않습니다

만일 이를 컴파일하고자 시도하면, 다음과 같은 에러를 얻습니다:

```
error: `string2` does not live long enough
|
6 |         result = longest(string1.as_str(), string2.as_str());
|                         ----- borrow occurs here
7 |
8 |     }
|     ^ `string2` dropped here while still borrowed
9 |     println!("The longest string is {}", result);
|     }
| - borrowed value needs to live until here
```

이 에러는 `result`가 `println!`에서 유효하기 위해서는 `string2`가 외부 스코프의 끝까지 유효할 필요가 있음을 말해줍니다. 러스트는 이를 알고 있는데, 그 이유는 우리가 함수의 파라미터들과 반환 값에 대해 동일한 라이프타임 파라미터 '`a`'를 명시했기 때문입니다.

우리는 인간으로서 이 코드를 살펴볼 수 있고 `string1`이 더 길기 때문에 `result`는 `string1`의 참조자를 담게 될 것이라는 점을 알 수 있습니다. `string1`이 스코프 밖으로 아직 벗어나지 않았기 때문에, `string1`의 참조자는 `println!` 구문에서 여전히 유효할 것입니다. 그렇지만, 우리가 러스트에게 라이프타임 파라미터를 가지고 말해준 것은 `longest` 함수에 의해 반환되는 참조자의 라이프타임이 인자로 넘겨 준 라이프타임들 중 작은 쪽과 동일하다는 것이었지요. 따라서, 빌림 검사기는 잠재적으로 유효하지 않은 참조자를 가질 수 있는 문제로 인해 Listing 10-23의 코드를 허용하지 않습니다.

`longest` 함수에 넘겨질 참조자들의 값과 라이프타임들, 그리고 반환된 참조자를 어떻게 이용하는지를 다양화하여 더 많은 실험들을 디자인해 시도해보세요. 컴파일하기 전에 여러분의 실험이 빌림 검사기를 통과할지 안 할지에 대한 가설을 세워보고, 여러분이 맞았는지 확인해보세요!

라이프타임의 측면에서 생각하기

라이프타임 파라미터를 특정하는 정확한 방법은 여러분의 함수가 어떤 일을 하고 있는가에 따라 달린 문제입니다. 예를 들면, `longest` 함수의 구현을 제일 긴 스트링 슬라이스 대신 항상 첫 번째 인자를 반환하도록 바꾸었다면, `y` 파라미터에 대한 라이프타임을 특정할 필요는 없을 것입니다. 아래 코드는 컴파일됩니다:

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

이 예제에서, 파라미터 `x`와 반환 값에 대한 라이프타임 파라미터 `'a`는 특정하였지만, 파라미터 `y`는 특정하지 않았는데, 그 이유는 `y`의 라이프타임이 `x` 혹은 반환 값의 라이프타임과 어떠한 관련도 없기 때문입니다.

함수로부터 참조자를 반환할 때, 반환 타입에 대한 라이프타임 파라미터는 인자 중 하나의 라이프타임 파라미터와 일치할 필요가 있습니다. 만일 반환되는 참조가 인자들 중 하나를 참조하지 않는다면, 다른 유일한 가능성은 이 함수 내에서 생성된 값을 참조하는 경우인데, 이 값은 함수가 끝나는 시점에서 스코프 밖으로 벗어나기 때문에 데블링 참조자가 될 것입니다. `longest` 함수에 대한 아래와 같은 구현 시도는 컴파일되지 않습니다:

Filename: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

우리가 반환 타입에 대해 라이프타임 파라미터 `'a`를 특정했을지라도, 이러한 구현은 컴파일에 실패하게 되는데 이는 반환되는 값의 라이프타임이 파라미터의 라이프타임과 아무런 관련이 없기 때문입니다. 여기 우리가 얻게 되는 에러 메시지를 보시죠:

```

error: `result` does not live long enough
|
3 |     result.as_str()
|     ^^^^^^ does not live long enough
4 | }
| - borrowed value only lives until here
|
note: borrowed value must be valid for the lifetime 'a as defined on the
block
at 1:44...
|
1 | fn longest<'a>(x: &str, y: &str) -> &'a str {
|           ^

```

문제는 `result`가 `longest` 함수가 끝나는 지점에서 스코프 밖으로 벗어나게 되어 메모리 해제가 일어나게 되는데, 이 함수로부터 `result`의 참조자를 반환하려는 시도를 한다는 점입니다. 이 데그링 참조자를 변경시킬 라이프타임 파라미터를 특정할 방법은 없으며, 러스트는 우리가 데그링 참조자를 만들게끔 놔두지 않습니다. 이 경우, 가장 좋은 수정 방법은 참조자보다는 차라리 값을 소유한 데이터 타입을 리턴하도록 하여 호출하는 함수가 값을 할당 해제하도록 하는 것입니다.

궁극적으로, 라이프타임 문법은 함수들의 다양한 인자들과 반환 값 사이를 연결하는 것에 대한 것입니다. 이들이 일단 연결되고 나면, 러스트는 메모리에 안전한 연산들을 허용하고 데그링 포인터를 생성하거나 그렇지 않은 경우 메모리 안전을 위해 될 연산들을 배제하기에 충분한 정보를 갖게 됩니다.

구조체 정의 상에서의 라이프타임 명시

현재까지 우리는 소유권 있는 타입만 들고 있는 구조체들만 정의해왔습니다. 구조체가 참조자를 들고 있도록 할 수 있지만, 구조체 정의 내의 모든 참조자들에 대하여 라이프타임을 표시할 필요가 있습니다. Listing 10-24에 스트링 슬라이스를 들고 있는 `ImportantExcerpt`라고 명명된 구조체가 있습니다:

Filename: src/main.rs

```

struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentence };
}

```

Listing 10-24: 참조자를 들고 있는 구조체, 따라서 정의 부분에 라이프타임 명시가 필요합니다

이 구조체는 스트링 슬라이스를 담을 수 있는 `part`라는 하나의 필드를 갖고 있는데, 이것이 참조자입니다. 제네릭 데이터 타입과 마찬가지로, 제네릭 라이프타임 파라미터의 이름을 구조체의 이름 뒤편에 꺠쇠괄호 안에다 선언하여 구조체 정의의 본체 내에서 이 라이프타임 파라미터를 이용할 수 있도록 해야 합니다.

여기 이 `main` 함수는 변수 `novel`이 소유하고 있는 `String`의 첫 문장에 대한 참조자를 들고 있는 `ImportantExcerpt` 구조체의 인스턴스를 생성합니다.

라이프타임 생략

이 절에서, 우리는 모든 참조자가 라이프타임을 가지고 있으며, 참조자를 사용하는 함수나 구조체에 대하여 라이프타임 파라미터를 특정할 필요가 있다고 배웠습니다. 하지만, Listing 10-25에서 다시 보여주듯이, 4장의 "스트링 슬라이스"절의 함수는 라이프타임 명시 없이도 컴파일이 됐었지요:

Filename: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Listing 10-25: 파라미터와 반환 값의 타입이 참조자임에도 불구하고 라이프타임 명시 없이 컴파일되었던, 4장에서 정의한 바 있는 함수

이 함수가 라이프타임 없이 컴파일되는 이유는 역사가 있습니다: 1.0 이전 시절의 러스트에서는 이 코드가 실제로 컴파일되지 않았습니다. 모든 참조자들은 명시적인 라이프타임이 필요했지요. 그 시절, 함수 시그니처는 아래와 같이 작성되었습니다:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

수많은 러스트 코드를 작성하고 난 후, 러스트 팀은 러스트 프로그래머들이 특정한 상황에서 똑같은 라이프타임 명시를 계속하여 타이핑하고 있다는 사실을 발견하게 되었습니다. 이 상황들은 예측 가능하며 몇 가지 결정론적인 패턴을 따르고 있었습니다. 그리하여 러스트 팀은 러스트 컴파일러 코드 내에 이 패턴들을 프로그래밍하여 이러한 상황 내에서는 프로그래머가 명시적으로 라이프타임 명시를 추가하도록 강제하지 않고 빌림

검사기가 라이프타임을 추론할 수 있도록 하였습니다.

더 많은 결정론적인 패턴들이 출현하여 컴파일러 내에 추가될 가능성이 충분하기에 이러한 러스트의 역사에 대해 언급하였습니다. 나중에는 더욱 적은 라이프타임 명시만이 필요할지도 모르지요.

참조자에 대한 러스트의 분석 기능 내에 프로그래밍된 패턴들을 일컬어 **라이프타임 생략 규칙(lifetime elision rules)**이라고 합니다. 이들은 프로그래머가 따라야 하는 규칙들이 아닙니다; 이 규칙들은 컴파일러가 고려할 특정한 경우의 집합이고, 여러분의 코드가 이러한 경우에 들어맞으면, 여러분은 명시적으로 라이프타임을 작성할 필요가 없어집니다.

생략 규칙들은 모든 추론을 제공하지는 않습니다: 만일 러스트가 결정론적으로 이 규칙들을 적용했지만 여전히 참조자들이 어떤 라이프타임을 가지고 있는지에 대하여 모호하다면, 해당하는 남은 참조자들의 라이프타임이 어떻게 되어야 하는지에 대해 추측하지 않을 것입니다. 이러한 경우, 컴파일러는 여러분에게 이 참조자들이 서로 어떻게 연관되는지에 대하여 여러분의 의도에 맞게끔 라이프타임을 추가함으로써 해결 가능한 에러를 표시할 것입니다.

먼저 몇 가지 정의들을 봅시다: 함수나 메소드의 파라미터에 대한 라이프타임을 **입력 라이프타임(input lifetime)**이라고 하며, 반환 값에 대한 라이프타임을 **출력 라이프타임(output lifetime)**이라고 합니다.

이제 명시적인 라이프타임이 없을 때 참조자가 어떤 라이프타임을 가져야 하는지 알아내기 위해서 컴파일러가 사용하는 규칙들을 봅시다. 첫 번째 규칙은 입력 라이프타임에 적용되고, 다음의 두 규칙들은 출력 라이프타임에 적용됩니다. 만일 컴파일러가 이 세 가지 규칙의 끝에 도달하고 여전히 라이프타임을 알아낼 수 없는 참조자가 있다면, 컴파일러는 에러와 함께 멈출 것입니다.

1. 참조자인 각각의 파라미터는 고유한 라이프타임 파라미터를 갖습니다. 바꿔 말하면, 하나의 파라미터를 갖는 함수는 하나의 라이프타임 파라미터를 갖고: `fn foo<'a>(x: &'a i32)`, 두 개의 파라미터를 갖는 함수는 두 개의 라이프타임 파라미터를 따로 갖고: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`, 이와 같은 식입니다.
2. 만일 정확히 딱 하나의 라이프타임 파라미터만 있다면, 그 라이프타임이 모든 출력 라이프타임 파라미터들에 대입됩니다: `fn foo<'a>(x: &'a i32) -> &'a i32`.
3. 만일 여러 개의 입력 라이프타임 파라미터가 있는데, 메소드라서 그중 하나가 `&self` 혹은 `&mut self`라고 한다면, `self`의 라이프타임이 모든 출력 라이프타임 파라미터에 대입됩니다. 이는 메소드의 작성을 더욱 멋지게 만들어줍니다.

우리가 직접 컴파일러가 된 척하여 Listing 10-25의 `first_word` 함수의 시그니처에 있는 참조자들의 라이프타임이 무엇인지 알아내기 위해 이 규칙들을 적용해 봅시다. 이 시그니처는 참조자들과 관련된 아무런 라이프타임도 없이 시작합니다:

```
fn first_word(s: &str) -> &str {
```

그러면 (컴파일러로서의) 우리는 첫 번째 규칙을 적용하는데, 이는 각각의 파라미터가 고유의 라이프타임을 갖는다고 말해주고 있습니다. 우리는 이를 평범하게 '`a`'라고 명명할 것이며, 따라서 이제 시그니처는 다음과 같습니다:

```
fn first_word<'a>(s: &'a str) -> &str {
```

두 번째 규칙 상에 놓이게 되는데, 이는 정확히 단 하나의 입력 라이프타임만 존재하기 때문에 적용됩니다. 두 번째 규칙은 그 하나의 입력 파라미터에 대한 라이프타임이 출력 라이프타임에 대입된다고 말하고 있으므로, 이제 시그니처는 다음과 같아집니다:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

이제 이 함수 시그니처의 모든 참조자들이 라이프타임을 갖게 되었고, 컴파일러는 프로그래머에게 이 함수 시그니처 내의 라이프타임을 명시하도록 요구하지 않고도 분석을 계속할 수 있게 되었습니다.

또 다른 예제를 해보려는데, 이번에는 Listing 10-20에서와 같이 우리가 처음 시작할 때의 아무런 라이프타임 파라미터도 가지고 있지 않은 `longest` 함수를 가지고 해 봅시다:

```
fn longest(x: &str, y: &str) -> &str {
```

다시 한번 우리가 컴파일러가 된 척하여, 첫 번째 규칙을 적용해봅시다: 각각의 파라미터는 고유의 라이프타임을 갖습니다. 이번에는 두 개의 파라미터들이 있으므로, 두 개의 라이프타임을 갖게 됩니다:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

두 번째 규칙을 살펴봤을 때, 하나 이상의 입력 라이프타임이 있으므로 적용되지 않습니다. 세번째 규칙을 살펴봤을 때, 이 또한 적용되지 않는데 이는 이것이 메소드가 아니라 함수이고, 따라서 어떠한 파라미터도 `self`가 아니기 때문입니다. 따라서 규칙이 더 이상 남아있지 않은데, 우리는 아직 반환 다임의 라이프타임이 무엇인지 알아내지 못했습니다. 이것이 바로 Listing 10-20의 코드를 컴파일하려 시도했을 때 에러가 발생한 이유입니다: 컴파일러는 자신이 알고 있는 라이프타임 생략 규칙들을 통해 작업을 수행했지만, 여전히 이 시그니처의 참조자들에 대한 모든 라이프타임을 알아낼 수 없으니까요.

세번째 규칙이 오직 메소드 시그니처에 대해서만 실제로 적용되므로, 이제 그러한 경우에서의 라이프타임을 살펴보고, 어째서 세번쩨 규칙이 메소드 시그니처의 라이프타임을 매우 흔하게 생략해도 된다는 것을 의미하는지 알아봅시다.

메소드 정의 내에서의 라이프타임 명시

라이프타임을 가진 구조체에 대한 메소드를 구현할 때, 문법은 또다시 Listing 10-10에서 보신 바와 같이 제

네릭 타입 파라미터의 그것과 같습니다: 라이프타임 파라미터가 선언되고 사용되는 곳은 라이프타임 파라미터가 구조체의 필드들 혹은 메소드 인자와 반환 값과 연관이 있는지 없는지에 따라 달린 문제입니다.

구조체 필드를 위한 라이프타임 이름은 언제나 `impl` 키워드 뒤에 선언되어야 하며, 그리고 나서 구조체의 이름 뒤에 사용되어야 하는데, 이 라이프타임들은 구조체 타입의 일부이기 때문입니다.

`impl` 블록 안에 있는 메소드 시그니처에서, 참조자들이 구조체 필드에 있는 참조자들의 라이프타임과 묶일 수도 있고, 혹은 서로 독립적일 수도 있습니다. 여기에 더해, 라이프타임 생략 규칙이 종종 적용되어 메소드 시그니처 내에 라이프타임 명시를 할 필요가 없습니다. Listing 10-24에서 정의했던 `ImportantExcerpt`라는 이름의 구조체를 이용한 몇 가지 예제를 볼시다.

먼저, 여기 `level`라는 이름의 메소드가 있습니다. 파라미터는 오직 `self`에 대한 참조자이며, 반환 값은 무언가에 대한 참조자가 아닌, 그냥 `i32`입니다:

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

`impl` 뒤의 라이프타임 파라미터 선언부와 타입 이름 뒤에서 이를 사용하는 것이 필요하지만, 첫 번째 생략 규칙 때문에 `self`로의 참조자의 라이프타임을 명시할 필요는 없습니다.

아래는 세번째 라이프타임 생략 규칙이 적용되는 예제입니다:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

두 개의 입력 라이프타임이 있으므로, 러스트는 첫 번째 라이프타임 생략 규칙을 적용하여 `&self`와 `announcement`에게 각각 라이프타임을 부여합니다. 그다음, 파라미터 중 하나가 `&self`이므로, 반환 타입은 `&self`의 라이프타임을 얻고, 모든 라이프타임들이 추론되었습니다.

정적 라이프타임(Static lifetime)

우리가 논의할 필요가 있는 특별한 라이프타임이 딱 하나 있습니다: 바로 `'static`입니다. `'static` 라이프타임은 프로그램의 전체 생애주기를 가리킵니다. 모든 스트링 리터럴은 `'static` 라이프타임을 가지고 있는데, 아래와 같이 명시하는 쪽을 선택할 수 있습니다:

```
let s: &'static str = "I have a static lifetime.;"
```

이 스트링의 텍스트는 여러분의 프로그램의 바이너리 내에 직접 저장되며 여러분 프로그램의 바이너리는 항상 이용이 가능하지요. 따라서, 모든 스트링 리터럴의 라이프타임은 `'static`입니다.

여러분은 어쩌면 여러 메시지 도움말에서 `'static` 라이프타임을 이용하라는 제안을 보셨을지도 모릅니다만, 참조자의 라이프타임으로서 `'static`으로 특정하기 전에, 여러분이 가지고 있는 참조자가 실제로 여러분 프로그램의 전체 라이프타임 동안 사는 것인지 대해 생각해보세요 (혹은 가능하다면 그렇게 오래 살게끔 하고 싶어 할지라도 말이죠). 대부분의 경우, 코드 내의 문제는 데그깅 참조자를 만드는 시도 혹은 사용 가능한 라이프타임들의 불일치이며, 해결책은 이 문제들을 해결하는 것이지 `'static` 라이프타임으로 특정하는 것이 아닙니다.

제네릭 타입 파라미터, 트레이트 바운드, 라이프타임을 함께 써보기

그럼 제네릭 타입 파라미터, 트레이트 바운드, 그리고 라이프타임이 하나의 함수에 모두 특정된 문법을 간단하게 살펴봅시다!

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

이것은 Listing 10-21에 나온 바 있는 두 스트링 슬라이스 중 긴 쪽을 반환하는 `longest` 함수지만, `ann`이라는 이름의 추가 인자를 가지고 있습니다. `ann`의 타입은 제네릭 타입 `T`인데, `where` 절을 가지고 특정한 바와 같이 `Display` 트레이트를 구현한 어떤 타입으로도 채워질 수 있습니다. 이 추가 인자는 함수가 스트링 슬라이스들의 길이를 비교하기 전 출력될 것인데, 이것이 `Display` 트레이트 바운드가 필요한 이유지요. 라이프타임이 제네릭의 한 종류이므로, 라이프타임 파라미터 `'a`와 제네릭 타입 파라미터 `T` 둘 모두에 대한 선언이 함수 이름 뒤 꺽쇠괄호 내에 나열되어 있습니다.

정리

이번 절에서 참 많은 것을 다루었습니다! 이제 여러분은 제네릭 타입 파라미터, 트레이트와 트레이트 바운드, 그리고 제네릭 라이프타임 파라미터에 대해 알게되었으니, 여러분은 중복되지 않지만 많은 서로 다른 상황들에서 사용 가능한 코드를 작성할 준비가 되었습니다. 제네릭 타입 파라미터는 코드가 서로 다른 타입에 대해서 적용될 수 있음을 의미합니다. 트레이트와 트레이트 바운드는 그 타입이 제네릭일지라도 해당 타입들이 코드에 필요한 동작을 할 수 있음을 보장합니다. 라이프타임 명시에 의해 특정된 참조자들의 라이프타임 간의 관계는 이 유연한 코드가 어떠한 데그링 참조자도 만들지 않을 것임을 확신시켜줍니다. 그리고 이 모든 것들이 컴파일 타임에 이루어지므로 런타임 성능에는 영향을 주지 않지요!

믿을진 모르겠지만, 이 부분에 대해 배울 것이 심지어 더 있습니다: 17장에서는 트레이트 객체(trait object)에 대해 다룰 예정인데, 이는 트레이트를 사용하는 또 다른 방법입니다. 19장에서는 라이프타임 명시를 포함하는 더 복잡한 시나리오를 다룰 것입니다. 20장에서는 더 고급 수준의 타입 시스템 특성을 다룰 것입니다. 하지만, 다음 절에서는 러스트에서 어떻게 테스트를 작성하여 우리의 코드가 우리가 원했던 방식대로 모든 기능들을 작동시킨다는 것을 확신할 수 있도록 하는 방법에 대해 이야기해봅시다!

자동화된 테스트 작성하기

프로그램 테스팅은 버그의 존재를 보여주는 매우 효율적인 방법일 수 있지만, 버그의 부재를 보여주기에는 절망적으로 불충분하다.

에츠하르 W. 데이크스트라(Edsger W. Dijkstra), "겸손한 프로그래머(The Humble Programmer)" (1972) 에츠하르 W. 데이크스트라(Edsger W. Dijkstra)는 그의 1972년 에세이 "겸손한 프로그램 (The Humble Programmer)"에서 "프로그램 테스팅은 버그의 존재를 보여주는 매우 효율적인 방법일 수 있지만, 버그의 부재를 보여주기에는 절망적으로 불충분하다"라고 말했습니다. 이는 우리가 할 수 있는 한 많은 테스트를 시도하지 않아도 된다는 의미가 아닙니다! 우리 프로그램이 정확하다는 것은 즉 우리가 의도한 바를 그대로 우리가 작성한 코드가 수행한다는 뜻입니다. 러스트는 정확성에 매우 많이 신경 쓴 프로그래밍 언어이지만, 정확성이란 복잡한 주제이며 증명하기 쉽지 않습니다. 러스트의 타입 시스템은 이 짐의 큰 부분을 짊어지고 있지만, 타입 시스템이 모든 종류의 부정 확성을 잡아낼 수는 없습니다. 러스트에는 보통 말하는 그런 자동화된 소프트웨어 테스트를 작성하기 위한 지원이 언어 내부에 포함되어 있습니다.

예를 들어 어떤 숫자든 입력되면 2를 더하는 `add_two`라는 함수를 작성한다 칩시다. 이 함수의 시그니처는 정수를 파라미터로 받아들여서 정수를 결과로 반환합니다. 이 함수를 구현하여 컴파일할 때, 러스트는 우리가 이제껏 봄은 모든 종류의 타입 검사 및 빌림 검사를 할 것입니다. 이러한 검사는, 이를테면 `String` 값이나 유효하지 않은 참조자를 이 함수로 넘기지 않음을 보장해 줄 것입니다. 그러나 러스트는 우리가 정확히 의도한 것을 이 함수가 수행하는가에 대해서는 검사할 수 없는데, 말하자면 파라미터 더하기 10 혹은 파라미터 빼기 50이 아니라 파라미터 더하기 2여야 합니다! 이러한 지점이 바로 테스트가 필요해지는 부분입니다.

예를 들면 우리가 3을 `add_two` 함수에 넘겼을 때, 반환 값은 5임을 단언하는(`assert`) 테스트를 작성할 수 있습니다. 우리는 어떤 종류의 코드 변경이라도 있을 때마다 기존의 정확히 동작하던 부분에 어떠한 변화도 없음을 확신할 수 있도록 이 테스트들을 실행할 수 있습니다.

테스팅은 복잡한 기술입니다: 하나의 장 내에서 어떻게 좋은 테스트를 작성하는지에 대한 모든 상세한 부분을 다룰 수는 없을지라도, 러스트의 테스팅 설비의 역학을 논의할 것입니다. 우리는 여러분이 테스트를 작성할 때 이용 가능한 어노테이션(annotation)과 매크로, 여러분의 테스트를 실행하기 위해 제공되는 기본 동작 및 옵션, 그리고 테스트들을 유닛(unit) 테스트와 통합(integration) 테스트로 조직화하는 방법에 대해 이야 기할 것입니다.

테스트를 작성하는 방법

테스트는 테스트 아닌 코드가 프로그램 내에서 기대했던 대로 기능을 하는지 검증하는 러스트 함수입니다. 테스트 함수의 본체는 통상적으로 다음의 세 가지 동작을 수행합니다:

1. 필요한 데이터 혹은 상태를 설정하기
2. 우리가 테스트하고 싶은 코드를 실행하기
3. 그 결과가 우리 예상대로인지 단언하기(`assert`)

이러한 동작을 하는 테스트 작성을 위해 러스트가 특별히 제공하는 기능들을 살펴봅시다. `test` 속성, 몇 가지 매크로, 그리고 `should_panic` 속성들을 포함해서 말이죠.

테스트 함수의 해부

가장 단순하게 말하면, 러스트 내의 테스트란 `test` 속성(attribute)이 주석으로 달려진 (annotated) 함수입니다. 속성은 러스트 코드 조각에 대한 메타데이터입니다: 한 가지 예로 5장에서 우리가 구조체와 함께 사용했던 `derive` 속성이 있습니다. 함수를 테스트 함수로 변경하기 위해서는, `fn` 전 라인에 `#[test]`를 추가합니다. `cargo test` 커맨드를 사용하여 테스트를 실행시키면, 러스트는 `test` 속성이 달려있는 함수들을 실행하고 각 테스트 함수가 성공 혹은 실패했는지를 보고하는 테스트 실행용 바이너리를 빌드할 것입니다.

7장에서 여러분이 카고를 통해 새로운 라이브러리 프로젝트를 만들었을 때, 테스트 함수를 갖고 있는 테스트 모듈이 자동으로 생성되는 것을 보았습니다. 이 모듈은 우리의 테스트를 작성하기 시작하도록 도움을 주는데, 즉 우리가 새로운 프로젝트를 시작할 때마다 매번 테스트 함수를 위한 추가적인 구조 및 문법을 찾아보지 않아도 되게 해 줍니다. 우리는 원하는 만큼 추가적인 테스트 함수들과 테스트 모듈들을 추가할 수 있습니다!

우리는 실제 코드를 테스팅하지는 않으면서 자동으로 만들어진 템플릿 테스트를 가지고 실험하는 식으로 테스트가 어떻게 동작하는지를 몇 가지 관점에서 탐구할 것입니다. 그리고 나서 우리가 작성한 몇몇 코드를 호출하고 동작이 정확한지를 확고히 하는 실제의 테스트를 작성해 볼 것입니다.

`adder`라고 하는 새로운 라이브러리 프로젝트를 만듭시다:

```
$ cargo new adder --lib
    Created library `adder` project
$ cd adder
```

여러분의 `adder` 라이브러리 내에 있는 `src/lib.rs` 파일의 내용물은 Listing 11-1과 같아야 합니다:

Filename: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Listing 11-1: `cargo new`를 이용하여 자동으로 생성된 테스트 모듈과 함수

지금은 제일 위의 두 줄은 무시하고 함수가 어떻게 작동하는지 알아보는데 집중합시다. `fn` 라인 전의 `# [test]` 어노테이션을 주목하세요: 이 속성이 바로 이것이 테스트 함수임을 나타내므로, 테스트 실행기는 이 함수를 테스트로 다루어야 한다는 것을 알게 됩니다. 또한 우리는 `tests` 모듈 내에 일반적인 시나리오를 셋업하거나 일반적인 연산을 수행하는 것을 돋기 위한 테스트 아닌 함수를 넣을 수 있으므로, 어떤 함수가 테스트 함수인지 `# [test]`를 이용하여 나타낼 필요가 있습니다.

이 함수의 본체는 $2 + 2$ 가 4와 같음을 단언하기 위해 `assert_eq!` 매크로를 사용합니다. 이 단언은 통상적인 테스트에 대한 형식 예제로서 제공됩니다. 실행하여 이 테스트가 통과되는지 확인해봅시다.

`cargo test` 커맨드는 Listing 11-2에서 보는 바와 같이 우리 프로젝트에 있는 모든 테스트를 실행합니다:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs
Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Listing 11-2: 자동으로 생성된 테스트를 실행한 결과

카고는 테스트를 컴파일하고 실행했습니다. `Compiling`, `Finished`, 그리고 `Running` 라인 이후에는 `running 1 test` 라인이 있습니다. 그다음 라인에는 생성된 테스트 함수의 이름인 `it_works`가 나타나고, 테스트의 실행 결과 `ok`가 나타납니다. 그리고 나서 테스트 실행의 전체 요약이 나타납니다. `test result: ok.` 는 모든 테스트가 통과했다는 뜻입니다. `1 passed; 0 failed`는 통과하거나 실패한 테스

트의 개수를 추가적으로 보여줍니다.

우리가 무시하라고 표시한 테스트가 없기 때문에, 요약문에 `0 ignored`라고 표시됩니다. 다음 절인 "테스트의 실행방식 제어하기"에서 테스트를 무시하는 것에 대해 다룰 것입니다.

`0 measured` 통계는 성능을 측정하는 벤치마크 테스트를 위한 것입니다. 벤치마크 테스트는 이 글이 쓰인 시점에서는 오직 나이틀리(nightly) 러스트에서만 사용 가능합니다. 나이틀리 러스트에 대한 더 많은 정보는 1장을 보세요.

`Doc-tests adder`로 시작하는 테스트 출력의 다음 부분은 문서 테스트의 결과를 보여주기 위한 것입니다. 아직 어떠한 문서 테스트도 없긴 하지만, 러스트는 우리의 API 문서 내에 나타난 어떠한 코드 예제라도 컴파일할 수 있습니다. 이 기능은 우리의 문서와 코드가 동기화를 유지하도록 돕습니다! 우리는 14장의 "문서 주석"절에서 문서 테스트를 작성하는 방법에 대해 이야기할 것입니다. 지금은 `Doc-tests` 출력을 무시할 것입니다.

우리의 테스트의 이름을 변경하고 테스트 출력이 어떻게 변하는지를 살펴봅시다. 다음과 같이 `it_works` 함수의 이름을 `exploration`으로 변경하세요:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

그리고 나서 `cargo test`를 다시 실행시킵니다. 이제 출력 부분에서 `it_works` 대신 `exploration`을 볼 수 있을 것입니다:

```
running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

다른 테스트를 추가해봅시다. 하지만 이번에는 실패하는 테스트를 만들 것입니다! 테스트 함수 내의 무언가가 패닉을 일으키면 테스트는 실패합니다. 각 테스트는 새로운 스레드 내에서 실행되며, 테스트 스레드가 죽은 것을 메인 스레드가 알게 되면, 테스트는 실패한 것으로 표시됩니다. 9장에서 패닉을 유발하는 가장 단순한 방법에 대해 이야기했습니다: 바로 `panic!` 매크로를 호출하는 것이죠! 새로운 테스트를 입력하여 여러분의 `src/lib.rs`가 Listing 11-3과 같은 모양이 되게 해 보세요:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

Listing 11-3: `panic!` 매크로를 호출하기 때문에 실패하게 될 두번째 테스트 추가

`cargo test`를 이용하여 다시 한번 테스트를 실행시키세요. 결과 출력은 Listing 11-4와 같이 나올 것인데, 이는 `exploration` 테스트는 통과하고 `another`는 실패했음을 보여줍니다:

```
running 2 tests
test tests::exploration ... ok
test tests::another ... FAILED

failures:

---- tests::another stdout ----
    thread 'tests::another' panicked at 'Make this test fail',
src/lib.rs:10:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

error: test failed
```

Listing 11-4: 한 테스트는 통과하고 다른 한 테스트는 실패할 때의 테스트 결과

`test tests::another` 라인은 `ok` 대신 `FAILED`을 보여줍니다. 개별 결과 부분과 요약 부분 사이에 새로운 두 개의 섹션이 나타납니다: 첫번째 섹션은 테스트 실패에 대한 구체적인 이유를 표시합니다. 이 경우, `another`는 `panicked at 'Make this test fail'` 때문에 실패했는데, 이는 `src/lib.rs`의 9번 라인에서 발생했습니다. 다음 섹션은 실패한 모든 테스트의 이름만 목록화한 것인데, 이는 테스트들이 많이 있고 구체적인 테스트 실패 출력이 많을 때 유용합니다. 실패하는 테스트의 이름은 이를 더 쉽게 디버깅하기 위해서 해당 테스트만을 실행시키는데 사용될 수 있습니다; "테스트의 실행방식 제어하기" 절에서 테스트를 실행

시키는 방법에 대한 더 많은 내용을 이야기할 것입니다.

요약 라인이 가장 마지막에 표시됩니다: 전체적으로, 우리의 테스트 결과는 **FAILED**입니다. 우리는 하나의 테스트에 통과했고 하나의 테스트에 실패했습니다.

이제 서로 다른 시나리오에서 테스트 결과가 어떻게 보이는지를 알았으니, **panic!** 외에 테스트 내에서 유용하게 쓰일 수 있는 몇 가지 매크로를 봅시다.

assert! 매크로를 이용하여 결과 확인하기

표준 라이브러리에서 제공하는 **assert!** 매크로는 여러분이 테스트가 어떤 조건이 **true**임을 보장하기를 원하는 경우 유용합니다. **assert!** 매크로에는 부울린 타입으로 계산되는 인자가 제공됩니다. 만일 값이 **true**라면 **assert!**는 아무일도 하지 않고 테스트는 통과됩니다. 만일 값이 **false**라면, **assert!**는 **panic!** 매크로를 호출하는데, 이것이 테스트를 실패하게 합니다. 이는 우리의 코드가 우리 의도대로 기능하고 있는지를 체크하는 것을 도와주는 매크로 중 하나입니다.

5장에 있는 Listing 5-9에서, **Rectangle** 구조체와 **can_hold** 메소드를 다루었는데, 여기 Listing 11-5에 다시 나왔습니다. 이 코드를 *src/lib.rs*에 넣고, **assert!** 매크로를 사용하여 테스트를 작성해봅시다.

Filename: *src/lib.rs*

```
#[derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

Listing 11-5: 5장의 **Rectangle** 구조체와 **can_hold** 메소드 이용하기

can_hold 메소드는 부울린 값을 반환하는데, 이는 **assert!** 매크로를 위한 완벽한 사용 사례라는 의미입니다! Listing 11-6에서는 길이 8에 너비 7인 **Rectangle** 인스턴스를 만들고, 이것이 길이 5에 너비 1인 다른 **Rectangle** 인스턴스를 포함할 수 있는지 단언(assert)해보는 것으로 **can_hold** 메소드를 시험하는 테스트를 작성합니다:

Filename: *src/lib.rs*

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(larger.can_hold(&smaller));
    }
}
```

Listing 11-6: 큰 사각형이 작은 사각형을 정말로 담을 수 있는지 검사하는 `can_hold`를 위한 테스트

`tests` 모듈 내에 새로운 라인이 추가된 것을 주목하세요: `use super::*;

tests 모듈은 우리가 7장에서 다루었던 보통의 가시성 규칙을 따르는 일반적인 모듈입니다. 우리가 내부 모듈 내에 있기 때문에, 외부 모듈에 있는 코드를 내부 모듈의 스코프로 가져올 필요가 있습니다. 여기서는 글롭(*)을 사용하기로 선택했고 따라서 우리가 외부 모듈에 정의한 어떠한 것이든 이 tests 모듈에서 사용 가능합니다.`

우리의 테스트는 `larger_can_hold_smaller`로 명명되었고, 요구된 바와 같이 `Rectangle` 인스턴스를 두 개 생성했습니다. 그 뒤 `assert!` 매크로를 호출하고 `larger.can_hold(&smaller)` 호출의 결과값을 인자로서 넘겼습니다. 이 표현식은 `true`를 반환할 예정이므로, 우리의 테스트는 통과해야 합니다. 자, 이제 알아봅시다!

```
running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

통과되었군요! 이번에는 작은 사각형이 큰 사각형을 포함시킬수 없음을 단언하는 또 다른 테스트를 추가합시다:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(!smaller.can_hold(&larger));
    }
}
```

이 경우 `can_hold` 함수의 올바른 결과값은 `false`이므로, `assert!` 매크로에게 넘기기 전에 이 결과를 반대로 만들 필요가 있습니다. 결과적으로, 우리의 테스트는 `can_hold`가 `false`를 반환할 경우에만 통과할 것입니다:

```
running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

통과하는 테스트가 두 개가 되었습니다! 이제는 만약 우리의 코드에 버그가 있을 때는 테스트 결과가 어찌되는지 봅시다. `can_hold` 메소드의 구현 부분 중 큰(>) 부등호를 이용해 길이를 비교하는 부분을 작은(<) 부등호로 바꿔봅시다:

```
// --snip--

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length < other.length && self.width > other.width
    }
}
```

테스트를 실행시키면 이제 아래와 같이 출력됩니다:

```

running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... FAILED

failures:

---- tests::larger_can_hold_smaller stdout ----
    thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed:
      larger.can_hold(&smaller)', src/lib.rs:22:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

```

우리의 테스트가 버그를 찾았습니다! `larger.length`는 80이고 `smaller.length`는 50이므로, `can_hold`의 길이 부분에 대한 비교값은 이제 `false`를 반환합니다: 80이 50보다 작지 않으니까요.

`assert_eq!`와 `assert_ne!`를 이용한 동치(equality) 테스트

기능성을 테스트하는 일반적인 방법은 테스트 내의 코드의 결과값과 우리가 기대하는 값을 비교하여 둘이 서로 같은지를 확실히 하는 것입니다. 이를 `assert!` 매크로에 `==`를 이용한 표현식을 넘기는 식으로 할 수도 있습니다. 그러나 이러한 테스트를 더 편리하게 수행해주는 표준 라이브러리가 제공하는 한 쌍의 매크로 - `assert_eq!` 와 `assert_ne!` - 가 있습니다. 이 매크로들은 각각 동치(equality)와 부동(inequality)을 위해 두 인자를 비교합니다. 또한 이들은 만일 단언에 실패한다면 두 값을 출력해 주는데, 이는 왜 테스트가 실패했는지를 포기 더 쉬워집니다; 반면, `assert!`는 `==` 표현식에 대해 `false` 값을 얻었음을 가리킬 뿐, 어떤 값이 `false` 값을 야기했는지는 알려주지 않습니다.

Listing 11-7와 같이, 파라미터에 `2`를 더하여 결과를 반환하는 `add_two` 함수를 작성합시다. 그 후 `assert_eq!` 매크로를 이용하여 이 함수를 테스트하겠습니다.

Filename: `src/lib.rs`

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

Listing 11-7: `assert_eq!` 매크로를 이용하는 `add_two` 함수 테스트

이게 통과하는지 확인해 봅시다!

```
running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

`assert_eq!` 매크로에 제공한 첫번째 인자 4는 `add_two(2)` 호출의 결과와 동일합니다. 이 테스트에 대한 라인은 `test tests::it_adds_two ... ok`이고, `ok` 문자열은 테스트가 통과했음을 나타냅니다!

`assert_eq!`를 이용하는 테스트가 실패했을 때는 어떻게 보이는지를 알아보기 위해 테스트에 버그를 집어 넣어 봅시다. `add_two` 함수에 3을 대신 더하는 형태로 구현을 변경해 보세요:

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

테스트를 다시 실행해 보세요:

```

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
    thread 'tests::it_adds_two' panicked at 'assertion failed: `(left ==
right)`
  left: `4`,
  right: `5`', src/lib.rs:11:8

failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

```

우리의 테스트가 버그를 잡았어요! `it_adds_two` 테스트는 `assertion failed: `(left == right)``라는 메세지와 `left`는 4였고 `right`는 5였다는 것으로 보여줌과 함께 실패했습니다. 이 메세지는 우리가 디버깅을 시작하는데 유용한 도움을 줍니다: `assert_eq!`의 `left` 인자는 4였는데, `add_two(2)`를 넣은 `right` 인자는 5라고 말해주고 있습니다.

몇몇 언어와 테스트 프레임워크 내에서는, 두 값이 같은지를 단언하는 함수의 파라미터를 `expected`와 `actual`로 부르며, 우리가 인자를 넣는 순서가 중요하다는 점을 기억하세요. 하지만 러스트에서는 그 대신 `left`와 `right`라고 불리며 우리가 기대한 값과 테스트 내의 코드가 생성하는 값을 지정하는 순서는 중요치 않습니다. 이 테스트의 단언을 `assert_eq!(add_two(2), 4)`로 작성할 수도 있는데, 이는 `assertion failed: `(left == right)``와 `left`는 5고 `right`는 4라는 실패 메세지를 만들어 낼 것입니다.

`assert_ne!` 매크로는 우리가 제공한 두 개의 값이 서로 갖지 않으면 통과하고 동일하면 실패할 것입니다. 이 매크로는 어떤 값이 될 것인지는 정확히 확신하지 못하지만, 어떤 값이라면 절대로 될 수 없는지는 알고 있을 경우에 가장 유용합니다. 예를 들면, 만일 어떤 함수가 입력값을 어떤 방식으로든 변경한다는 것을 보장하지만, 그 입력값이 우리가 테스트를 실행한 요일에 따라 달라지는 형태라면, 단언을 하는 가장 좋은 방법은 함수의 결괏값이 입력값과 같지 않다는 것일지도 모릅니다.

표면 아래에서, `assert_eq!`와 `assert_ne!` 매크로는 각각 `==`과 `!=` 연산자를 이용합니다. 단언에 실패하면, 이 매크로들은 디버그 포맷팅을 사용하여 인자들을 출력하는데, 이는 비교되는 값들이 `PartialEq`와 `Debug` 트레이트를 구현해야 한다는 의미입니다. 모든 기본 타입과 표준 라이브러리가 제공하는 대부분의 타입들은 이 트레이트들을 구현하고 있습니다. 여러분이 정의한 구조체나 열거형에 대해서, 해당 타입의 값이 서로 같은지 혹은 다른지를 단언하기 위해서는 `PartialEq`를 구현할 필요가 있습니다. 단언에 실패할 경우에 값을 출력하기 위해서는 `Debug`를 구현해야 합니다. 5장에서 설명한 바와 같이 두 트레이트 모두 추론 가능한(derivable) 트레이트이기 때문에, 이 트레이트의 구현은 보통 `#[derive(PartialEq, Debug)]` 어노테이션을 여러분의 구조체나 열거형 정의부에 추가하는 정도로 간단합니다. 이에 대한 것과 다른 추론 가능한

트레이에 대한 더 자세한 내용은 부록 C를 참고하세요.

커스텀 실패 메세지 추가하기

또한 우리는 `assert!`, `assert_eq!` 및 `assert_ne!` 매크로의 추가 인자로서 커스텀 메세지를 입력하여 실패 메세지와 함께 출력되도록 할 수 있습니다. `assert!`가 요구하는 하나의 인자 후에 지정된 인자들이나 `assert_eq!`와 `assert_ne!`가 요구하는 두 개의 인자 후에 지정된 인자들은 우리가 8장의 “`+` 연산자나 `format!` 매크로를 이용한 접합”절에서 다루었던 `format!` 매크로에 넘겨지므로, 여러분은 `{}` 변경자 (placeholder)를 갖는 포맷 스트링과 이 변경자에 입력될 값들을 넘길 수 있습니다. 커스텀 메세지는 해당 단언의 의미를 문서화하기 위한 용도로서 유용하므로, 테스트가 실패했을 때, 코드에 어떤 문제가 있는지에 대해 더 좋은 생각을 가질 수 있습니다.

예를 들어, 이름을 부르며 사람들을 환영하는 함수가 있고, 이 함수에 넘겨주는 이름이 출력 내에 있는지 테스트하고 싶다고 칩니다:

Filename: src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

여기서 이 프로그램의 요구사항은 아직 합의되지 않았고, 인사말의 시작 지점에 있는 `Hello` 텍스트가 변경될 것이라는 점이 꽤나 확실한 상태라고 칩니다. 우리는 그런 변경사항이 생기더라도 이름에 대한 테스트를 갱신할 필요는 없다고 결정했고, 따라서 `greeting` 함수로부터 반환된 값과 정확히 일치하는 체크 대신, 출력 값이 입력 파라미터의 텍스트를 포함하고 있는지만 단언할 것입니다.

`greeting`이 `name`을 포함하지 않도록 변경하는 것으로 버그를 집어넣어 테스트 실패가 어떻게 보이는지 살펴봅시다:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

이 테스트를 수행하면 다음을 출력합니다:

```
running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at 'assertion failed:
result.contains("Carol")', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
tests::greeting_contains_name
```

이 결과는 그저 단언이 실패했으며 몇 번째 줄의 단언이 실패했는지만을 나타냅니다. 이 경우에서 더 유용한 실패 메세지는 `greeting` 함수로부터 얻은 값을 출력하는 것일 테지요. 테스트 함수를 바꿔서 `greeting` 함수로부터 얻은 실제 값으로 채워질 변경자를 이용한 포맷 스트링으로부터 만들어지는 커스텀 실패 메세지를 줄 수 있도록 해봅시다:

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`, result"
    );
}
```

이제 테스트를 다시 실행시키면, 더 많은 정보를 가진 에러 메세지를 얻을 것입니다:

```
---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at 'Greeting did not
contain
name, value was `Hello!`', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

이제 실제로 테스트 출력에서 얻어진 값을 볼 수 있고, 이는 우리가 기대했던 일 대신 실제 어떤 일이 일어났는지 디버깅하는데 도움을 줄 것입니다.

should_panic을 이용한 패닉에 대한 체크

우리의 코드가 우리가 기대한 정확한 값을 반환하는 것을 체크하는 것에 더하여, 우리의 코드가 우리가 기대한 대로 에러가 나는 경우를 처리할 수 있는지 체크하는 것 또한 중요합니다. 예를 들어, 9장의 Listing 9-9에서 우리가 만들었던 `Guess` 타입을 떠올려보세요. `Guess`를 이용하는 다른 코드는 `Guess` 인스턴스가 1과 100 사이의 값만 가질 것이라는 보장에 의존적입니다. 우리는 범위 밖의 값으로 `Guess` 인스턴스를 만드는 시도가 패닉을 일으킨다는 것을 확실히 하는 테스트를 작성할 수 있습니다.

이는 또 다른 속성인 `should_panic`를 테스트 함수에 추가함으로써 할 수 있습니다. 이 속성은 함수 내의 코드가 패닉을 일으키면 테스트가 통과하도록 만들어줍니다; 함수 내의 코드가 패닉을 일으키지 않는다면 테스트는 실패할 것입니다.

Listing 11-8은 `Guess::new`의 에러 조건이 우리 예상대로 발동되는지를 검사하는 테스트를 보여줍니다:

Filename: src/lib.rs

```
pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Listing 11-8: 어떤 조건이 `panic!`을 일으키는지에 대한 테스트

`#[should_panic]` 속성이 `#[test]` 속성 뒤, 그리고 적용될 테스트 함수 앞에 붙었습니다. 이 테스트가

통과될 때의 결과를 봅시다:

```
running 1 test
test tests::greater_than_100 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

좋아 보이는군요! 이제 `new` 함수가 100 이상의 값일 때 패닉을 발생시키는 조건을 제거함으로써 코드에 버그를 넣어봅시다:

```
impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }
}
```

Listing 11-8의 테스트를 실행시키면, 아래와 같이 실패할 것입니다:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:
    failures:
        tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out
```

이 경우에는 그다지 쓸모 있는 메세지를 얻지 못하지만, 한번 테스트 함수를 살펴보게 되면, 함수가 `# [should_panic]`으로 어노테이션 되었다는 것을 볼 수 있습니다. 우리가 얻은 실패는 함수 내의 코드가 패닉을 일으키지 않았다는 의미가 됩니다.

`should_panic` 테스트는 애매할 수 있는데, 그 이유는 이 속성이 단지 코드에서 어떤 패닉이 유발되었음만을 알려줄 뿐이기 때문입니다. `should_panic` 테스트는 일어날 것으로 예상한 것 외의 다른 이유로 인한 패닉이 일어날 지라도 통과할 것입니다. `should_panic` 테스트를 더 엄밀하게 만들기 위해서, `should_panic` 속성에 `expected` 파라미터를 추가할 수 있습니다. 이 테스트 도구는 실패 메세지가 제공된 텍스트를 담고 있는지 확실히 할 것입니다. 예를 들면, Listing 11-9와 같이 입력된 값이 너무 작거나 혹은

너무 클 경우에 대해 서로 다른 메세지를 가진 패닉을 일으키는 `new` 함수를 갖고 있는 수정된 `Guess` 코드를 고려해봅시다:

Filename: src/lib.rs

```
// --snip

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 {
            panic!("Guess value must be greater than or equal to 1, got {}.", value);
        } else if value > 100 {
            panic!("Guess value must be less than or equal to 100, got {}.", value);
        }
        Guess {
            value
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Listing 11-9: 어떤 조건이 특정 패닉 메세지를 가진 `panic!`을 일으키는 테스트

이 테스트는 통과할 것인데, 그 이유는 `should_panic` 속성에 추가한 `expected` 파라미터 값이 `Guess::new` 함수가 패닉을 일으킬 때의 메세지의 서브 스트링이기 때문입니다. 우리가 예상하는 전체 패닉 메세지로 특정할 수도 있는데, 그러한 경우에는 `Guess value must be less than or equal to 100, got 200.`이 되겠지요. 여러분이 `should_panic`에 대한 기대하는 파라미터를 특정하는 것은 패닉 메세지가 얼마나 유일한지 혹은 유동적인지, 그리고 여러분의 테스트가 얼마나 정확하기를 원하는지에 따라서 달라집니다. 위의 경우, 패닉 메세지의 서브 스트링은 실행된 함수의 코드가 `else if value > 100` 경우에 해당함을 확신하기에 충분합니다.

`expect` 메세지를 가진 `should_panic` 테스트가 실패하면 어떻게 되는지 보기 위해서, 다시 한번 `if`

`value < 1` 아래 코드 블록과 `else if value > 100` 아래 코드 블록을 바꿔서 버그를 만들어봅시다:

```
if value < 1 {
    panic!("Guess value must be less than or equal to 100, got {}.", value);
} else if value > 100 {
    panic!("Guess value must be greater than or equal to 1, got {}.", value);
}
```

이번에는 `should_panic` 테스트를 실행하면, 아래와 같이 실패합니다:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
        thread 'tests::greater_than_100' panicked at 'Guess value must be
greater than or equal to 1, got 200.', src/lib.rs:11:12
note: Run with `RUST_BACKTRACE=1` for a backtrace.
note: Panic did not include expected string 'Guess value must be less than or
equal to 100'

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out
```

실패 메세지는 이 테스트가 우리 예상에 맞게 실제로 패닉에 빠지기는 했으나, 패닉 메세지가 예상하는 스트링을 포함하지 않고 있다고 말하고 있습니다 (`did not include expected string 'Guess value must be less than or equal to 100'`). 우리가 얻어낸 패닉 메세지를 볼 수 있는데, 이 경우에는 `Guess value must be greater than or equal to 1, got 200.` 이었습니다. 그러면 우리는 어디에 우리의 버그가 있는지를 찾아내기 시작할 수 있습니다!

이제까지 테스트를 작성하는 몇 가지 방법을 알게 되었으니, 우리의 테스트를 실행할 때 어떤 일이 벌어지는지를 살펴보고 `cargo test` 와 함께 사용할 수 있는 어려가지 옵션들에 대해서 탐구해봅시다.

테스트의 실행 방식 제어하기

`cargo run`이 여러분의 코드를 컴파일하고 난 뒤 그 결과인 바이너리를 실행하는 것과 마찬가지로, `cargo test`는 여러분의 코드를 테스트 모드에서 컴파일하고 결과로 발생한 테스트 바이너리를 실행합니다. 여러분은 커맨드 라인 옵션을 지정하여 `cargo test`의 기본 동작을 변경할 수 있습니다. 예를 들어, `cargo test`를 통해 생성된 바이너리의 기본 동작은 모든 테스트를 병렬적으로 수행하고 테스트가 실행되는 동안 생성된 결과를 캡처하는 것으로, 테스트 결과와 연관된 출력을 읽기 쉽도록 화면에 표시되는 것을 막아버립니다.

어떤 커맨드 라인 옵션은 `cargo test`에 입력되고 어떤 옵션은 결과 테스트 바이너리에 입력됩니다. 이 두 가지 타입의 인자를 구분하기 위해서, `cargo test`에 주어질 인자를 먼저 나열하고, 그다음 구분자 (separator)로 `--`를 넣고, 그 뒤 테스트 바이너리에 입력될 인자를 나열합니다. `cargo test --help`를 실행하는 것은 `cargo test`에서 사용할 수 있는 옵션을 표시하고, `cargo test -- --help`를 실행하는 것은 구분자 `--` 이후에 나올 수 있는 옵션을 표시합니다.

테스트를 병렬 혹은 연속으로 실행하기

여러 개의 테스트를 실행할 때는, 기본적으로 스레드를 이용하여 병렬적으로 수행됩니다. 이는 테스트가 더 빠르게 실행되어 끝낼 수 있다는 의미이므로, 우리의 코드가 잘 동작하는지 혹은 그렇지 않은지에 대한 피드백을 더 빨리 얻을 수 있습니다. 테스트가 동시에 실행되므로, 여러분의 테스트가 서로 다른 테스트 혹은 공유 상태 값에 의존하지 않는지 주의해야 하는데, 이는 이를테면 현재 작업 디렉토리나 환경 변수와 같은 공유 환경 값을 포함합니다.

예를 들면, 여러분이 작성한 테스트 각각이 `test-output.txt`라는 파일을 디스크에 만들고 이 파일에 어떤 데이터를 쓰는 코드를 실행한다고 가정해봅시다. 그런 다음 각 테스트는 그 파일로부터 데이터를 읽고, 이 파일이 특정한 값을 담고 있는지 단언하는데, 이 값들은 테스트마다 다릅니다. 모든 테스트들이 동시에 실행되기 때문에, 어떤 테스트가 파일을 쓰고 읽는 동안 다른 테스트가 파일을 덮어쓸지도 모릅니다. 두 번째 테스트는 실패할 것인데, 이는 코드가 정확히 않아서가 아니라 테스트들이 병렬적으로 실행하는 동안 서로에게 간섭을 일으켰기 때문입니다. 한 가지 해결책은 각 테스트가 서로 다른 파일을 쓰도록 확실히 하는 것입니다; 또 다른 해결책은 테스트를 한 번에 하나씩만 실행하는 것입니다.

만일 여러분이 테스트들을 병렬적으로 실행하고 싶지 않을 경우, 혹은 여러분이 사용되는 스레드의 개수에 대한 더 정밀한 제어를 하고 싶을 경우, 여러분은 `--test-threads` 플리그와 테스트 바이너리에서 사용하고 싶은 스레드 개수를 넘길 수 있습니다. 다음 예제를 봅시다:

```
$ cargo test -- --test-threads=1
```

여기서는 테스트 스레드의 개수에 1을 지정했는데, 이는 프로그램이 어떠한 병렬 처리도 사용하지 않음을 애기해줍니다. 테스트를 하나의 스레드에서 실행하는 것은 병렬로 수행하는 것에 비해 시간이 더 오래 걸리겠지

만, 테스트들이 어떤 상태를 공유할 경우 서로가 간섭할 가능성이 없어질 것입니다.

함수 결과 보여주기

기본적으로 어떤 테스트가 통과하면, 러스트의 테스트 라이브러리는 표준 출력(standard output)으로 출력되는 어떤 것이든 캡처합니다. 예를 들면, 우리가 테스트 내에서 `println!`을 호출하고 이 테스트가 통과하면, `println!` 출력을 터미널에서 볼 수 없습니다: 우리는 오직 그 테스트가 통과되었다고 표시된 라인만 볼 뿐입니다. 만일 테스트가 실패하면, 실패 메세지 아래에 표준 출력으로 출력되었던 어떤 것이든 보게 될 것입니다.

예를 들어, Listing 11-10은 파라미터의 값을 출력한 뒤 10을 반환하는 바보 같은 함수를 보여주고 있습니다. 그리고 통과하는 테스트와 실패하는 테스트를 갖추고 있습니다:

Filename: src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

Listing 11-10: `println!`을 호출하는 함수를 위한 테스트

`cargo test`를 이용하여 이 테스트를 실행했을 때 보게 될 출력은 다음과 같습니다:

```
running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ----
    I got the value 8
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left ==
right)`
  left: `5`,
  right: `10`', src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out
```

I got the value 4라는 메세지를 어디에서도 볼 수 없는데, 이는 성공하는 테스트가 실행시키는 출력이라는 점을 주목하세요. 이 출력 메세지는 캡처되었습니다. 실패한 테스트로부터 얻어진 출력 메세지인 I got the value 8은 테스트 정리 출력 부분에 나타나는데, 이는 테스트 실패 원인 또한 함께 보여줍니다.

만일 성공하는 테스트에 대한 출력 값 또한 볼 수 있기를 원한다면, --nocapture 플래그를 이용하여 출력 캡처 동작을 비활성화시킬 수 있습니다:

```
$ cargo test -- --nocapture
```

Listing 11-10의 테스트를 --nocapture 플래그와 함께 실행시키면 다음과 같이 나옵니다:

```
running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left ==
right)`
  left: `5`,
  right: `10`, src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
test tests::this_test_will_fail ... FAILED

failures:

failures:
  tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out
```

테스트에서의 출력과 테스트 결과 출력이 분리된 점을 주목하세요; 이는 우리가 이전 절에서 다룬 내용처럼 테스트가 병렬적으로 수행되기 때문입니다. `--test-threads=1` 옵션과 `--nocapture` 기능을 동시에 시도하고 출력이 어떻게 바뀌는지를 확인해 보세요!

이름으로 테스트의 일부분만 실행하기

가끔, 모든 테스트 셋을 실행하는 것은 시간이 오래 걸릴 수 있습니다. 만일 여러분이 특정 영역의 코드에 대해서 작업하고 있다면, 그 코드와 연관된 테스트만 실행시키고 싶어 할 수도 있습니다. 여러분은 `cargo test`에 여러분이 실행시키고 싶어 하는 테스트(들)의 이름들을 인자로 넘김으로써 어떤 테스트들을 실행시킬지 고를 수 있습니다.

테스트의 일부분만을 실행시키는 법을 보여드리기 위해서, Listing 11-11에서 보시는 바와 같이 `add_two` 함수를 위한 세 개의 테스트를 만들어서 하나만 골라 실행해보겠습니다:

Filename: `src/lib.rs`

```

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}

```

Listing 11-11: 여러 이름으로 된 세 가지 테스트

만일 테스트를 어떠한 인자 없이 실행시키면, 전에 본 것과 같이 모든 테스트가 병렬적으로 수행될 것입니다:

```

running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

단일 테스트 실행하기

단 하나의 테스트만 실행시키기 위해 `cargo test`에 그 테스트 함수의 이름을 넘길 수 있습니다:

```
$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out
```

`one_hundred`라는 이름의 테스트만 실행되었습니다; 다른 두 개의 테스트는 이 이름에 맞지 않습니다. 테스트 출력은 정리 라인의 끝에 `2 filtered out`이라고 표시함으로써 이 커맨드로 지정한 것보다 많은 테스트를 가지고 있음을 우리에게 알려줍니다.

이 방법으로는 여러 테스트의 이름들을 특정할 수는 없고, `cargo test`에 주어진 제일 첫 번째 값만 이용될 것입니다.

여러 개의 테스트를 실행시키기 위한 필터링

우리는 테스트 이름의 일부분을 특정할 수 있고, 해당 값과 일치하는 이름의 테스트가 실행될 것입니다. 예를 들면, 우리의 테스트 이름들 중에서 두 개가 `add`를 포함하므로, `cargo test add`라고 실행하여 이 두 개의 테스트를 실행시킬 수 있습니다:

```
$ cargo test add
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

이는 `add`가 이름에 포함된 모든 테스트를 실행시켰고 `one_hundred`라는 이름의 테스트를 걸러냈습니다. 또한 테스트가 있는 모듈이 테스트의 이름의 일부가 되어 있으므로, 모듈의 이름으로 필터링하여 그 모듈 내의 모든 테스트를 실행시킬 수 있다는 점도 주목하세요.

특별한 요청이 없는 한 몇몇 테스트들 무시하기

이따금씩 몇몇 특정 테스트들은 실행하는데 너무나 시간이 많이 소모될 수 있어서, 여러분은 `cargo test`의 실행 시 이 테스트들을 배제하고 싶어 할지도 모릅니다. 여러분이 실행시키고자 하는 모든 테스트들을 인자로서 열거하는 것 대신, 다음과 같이 시간이 많이 걸리는 테스트들에 `ignore` 속성을 어노테이션하여 이

들을 배제시킬 수 있습니다:

Filename: src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

배제시키고 싶은 테스트에 대하여 `#[test]` 다음 줄에 `#[ignore]`를 추가하였습니다. 이제 우리의 테스트들을 실행시키면, `it_works`가 실행되는 것은 보이지만, `expensive-test`는 실행되지 않는 것을 볼 수 있습니다:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
Running target/debug/deps/adder-ce99bcc2479f4607

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out
```

`expensive_test`는 `ignored` 리스트에 포함되었습니다. 만일 무시된 테스트들만 실행시키고 싶다면, `cargo test -- --ignored`라고 실행함으로써 이를 요청할 수 있습니다.

```
$ cargo test -- --ignored
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

어떠한 테스트를 실행시킬지를 제어함으로써, 여러분은 `cargo test`의 결과가 빠르게 나오도록 확실히 할 수 있습니다. `ignored` 테스트들의 결과를 확인하기에 타당한 시점에 있고 해당 결과를 기다릴 시간을 가지고 있을 때, 여러분은 대신 `cargo test -- --ignored`를 실행시킬 수 있습니다.

테스트 조직화

이 장의 시작 부분에서 언급했듯이, 테스팅은 복잡한 분야이고, 여러 사람들이 서로 다른 용어와 조직화 방식을 이용합니다. 러스트 커뮤니티에서는 테스트에 대해서 두 개의 주요한 카테고리로 나눠 생각합니다: 단위 테스트(*unit test*) 그리고 *통합 테스트(integration test)*입니다. 단위 테스트는 작고 하나에 더 집중하며, 한 번에 하나의 모듈만 분리하여 테스트하고, 비공개 인터페이스 (private interface)를 테스트합니다. 통합 테스트는 완전히 여러분의 라이브러리 외부에 있으며, 공개 인터페이스 (public interface)를 이용하고 테스트마다 여러 개의 모듈을 잠재적으로 실험함으로써, 다른 외부의 코드가 하는 방식과 동일한 형태로 여러분의 코드를 이용합니다.

두 종류의 테스트 작성 모두가 여러분의 라이브러리 코드 조각들이 따로따로 혹은 함께 사용되었을 때 여러분이 기대하는 바와 같이 작동하는지를 확신시키는데 중요합니다.

단위 테스트

단위 테스트의 목적은 각 코드의 단위를 나머지 부분과 분리하여 테스트하는 것인데, 이는 코드가 어디 있고 어느 부분이 기대한 대로 동작하지 않는지를 빠르게 정확히 찾아낼 수 있도록 하기 위함입니다. 단위 테스트는 `src` 디렉토리 내에 넣는데, 각 파일마다 테스트하는 코드를 담고 있습니다. 관례는 각 파일마다 테스트 함수를 담고 있는 `tests`라는 이름의 모듈을 만들고, 이 모듈에 `cfg(test)`라고 어노테이션 하는 것입니다.

테스트 모듈과 `#[cfg(test)]`

테스트 모듈 상의 `#[cfg(test)]` 어노테이션은 러스트에게 우리가 `cargo build`를 실행시킬 때가 아니라 `cargo test`를 실행시킬 때에만 컴파일하고 실행시키라고 말해줍니다. 이는 우리가 오직 라이브러리만 빌드하고 싶을 때 컴파일 시간을 절약시켜주고, 테스트가 포함되어 있지 않으므로 컴파일 결과물의 크기를 줄여줍니다. 통합 테스트는 다른 디렉토리에 위치하기 때문에, 여기에는 `#[cfg(test)]` 어노테이션이 필요치 않음을 앞으로 보게 될 것입니다. 하지만, 단위 테스트가 해당 코드와 동일한 파일에 위치하기 때문에, `#[cfg(test)]`를 사용하여 컴파일 결과물에 이들이 포함되지 않아야 함을 특정합니다.

이 장의 첫 번째 절에서 새로운 `adder` 프로젝트를 생성했을 때, 카고가 우리를 위하여 아래와 같은 코드를 생성했던 것을 상기하세요:

Filename: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

이 코드는 자동으로 생성되는 테스트 모듈입니다. `cfg` 속성은 환경 설정(*configuration*)을 의미하며, 러스트에게 뒤따르는 아이템이 특정한 환경 값에 대해서만 포함되어야 함을 말해줍니다. 위의 경우, 환경 값이 `test` 인데, 테스트를 컴파일하고 실행하기 위해 러스트로부터 제공되는 것입니다. 이 속성을 이용함으로써, 카고는 우리가 능동적으로 `cargo test`를 이용해서 테스트를 실행시킬 경우에만 우리의 테스트 코드를 컴파일합니다. 이는 이 모듈 내에 있을지도 모를 어떠한 헬퍼 함수들, 추가적으로 `#[test]` 라고 어노테이션된 함수들을 포함합니다.

비공개 함수 테스트하기

테스팅 커뮤니티 내에서 비공개 함수가 직접적으로 테스트되어야 하는지 혹은 그렇지 않은지에 대한 논쟁이 있었고, 다른 언어들은 비공개 함수를 테스트하는 것이 어렵거나 불가능하게 만들어두었습니다. 여러분이 어떤 테스트 이데올로기를 고수하는지와는 상관없이, 러스트의 비공개 규칙은 여러분이 비공개 함수를 테스트하도록 허용해줍니다. 비공개 함수 `internal_adder` 가 있는 Listing 11-12 내의 코드를 고려해 보시죠:

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

Listing 11-12: 비공개 함수 테스트하기

`internal_adder` 함수는 `pub`으로 표시되어 있지 않지만, 테스트가 그저 러스트 코드일 뿐이고 `tests` 모듈도 그냥 또 다른 모듈이기 때문에, `internal_adder`를 불러들여 호출하는 것이 그냥 되는 것을 주목하세요. 만약 여러분이 비공개 함수를 테스트해야 한다고 생각하지 않는다면, 러스트에서는 여러분이 그렇게 하도록 강제할 일은 없습니다.

통합 테스트

러스트에서 통합 테스트들은 완전히 여러분의 라이브러리 외부에 있습니다. 이들은 여러분의 라이브러리를 다른 코드들과 동일한 방식으로 이용하는데, 이는 이 외부 테스트들이 오직 여러분의 라이브러리의 공개 API 부분에 속하는 함수들만 호출할 수 있다는 의미입니다. 이들의 목적은 여러분의 라이브러리의 수많은 파트들이 함께 올바르게 동작하는지를 시험하는 것입니다. 그 자체로서는 올바르게 동작하는 코드의 단위들도 통합되었을 때는 문제를 일으킬 수 있으므로, 통합된 코드의 테스트 커버율 또한 중요합니다. 통합 테스트를 만들기 위해서는 `tests` 디렉토리를 먼저 만들 필요가 있습니다.

tests 디렉토리

프로젝트 디렉토리의 최상위, 그러니까 `src` 옆에 `tests` 디렉토리를 만듭니다. 카고는 이 디렉토리 내의 통합 테스트 파일들을 찾을 줄 압니다. 그런 후에는 이 디렉토리에 원하는 만큼 많은 테스트 파일을 만들 수 있으며, 카고는 각각의 파일들을 개별적인 크레이트처럼 컴파일할 것입니다.

한 번 통합 테스트를 만들어봅시다. Listing 11-12의 `src/lib.rs` 코드를 그대로 유지한 채로, `tests` 디렉토리를 만들고, `tests/integration_test.rs`라는 이름의 새 파일을 만든 다음, Listing 11-13의 코드를 집어넣으세요.

Filename: `tests/integration_test.rs`

```
extern crate adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Listing 11-13: `adder` 크레이트 내의 함수에 대한 통합 테스트

코드의 상단에 `extern crate adder`를 추가했는데, 이는 단위 테스트에서는 필요 없었지요. 이는 `tests` 디렉토리 내의 각 테스트가 모두 개별적인 크레이트이라서, 우리의 라이브러리를 각각에 가져올 필요가 있기 때문입니다.

`tests/integration_test.rs`에는 `#[cfg(test)]`를 이용한 어노테이션을 해줄 필요가 없습니다. 카고는

`test` 디렉토리를 특별 취급하여 `cargo test`를 실행시켰을 때에만 이 디렉토리 내의 파일들을 컴파일합니다. 이제 `cargo test` 실행을 시도해봅시다:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running target/debug/deps/adder-abcabcbc

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

출력에 세 개의 섹션이 생겼습니다: 단위 테스트, 통합 테스트, 그리고 문서 테스트입니다. 단위 테스트를 위한 첫 번째 섹션은 우리가 봐오던 것과 동일합니다: 각각의 단위 테스트마다 한 라인 (Listing 11-12에서 우리가 추가한 `intenal`이라는 이름의 것이 있었죠), 그다음 단위 테스트들의 정리 라인이 있습니다.

통합 테스트 섹션은 `Running target/debug/deps/integration-test-ce99bcc2479f4607`이라고 말하는 라인과 함께 시작합니다 (여러분의 출력 값 끝의 해쉬값은 다를 것입니다). 그다음 이 통합 테스트 안의 각 테스트 함수를 위한 라인이 있고, `Doc-tests adder` 섹션이 시작되기 직전에 통합 테스트의 결과를 위한 정리 라인이 있습니다.

어떠한 `src` 파일에 단위 테스트 함수를 더 추가하는 것이 단위 테스트 섹션의 테스트 결과 라인을 더 늘린다는 점을 상기하세요. 통합 테스트 파일에 테스트 함수를 더 추가하는 것은 그 파일의 섹션의 라인을 더 늘릴 것입니다. 각 통합 테스트 파일은 고유의 섹션을 가지고 있으므로, 만일 우리가 `tests` 디렉토리에 파일을 더 추가하면, 통합 테스트 섹션이 더 생길 것입니다.

`cargo test`의 인자로서 테스트 함수의 이름을 명시하는 식으로 특정 통합 테스트 함수를 실행시키는 것도 여전히 가능합니다. 특정한 통합 테스트 파일 내의 모든 테스트를 실행시키기 위해서는, `cargo test`에 파일 이름을 뒤에 붙인 `--test` 인자를 사용하세요:

```
$ cargo test --test integration_test
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

이 커맨드는 *tests/integration_test.rs* 내의 테스트만 실행합니다.

통합 테스트 내의 서브모듈

더 많은 통합 테스트를 추가하게 되면, 이들을 조직화하기 쉽도록 *tests* 디렉토리 내에 하나 이상의 파일을 만들고 싶어 할지도 모릅니다; 예를 들면, 여러분은 이들이 테스트하는 기능별로 테스트 함수들을 묶을 수 있습니다. 앞서 언급했듯이, *tests* 디렉토리 내의 각 파일은 고유의 개별적인 크레이트인 것처럼 컴파일됩니다.

각 통합 테스트 파일을 고유한 크레이트인 것처럼 다루는 것은 여러분의 크레이트를 이용하게 될 사용자들의 방식과 더 유사하게 분리된 스코프를 만들어 내기에 유용합니다. 하지만, 이는 *src* 내의 파일들이 동일한 동작을 공유하는 것을 *tests* 디렉토리 내의 파일들에서는 할 수 없음을 의미하는데, 이는 여러분이 7장에서 코드를 모듈과 파일로 나누는 법에 대해 배웠던 것입니다.

만일 여러분이 여러 개의 통합 테스트 파일들 내에서 유용하게 사용될 헬퍼 함수들 묶음을 가지고 있으며, 이들을 공통 모듈로 추출하기 위해 7장의 "모듈을 다른 파일로 옮기기"절에 있는 단계를 따르는 시도를 한다면, 이러한 *tests* 디렉토리 내의 파일에 대한 이색적인 동작 방식은 가장 주목할 만 점입니다. 이를테면, 만일 우리가 *tests/common.rs* 이라는 파일을 만들어서 그 안에 아래와 같이 **setup**이라는 이름의 함수를 위치시키고, 여기에 여러 테스트 파일들 내의 여러 테스트 함수로부터 호출될 수 있기를 원하는 어떤 코드를 집어넣는다면:

Filename: *tests/common.rs*

```
pub fn setup() {
    // 여러분의 라이브러리 테스트에 특화된 셋업 코드가 여기 올 것입니다
}
```

만약 테스트를 다시 실행시키면, 비록 이 코드가 어떠한 테스트 함수도 담고 있지 않고, **setup** 함수를 다른 어딘가에서 호출하고 있지 않을지라도, *common.rs* 파일을 위한 테스트 출력 내의 새로운 섹션을 보게 될 것입니다:

```
running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Running target/debug/deps/common-b8b07b6f1be2db70

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Running target/debug/deps/integration_test-d993c68b431d39df

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

`running 0 tests`이 표시되는 테스트 출력이 보이는 `common`을 만드는 건 우리가 원하던 것이 아닙니다. 우리는 그저 다른 통합 테스트 파일들에서 어떤 코드를 공유할 수 있기를 원했지요.

`common`이 테스트 출력에 나타나는 것을 막기 위해서는, `tests/common.rs`를 만드는 대신, `tests/common/mod.rs`를 만듭니다. 7장의 "모듈 파일 시스템의 규칙"절에서 서브모듈을 가지고 있는 모듈의 파일들을 위해 `module_name/mod.rs`라는 이름 규칙을 이용했었고, 여기서 `common`에 대한 서브모듈을 가지고 있지는 않지만, 이러한 방식으로 파일명을 정하는 것이 러스트에게 `common` 모듈을 통합 테스트 파일로 취급하지 않게끔 전달해줍니다. `setup` 함수 코드를 `tests/common/mod.rs`로 옮기고 `tests/common.rs` 파일을 제거하면, 테스트 출력에서 해당 섹션이 더 이상 나타나지 않을 것입니다. `tests` 디렉토리의 서브 디렉토리 내의 파일들은 개별적인 크레이트처럼 컴파일되지도, 테스트 출력의 섹션을 갖지도 않습니다.

`tests/common/mod.rs`를 만든 뒤에는, 어떤 통합 테스트 파일에서라도 이를 모듈처럼 쓸 수 있습니다. 아래에 `tests/integration_test.rs` 내에 `it_adds_two` 테스트로부터 `setup` 함수를 호출하는 예제가 있습니다:

Filename: `tests/integration_test.rs`

```
extern crate adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

`mod common;` 선언은 Listing 7-4에서 보여주었던 모듈 선언과 동일한 점을 주목하세요. 그런 다음 테스트 함수 내에서 `common::setup()` 함수를 호출 할 수 있습니다.

바이너리 크레이트를 위한 통합 테스트

만약 우리의 프로젝트가 `src/lib.rs` 파일이 없고 `src/main.rs` 파일만 갖고 있는 바이너리 프로젝트라면, `tests` 디렉토리 내에 통합 테스트를 만들어서 `src/main.rs`에 정의된 함수를 가져오기 위하여 `extern crate`를 이용할 수 없습니다. 오직 라이브러리 크레이트만 다른 크레이트에서 호출하고 사용할 수 있는 함수들을 노출시킵니다; 바이너리 크레이트는 그 스스로 실행될 것으로 여겨집니다.

이는 바이너리를 제공하는 러스트 프로젝트들이 `src/lib.rs`에 위치한 로직을 호출하는 간단한 형태의 `src/main.rs`를 가지고 있는 이유 중 하나입니다. 이러한 구조와 함께라면, `extern crate`를 이용하여 중요한 기능들을 커버하도록 하기 위해 통합 테스트가 라이브러리 크레이트를 테스트할 수 있습니다. 만일 중요 기능이 작동한다면, `src/main.rs` 내의 소량의 코드 또한 동작할 것이고, 이 소량의 코드는 테스트할 필요가 없습니다.

정리

러스트의 테스트 기능은 코드를 변경하더라도 계속하여 우리가 기대한 대로 동작할 것이라는 확신을 주기 위하여 코드가 어떻게 기능하는지 명시하는 방법을 제공합니다. 단위 테스트는 라이브러리의 서로 다른 부분을 개별적으로 시험하며 비공개된 구현 세부사항을 테스트할 수 있습니다. 통합 테스트는 라이브러리의 많은 부분이 함께 작동하는 사용 상황을 다루며, 외부 코드가 사용하게 될 똑같은 방식대로 테스트하기 위해 그 라이브러리의 공개 API를 이용합니다. 비록 러스트의 타입 시스템과 소유권 규칙이 몇 가지 종류의 버그를 방지하는데 도움을 줄지라도, 테스트는 여러분의 코드가 어떻게 동작하기를 기대하는지와 함께 해야 하는 논리 버그를 잡는 일을 도와주는 데에 있어 여전히 중요합니다.

이 장과 이전 장들의 지식을 합쳐서 다음 장의 프로젝트 작업을 해봅시다!

I/O 프로젝트: 커맨드 라인 프로그램 만들기

이 장에서 우리는 지금까지 배운 많은 내용을 요약 정리하고 몇 가지 표준 라이브러리 기능을 탐색하고자 합니다. 현재 우리가 보유한 러스트 실력을 연습하기 위한 커맨드 라인 툴을 만들고 파일, 커맨드 라인 입출력 작업을 해보게 될 것 입니다.

러스트는 성능, 안전성, '단일 바이너리'로 출력, 그리고 교차 플랫폼 지원으로 커맨드 라인 툴을 제작하기 좋은 언어입니다. 그러니 우리는 고전적인 커맨드 라인 툴 `grep`을 우리 자체 버전으로 만들어 볼 것입니다. Grep은 "정규 표현식 검색 및 인쇄"의 약어입니다. `grep`의 간단한 사용 예로 다음의 단계를 거쳐 지정된 파일에서 지정된 문자를 검색합니다.

- 인자로 파일 이름과 문자를 취합니다.
- 파일을 읽어들입니다.
- 문자 인자를 포함하는 파일의 행들을 찾습니다.
- 해당 라인들을 표시합니다.

우리는 또한 환경 변수를 사용하는 방법과 표준 출력 대신 표준 에러로 표시하는 방법을 다루고자 합니다. 이러한 기법들은 일반적으로 커맨드 라인 도구들에서 사용됩니다.

한 러스트 커뮤니티 멤버인 Andrew Gallant가 이미 `grep`의 전체 기능이 구현됐으면서도 월등히 빠른 `ripgrep`을 만들었습니다. 이에 비해 우리의 `grep`은 훨씬 간단하게 만들 것 입니다, 이번 장에서 `ripgrep`과 같은 실제 프로젝트를 이해하는데 필요한 배경지식을 제공합니다.

이 프로젝트는 우리가 지금까지 학습한 다양한 개념을 종합하게 될 겁니다:

- 조직화된 코드 (7장 모듈 편에서 배운 내용)
- 벡터와 문자열의 사용 (8장 콜렉션)
- 에러 처리 (9장)
- 특성과 생명주기를 적절히 사용하기 (10장)
- 테스트 작성 (11장)

또한 우리는 클로저, 반복자, 특성 개체를 간단히 소개하고자 합니다. 이는 13장과 17장에서 상세히 다룰 겁니다.

언제나처럼 `cargo new`를 통해 새로운 프로젝트를 생성합니다. 새 프로젝트의 이름을 `greprs`로 이름 지어서 시스템에 이미 존재하는 `grep`과 구분짓도록 하겠습니다:

```
$ cargo new --bin greprs
   Created binary (application) `greprs` project
$ cd greprs
```

커맨드라인 인자 허용하기

우리의 첫 번째 작업은 `greprs` 가 두 개의 커맨드라인 인자를 받을 수 있도록 하는 것 입니다: 파일이름과 검색할 문자. 즉, `cargo run` 을 통해 함께 우리의 프로그램을 수행시킬 때, 검색할 문자와 검색할 파일의 경로를 사용할 수 있도록 하고자 합니다, 다음처럼 말이죠:

```
$ cargo run searchstring example-filename.txt
```

현재로서는, `cargo new` 를 통해 생성된 프로그램은 우리가 입력한 인자를 모두 무시합니다. crates.io 라이브러리에 커맨드라인 인자들을 받아들이도록 도와줄 라이브러리가 이미 존재하지만, 우리 스스로 이를 구현해봅시다.

인자값 읽어들이기

우리 프로그램에 전달된 커맨드라인 인자의 값을 얻으려면 Rust의 표준 라이브러리에서 제공되는 함수를 호출해야합니다: `std::env::args`. 이 함수는 반복자(iterator) 형식으로 커맨드라인 인자들을 우리 프로그램에 전달해줍니다. 우리는 아직 반복자에 대해 다루지 않았고, 13장에서 다룰 예정이니, 지금은 반복자에 대해서 두 가지 성질만 알고 갑시다.

1. 반복자는 하나의 연속된 값을 생성합니다.
2. 반복자에 `collect` 함수 호출을 통해 반복자가 생성하는 일련의 값을 벡터로 변환할 수 있습니다.

한번 해볼까요? 항목 12-1의 코드를 사용하여 모든 커맨드라인 인자들을 벡터 형태로 `greprs` 로 전달해봅시다.

Filename: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{}: {}", args);
}
```

항목 12-1: 커맨드라인 인자를 벡터 형태로 모으고 그들을 출력하기.

가장 먼저, 우리는 `std::env` 모듈을 `use` 를 통해 모듈 범위 내로 가져와서 그 안의 `args` 함수를 호출할 수 있도록 합니다. `std::env::args` 함수는 두 단계 모듈들로 중첩된 호출임을 주지하세요. 7장에서 우리가 이야기 나눴듯이, 원하는 함수가 두 개 이상의 중첩된 모듈에 있는 경우에는 함수 자체가 아닌 부모 모듈을 범위로 가져오는게 일반적입니다.

이런 방식은 우리가 `std::env`의 다른 함수를 사용하기 용이하도록 하며 덜 모호합니다. `use std::env::args;`를 사용하여 `args`처럼 함수를 호출하면 현재 모듈에 이 함수가 정의된 것처럼 착각할 수 있습니다.

참고: 어떤 인자가 잘못된 유니코드를 포함하고 있으면 `std::env::args`는 패닉을 발생합니다. 유효하지 않은 유니코드를 포함한 인자를 허용해야 하는 경우에는 `std::env::args_os`를 대신 사용하도록 하세요. 이 함수는 `String` 대신 `OsString` 값을 반환합니다. `OsString` 값은 플랫폼마다 다르며 `String` 값보다 다루기가 더 복잡하기 때문에 여기서는 `std::env::args`를 사용하여 좀더 단순화 했습니다.

`main`의 첫 번째 줄에서, 우리가 호출한 `env::args`, 그리고 동시에 사용한 `collect`는 반복자가 가진 모든 값을 벡터 형태로 변환하여 반환합니다. `collect` 함수는 많은 종류의 콜렉션들과 사용될 수 있기 때문에, 우리가 원하는 타입이 문자열 벡터라고 `args`의 타입을 명시합니다. Rust에서 타입 명시를 할 필요는 거의 없지만, Rust는 우리가 원하는 콜렉션의 타입을 추론 할 수 없기 때문에 `collect`는 타입을 명시할 필요가 있는 함수 중 하나입니다.

마지막으로, 우리는 디버그 형식자인 `:?`으로 벡터를 출력합니다. 인자 없이, 그리고 두 개의 인자들로 우리의 코드를 실행시켜 봅시다.

```
$ cargo run
["target/debug/greprs"]

$ cargo run needle haystack
...snip...
["target/debug/greprs", "needle", "haystack"]
```

벡터의 첫 번째 값은 바이너리의 이름인 "target / debug / minigrep"입니다. 이것은 C에서 인수 목록의 동작을 일치시키고 프로그램은 실행시 호출 된 이름을 사용하게합니다. 메시지를 인쇄하거나 프로그램을 호출하는 데 사용 된 명령 줄 별칭을 기반으로 프로그램의 동작을 변경하려는 경우 프로그램 이름에 액세스하는 것이 편리하지만 장의 목적을 위해 무시할 것입니다 우리가 필요로하는 두 가지 주장만 저장하면됩니다.

벡터의 첫 값이 "target/debug/greprs"으로 바이너리의 이름임을 알 수 있습니다. 왜 그런지에 대한 내용은 이번 장을 넘어가니, 우리가 필요한 두 인자를 저장하였음을 기억하면 되겠습니다.

변수에 인자 값을 저장하기

인자값들이 들어있는 벡터의 값을 통해 우리의 프로그램에서 커맨드라인 인자의 원하는 값에 접근하는 것이 가능하다는 것을 상상할 수 있습니다. 다음은 정확히 우리가 원하는 방식이 아니지만, 두개

의 인자값을 변수로 저장하여 그 값을 우리의 프로그램에서 사용할 수 있도록 합니다.

항목 12-2대로 해봅시다:

Filename: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

항목 12-2: 쿼리와 파일 이름 인자를 보관하는 두 변수를 만듭니다.

우리가 벡터를 출력했을 때 봤듯이, 프로그램의 이름이 벡터의 첫 번째 값으로 `args[0]`에 저장되어 있으니, 우리는 1 번째 색인부터 접근하면 됩니다. `greprs`의 첫 번째 인자는 검색하고자 하는 문자열이므로, 우리는 첫 번째 인자의 참조자를 `query`에 저장합니다. 두 번째 인자는 파일 이름이니, 두 번째 인자의 참조자를 변수 `filename`에 저장합니다.

임시적으로 우리는 이 값을 단순 출력하고 있으니, 우리의 코드가 우리가 원하는 방식으로 동작하고 있다는 것을 증명하기 위해, 이 프로그램을 `test` 와 `sample.txt`를 인자로 주고 다시 실행해봅시다:

```
$ cargo run test sample.txt
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/greprs test sample.txt`
Searching for test
In file sample.txt
```

훌륭하게, 동작하네요! 우리는 인자 값을 우리가 원하는 변수에 정확히 저장했습니다. 후에 사용자가 아무런 인자를 넣지 않은 상황을 다루기 위해 오류처리를 추가해볼 겁니다. 하지만 당장은 그것보다 파일 읽기 기능을 추가해봅시다.

파일 읽기

다음으로, 우리는 커맨드 라인 인자 파일이름으로 지정된 파일을 읽어볼 겁니다. 먼저, 함께 테스트 할 샘플 파일이 필요합니다. 'greprs'가 동작하는 것을 확신할 수 있기 위해 가장 좋은 종류의 파일은 몇 개의 반복되는 단어의 다수의 줄에 걸쳐 존재하는 작은 양의 텍스트입니다. 항목 12-3의 에밀리 딕킨스 시는 잘 작동할 겁니다. `poem.txt`로 명명된 파일을 당신의 프로젝트 최상위에 생성하고 시를 입력합시다 "I'm nobody! Who are you?":

Filename: poem.txt

```
I'm nobody! Who are you?  
Are you nobody, too?  
Then there's a pair of us – don't tell!  
They'd banish us, you know.
```

```
How dreary to be somebody!  
How public, like a frog  
To tell your name the livelong day  
To an admiring bog!
```

항목 12-3: 테스트 용으로 적합한 에밀리 딕킨슨의 시 "I'm nobody! Who are you?"

언급된 위치에 위의 파일을 생성한 후, `src/main.rs` 파일을 아래 항목 12-4의 내용을 참고하여 편집합니다.

Filename: src/main.rs

```

use std::env;
use std::fs::File;
use std::io::prelude::*;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);

    let mut f = File::open(filename).expect("file not found");

    let mut contents = String::new();
    f.read_to_string(&mut contents).expect("Something went wrong reading the
file");

    println!("With text:\n{}", contents);
}

```

항목 12-4: 두 번째 인자로 특정된 파일의 내용 읽어들이기

먼저, 우리는 `use` 문 몇 개를 추가하여 표준 라이브러리에서 관련 있는 부분을 가져옵니다: 우리는 파일 관련하여 `std::fs::File` 과, 파일 I/O를 포함한 I/O 작업을 위해 유용한 다양한 특성이 있는 `std::io::prelude::*`이 필요합니다.

Rust가 가진 지정된 것들을 영역 내로 가져오는 일반적인 도입부와 동일하게, `std::io` 모듈은 당신이 I/O 작업을 할 때 필요할만한 일반적인 것들에 대한 그 자신만의 도입부를 갖습니다. 기본적인 도입부와는 다르게, 우리는 반드시 `std::io`의 도입부를 명시적으로 `use` 해야 합니다.

`main`에서, 우리는 다음 세 가지를 추가했습니다: 첫 째, `File::open` 함수를 호출하고 `filename` 값을 전달하여 파일을 변경할 수 있는 핸들을 얻습니다. 두 번째로, `contents`라는 이름의 빈 `String` 가변 변수를 만들었습니다. 이 변수는 우리가 읽어들인 내용을 보관하기 위한 용도로 사용될 겁니다. 셋 째, 우리가 만들어 둔 파일 핸들에 `read_to_string`을 호출하여 가변 참조를 `contents`의 인자로 전달합니다.

이후, 임시로 `println!`을 추가하여 `contents`의 값을 출력함으로서 파일을 읽어들인 이후 우리 프로그램이 제대로 동작했는지 확인할 수 있습니다.

아무 문자나 첫 번째 커맨드라인 인자로 입력하고(우리가 아직 검색 부분을 구현하지 않았기 때문에) 두 번째는 우리가 만들어 둔 `poem.txt` 파일로 입력하여 이 코드를 실행해봅시다.

```
$ cargo run the poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
        Running `target/debug/greprs the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us – don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

좋군요! 우리의 코드가 파일 내용을 읽고 출력했습니다. 우리 프로그램은 몇 가지 결점이 있습니다: `main` 함수는 많은 책임을 지고(역주: [단일 책임 원칙](#) 참고), 우리가 할 수 있는 에러처리를 하지 않았습니다. 아직 우리의 프로그램이 작기 때문에, 이 결점들은 큰 문제가 아닐 수도 있습니다. 하지만 우리 프로그램 커져가면, 점점 이를 깔끔하게 수정하기 어렵게 됩니다. 프로그램의 개발 초기 단계에 리팩토링을 하면 코드의 양이 적은만큼 리팩토링을 하기 훨씬 쉬워지기 때문에 훌륭한 단련법입니다. 그러니 지금 해봅시다.

모듈성과 에러처리의 향상을 위한 리팩토링

우리 프로그램을 향상시키기 위해 네 가지 수정하고 싶은 문제가 있는데, 이들은 프로그램을 구조화하고 발생 가능한 에러를 처리하는 방식과 관련있습니다.

첫 번째, 우리 `main` 함수는 현재 두 가지 작업을 수행합니다: 인자들을 분석하고 파일을 열지요. 이런 작은 함수에서, 이건 큰 문제가 안됩니다. 하지만 우리가 계속해서 `main` 함수 안에 프로그램을 작성하여 커지게 되면, `main` 함수가 처리하는 작업의 수도 늘어나게 될 겁니다. 함수가 갖게되는 책임들만큼, 근원을 파악하기도, 테스트 하기에도, 부분 별로 나누지 않고는 수정하기도 어려워 집니다. 함수는 나뉘어 하나의 작업에 대해서만 책임을 지는 것이 더 좋은 구조입니다.

이 문제는 우리의 두 번째 문제와도 관련이 있습니다: `query` 와 `filename` 은 프로그램의 설정을 저장하는 변수이고 `f` 와 `contents` 같은 변수는 프로그램의 논리 수행에 사용됩니다. `main`이 길어질수록 범위 내에 더 많은 변수가 생깁니다. 범위 내에 더 많은 변수가 존재할수록, 각각의 변수를 추적하기 힘들어집니다. 목적을 분명히 하기 위해 설정 변수를 그룹화하여 하나의 구조로 결합시키는 것이 좋습니다.

세 번째 문제는 파일 열기가 실패 할 경우 `expect`를 사용하여 오류 메시지를 출력해주는데, 에러 메시지가 `Something went wrong reading the file` 밖에 없습니다. 파일이 존재하지 않는 경우 외에도 파일 열기가 실패하는 경우들이 있습니다. 예를 들어 파일은 존재하지만 파일을 열 수 있는 권한이 없을 수 있습니다. 현재는 이런 상황에도 `Something went wrong reading the file` 이란 오류 메시지를 출력하여 사용자에게 잘못된 조언을 해주게 됩니다.

넷째, 우리는 서로 다른 오류를 다루기 위해 `expect`를 반복적으로 사용하고 있습니다. 헌데 만약 사용자가 충분한 인수를 지정하지 않고 프로그램을 실행하면 Rust의 "index out of bounds" 오류가 발생하는데 이는 문제를 명확하게 설명하지 않습니다. 우리가 모든 오류처리 코드를 한 군데 모아놓으면 후에 관리자는 오류처리 로직을 변경해야 할 때 오직 이 곳의 코드만 참고하면 되니 더 좋죠. 또한, 모든 오류 처리 코드를 한 곳에 저장하면 우리가 최종 사용자에게 도움이 되는 메시지를 출력하고 있는지 확신하는데도 도움이 됩니다.

이런 문제들을 우리 프로젝트를 리팩토링하여 해결해보도록 하겠습니다.

바이너리 프로젝트를 위한 관심사의 분리

`main` 함수가 여러 작업에 책임을 갖게 되는 구조적 문제는 많은 바이너리 프로젝트에서 공통적입니다. 그래서 Rust 커뮤니티는 `main`이 커지기 시작할 때 바이너리 프로그램의 핵심기능을 나누기 위한 가이드라인 프로세스를 개발했습니다. 프로세스에는 다음 단계가 있습니다:

1. 당신의 프로그램을 `main.rs` 과 `/lib.rs` 로 나누고 프로그램의 로직을 `/lib.rs` 으로 옮깁니다.
2. 커맨드라인 파싱 로직이 크지 않으면, `main.rs` 에 남겨둬도 됩니다.
3. 커맨드라인 파싱 로직이 복잡해지기 시작할거 같으면, `main.rs` 에서 추출해서 `/lib.rs` 로 옮기세요.

4. 이런 절차를 통해 `main` 함수에는 다음의 핵심 기능들만 남아있어야 합니다:

- 인자 값들로 커맨드라인을 파싱하는 로직 호출
- 다른 환경들 설정
- `lib.rs`의 `run` 함수 호출
- `run`이 에러를 리턴하면, 에러 처리.

이 패턴이 핵심기능을 분리하는데 관한 모든 것입니다: `main.rs`는 프로그램 실행을 담당하고, `lib.rs`는 맑은 작업에 관한 로직을 담당합니다. `main` 함수는 직접 테스트 할 수 없지만, 이런 구조로 `lib.rs` 으로 프로그램의 모든 함수와 로직을 옮긴 후에는 테스트가 가능해집니다. `main.rs`에는 읽어서 옳바른지 여부를 검증할 수 있을 정도로 적은 코드만을 남겨두도록 합니다. 다음의 과정을 거치며 재작업을 해봅시다.

인자 파서의 추출

먼저 우리는 커맨드라인 인자를 분석하는 기능을 추출할 겁니다. 항목 12-5에서 `main`의 시작 부분이 새로운 함수 `parse_config`를 호출하는 것을 볼 수 있을텐데, 이는 아직은 `src/main.rs`에 정의되어 있을 겁니다.

Filename: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, filename) = parse_config(&args);

    // ...snip...
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

Listing 12-5: Extract a `parse_config` function from `main`

우리는 아직 커맨드라인 인자들을 벡터로 수집하고 있는데, 인덱스 1의 인수 값을 변수 `query` 에, 인덱스 2의 인수 값을 `main` 함수 내의 변수 `filename`에 할당하는 대신에 전체 벡터를 `parse_config` 함수로 전달합니다. `parse_config` 함수는 어디에 위치한 인자가 어떤 변수에 대입되는지에 대한 로직을 보유하고, 그 값을 `main`으로 되돌려 줍니다. 우리는 여전히 `query`와 `filename` 변수를 `main`에 생성하지만, `main`은 더 이상 커맨드라인 인자와 변수간의 상관 관계를 책임지지도 알아야 할 필요도 없죠.

이것이 우리가 작은 프로그램을 유지하기 위한 과도한 행동으로 보일 수도 있지만, 우리는 조금씩 점진적으로 리팩토링을 진행하고 있습니다. 이런 변화를 준 뒤에는, 프로그램을 다시 실행해 인자의 파싱이 정상적으로 동작하고 있는지 확인해보십시오. 진행 상황을 자주 확인하면 문제가 생겼을 때 원인을 파악하는데 도움이 됩니다.

설정 변수들을 그룹짓기

우리는 이 함수의 기능을 더 향상시키기 위해 또 다른 작은 행동을 할 수 있습니다. 현재 우리는 튜플을 반환하고 있는데, 그 시점에 즉시 튜플을 개별된 부분으로 나눌 수가 없습니다. 이는 우리가 아직은 제대로 된 추상화를 하지 못하고 있다는 신호일 수 있습니다.

또 다른 의미로는 `config`의 부분인 `parse_config`에 향상시킬 지점이 있다는 것으로, 우리가 반환하는 두 개의 값은 관련되어 있으며 모두 하나의 설정 값에 대한 부분이죠. 우리는 현재 두 값을 튜플로 그룹화하는 것 이외의 다른 의미를 전달하지 않습니다. 두 값을 하나의 구조체에 넣고 각 구조체 필드에 의미있는 이름을 지정할 수 있습니다. 이렇게 하면 이 코드의 향후 유지 보수 담당자가 서로 다른 값이 서로 어떻게 관련되어 있고 그 목적이 무엇인지 쉽게 이해할 수 있습니다.

주의: 어떤 사람들은 복합 타입(complex type)이 더 적절할 경우에도 기본 타입(primitive type)을 사용하는데 이러한 안티 패턴을 강박적 기본타입 사용(primitive obsession)이라 부릅니다

항목 12-6에서 `query`와 `filename`을 필드로 갖는 `Config`란 구조체 정의가 추가된 것을 볼 수 있습니다. 우리는 또한 `parse_config` 함수를 변경하여 `Config` 구조체의 객체를 반환하게 변경하였으며, `main`에서 별개의 변수가 아닌 구조체의 필드를 사용하도록 변경했습니다.

Filename: src/main.rs

```

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    let mut f = File::open(config.filename).expect("file not found");

    // ...snip...
}

struct Config {
    query: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let filename = args[2].clone();

    Config { query, filename }
}

```

Listing 12-6: Refactoring `parse_config` to return an instance of a `Config` struct

이제 `parse_config`의 선언은 `Config` 값을 반환한다는 것을 알려줍니다. `parse_config`의 내부에서는 `args`의 `String` 값을 참조하는 문자열 조각을 반환했었지만, 이제는 `Config`를 정의하고 자체 `String`의 값을 포함하도록 선택했습니다. `main`의 `args` 변수는 인자 값들의 소유주로 `parse_config`에는 그들을 대여해줄 뿐입니다. 그렇기에 만약 `Config`가 `args`의 값들에 대한 소유권을 가지려고 시도하면 Rust의 대여 규칙을 위반하게 됩니다.

우리가 `String` 데이터를 관리하는 방식은 여러가지가 있겠습니다만, 가장 쉽고 약간 비효율적인 방법은 `clone` 메소드를 호출하는 겁니다. 이 방식은 `Config` 객체에서 소유하게 할 `data` 전체에 대한 복사본을 만들 것이며, 이런 방식은 참조만 보관하는 것에 비해 약간 더 많은 비용과 메모리가 소비됩니다. 하지만 데이터의 복제본을 만드는 방식은 우리가 참조의 생명주기를 관리하지 않아도 되기 때문에 우리의 코드를 매우 직관적이게 합니다. 그래서 이런 상황에서는 약간의 성능을 포기하고 간소함을 유지하는 것이 매우 가치있는 거래입니다.

clone 사용의 기회비용

많은 Rust 사용자들은 런타임 비용 때문에 소유권 문제를 수정하기 위해 `clone`을 사용하지 않는 경향이 있습니다. 13장 이터레이터에서, 이런 상황에서보다 효율적인 메소드를 사용하는 법을 배우겠지만, 지금은 한 번만 `clone`하며 `query`와 `filename`이 매우 작기 때문에 몇 개의 문자열을 `clone`하여 진행하는 것이 좋습니다. 첫 번째 단계에서는 코드를 최대한 최적화하는 것보다 약간 비효율적이더라도 넘어가는게 좋습니다. Rust에 대한 경험이 많을수록 바람직한 방법으로 곧장 진행할 수 있을 겁니다. 지금은 `clone`을 호출하는 것이 완벽한 선택입니다.

`parse_config`에 의해 반환된 `Config`의 객체를 `config`라는 변수에 넣고 이전에 별도로 `query`와 `filename`이란 이름으로 나뉘어 있던 변수 대신 `Config` 구조체의 필드를 사용하도록 `main`을 업데이트했습니다.

우리의 코드는 이제 보다 분명하게 `query`와 `filename`이 연관되어 있으며 이들의 목적이 프로그램이 어떻게 동작할지에 대한 설정이라는 의도를 전달할 수 있습니다. 이 값을 사용하는 모든 코드는 그들의 의도에 맞게 지정된 필드를 `config` 객체에서 찾을 수 있습니다.

Config를 위한 생성자 만들기.

지금까지 우리는 `main`에서 `parse_config` 함수로 커맨드라인 인자를 파싱하는 로직을 추출했습니다. 이를 통해 우리 코드에서 `query`와 `filename` 값이 연관되어 있고 그 연결성이 전달되어야 한다는 것을 알았습니다. 그래서 우리는 `Config` 구조체를 추가하고 그 의도와 목적에 맞게 `query`와 `filename`을 명명했으며 `parse_config` 함수에서 변수의 이름을 구조체 필드 이름으로 반환 할 수 있게 했습니다.

그래서 이제 `parse_config` 함수의 목적은 `Config` 객체를 생성하는 것인데, 우리는 `parse_config`라는 평범한 함수를 `Config` 구조체와 관련된 `new`라는 함수로 변경 할 수 있습니다. 이런 변경은 우리의 코드를 보다 자연스럽게 만들어 줍니다: `String::new`를 호출하여 `String` 형의 객체를 생성하는 것처럼 표준 라이브러리들의 객체를 생성할 수 있습니다. 그리고 `parse_config`를 `Config`와 연관된 `new` 함수로 변경하게 되면, 우리는 `Config`의 객체를 `Config::new`를 호출하여 생성할 수 있게 됩니다. 항목 12-7는 우리가 해야할 변동사항 보여줍니다.

Filename: src/main.rs

```

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // ...snip...
}

// ...snip...

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}

```

Listing 12-7: Changing `parse_config` into `Config::new`

우리는 `main`을 갱신하여 `parse_config`를 호출하는 대신 `Config::new`를 호출하게 되었습니다. 우리는 `parse_config`의 이름을 `new`로 바꾸고 그를 `impl` 블록 안으로 옮겼는데, 이를 통해 `new` 함수가 `Config`와 연결되게 됩니다. 다시 컴파일을 하고 제대로 동작하는지 확인해보도록 합시다.

에러 처리 수정하기

이번에는 우리의 에러 처리를 수정해 볼 겁니다. 만일 `args` 벡터가 3개 미만의 아이템을 가지고 있을 때 인덱스 `2` 혹은 `3`의 값에 접근하려는 시도를 하면 프로그램은 패닉을 일으키게 된다고 했던 것을 상기시켜 드립니다. 프로그램을 인자 없이 실행해보시면; 다음같이 될 겁니다.

```

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/greprs`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', /stable-dist-rustc/build/src/libcollections/vec.rs:1307
note: Run with `RUST_BACKTRACE=1` for a backtrace.

```

`index out of bounds: the len is 1 but the index is 1` 줄은 프로그래머를 위해 의도된 에러 메시지이지, 최종 사용자에게는 무슨 일이 있었는지 무엇을 해야 하는지 이해하는데 아무런 도움이 되지 않습니다. 당장 한번 고쳐보겠습니다.

에러 메시지 향상시키기

항목 12-8에서 `new` 함수에 검사를 추가하여 인덱스 1과 2에 접근하기 전에 조각이 충분한 길이인지를 확인합니다. 조각이 충분히 길지 않다면, 프로그램은 더 좋은 에러메시지 `index out of bounds`를 보여주고 패닉을 일으킵니다:

Filename: src/main.rs

```
// ...snip...
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
    // ...snip...
```

항목 12-8: 인자의 숫자가 몇 개인지 검증 추가

이것은 항목 9-8에서 작성한 `Guess::new` 함수와 유사합니다. 이 함수는 `value` 인수가 유효한 값의 범위를 벗어난 경우 `panic!`을 호출했습니다. 값의 범위를 검사하는 대신에, 우리는 `args`의 길이가 적어도 3개인지 검사하면, 함수의 나머지 부분은 이 조건이 이미 충족되었다는 가정 하에서 동작할 수 있습니다. `args`가 3개 보다 적은 아이템을 가진다면, 이 조건은 `true`가 되고 우리는 `panic!` 매크로를 호출해 프로그램을 즉시 종료 시킬겁니다.

이런 몇 줄의 추가 코드들을 `new` 상에 추가하고, 우리 프로그램을 아무 인자도 없이 다시 실행시키면 다음과 같은 에러를 볼 수 있을 겁니다.

```
$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/greprs`
thread 'main' panicked at 'not enough arguments', src/main.rs:29
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

이 결과 더 합리적인 좋은 오류 메시지가 표시됩니다. 그러나 사용자에게 제공하고 싶지 않은 추가 정보가 있습니다. 따라서 항목 9-8에서 사용한 기술을 사용하는 것은 여기선 최선의 방법은 아닙니다. `panic!`에 대한 호출은 9장에서 논의했던 것처럼 사용 방법에 대한 문제가 아닌 아니라 프로그래밍 관련 문제에 더 적합합니다. 대신, 우리는 9장에서 배운 다른 기법으로 `Result`를 반환하는 것을 성공이나 오류를 나타낼 수 있습니다.

`new`에서 `panic!`을 호출하는 대신 `Result`를 반환하기.

우리는 `Result`를 반환 값으로 선택하여 성공인 경우에는 `Config` 객체를 포함시키고 에러가 발생한 경우에는 문제가 무엇인지 설명할 수 있게 만들 수 있습니다. `Config::new`가 `main`과 상호작용할 시에, 우리

는 `Result`를 사용하여 문제가 있다고 신호할 수 있습니다. 그리고 `main`에선 `Err`의 값을 사용자들에게 보다 실용적인 방식으로 변환하여 보여줄 수 있습니다. `thread 'main'`으로 시작하는 문자들과 `panic!`을 사용해서 보여지는 `RUST_BACKTRACE` 관련 메시지 없이.

항목 12-9에서 당신이 변경해야 할 `Config::new`의 반환 값과 `Result`를 반환하기 위한 함수 본문을 보여줍니다:

Filename: src/main.rs

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```

항목 12-9: `Config::new`에서 `Result` 반환

우리의 `new` 함수는 이제 성공 시에는 `Config` 객체가 에러 시에는 `&'static str`가 포함된 `Result`를 반환하게 됩니다. 10장의 "The Static Lifetime"에서 `&'static str`이 문자열 리터럴이라고 다뤘는데, 이게 현재 우리의 에러 타입입니다.

우리는 `new` 함수의 본문에서 두 가지 변경을 했습니다: 사용자가 충분한 인수를 전달하지 않을 때 `panic!`을 호출하는 대신 `Err` 값을 반환하고 `Config`를 반환할 때는 `Ok`로 포장하여 반환 합니다. 이런 변경으로 인해 함수는 새로운 타입 선언을 갖게 됩니다.

`Config::new` 가 `Err` 값을 반환하게 함으로써, `main` 함수는 `new` 함수로부터 반환된 `Result` 값을 처리하고 에러 상황에 프로세스를 더 깨끗하게 종료 할 수 있습니다.

Config::new를 호출하고 에러 처리하기

에러 케이스를 처리하고 사용자-친화적인 메시지를 출력하기 위해서, 항목 12-10에서처럼 `Config::new` 가리킨하는 `Result`를 처리하기 위해 `main`을 간신히 해야 합니다. 그리고 우리 커맨드라인 프로그램을 `panic!`으로 0이 아닌 값을 발생시킬 때에는 종료시켜야 하므로 직접 구현해보도록 합시다. 0이 아닌 종료 값은 우리 프로그램을 호출한 프로그램에게 우리의 프로그램이 에러 상태로 종료되었음을 알리는 규칙입니다.

Filename: src/main.rs

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // ...snip...
}
```

항목 12-10: new Config 가 실패했을 때 에러 코드와 함께 종료시키기

이 목록에서 우리는 이전에 다루지 않았던 메소드를 사용하고 있습니다: `unwrap_or_else`는 표준 라이브러리에 의해 `Result <T, E>`에 정의되어 있습니다. `unwrap_or_else`를 사용하면 `panic!` 이 아닌 에러 처리를 직접 정의 할 수 있습니다. `Result`가 `Ok` 값이면, 이 메소드의 동작은 `unwrap`과 유사합니다: 그것은 `Ok`로 포장한 내부 값을 반환합니다. 그러나 `Err` 값이면 메소드는 *closure*의 코드를 호출합니다. *closure*는 익명의 함수로 `unwrap_or_else`에 인수로 전달됩니다. 13장에서 클로저에 대해 더 자세히 다룰 것입니다. 여기서 알아 두어야 할 것은 `unwrap_or_else`가 `Err`의 내부 값, 이번 경우에는 항목 12-9에서 우리가 추가한 정적 문자열인 `not enough arguments`을, 수직파이프 사이에 위치하는 `err`로 인자로서 우리의 클로저로 전달한다는 겁니다. 클로저에 있는 코드는 이런 과정을 거쳐 실행 시에 `err` 값을 사용할 수 있습니다.

우리는 새 `use` 줄을 추가하여 `process`를 공유 라이브러리에서 import했습니다. 에러 상황에 실행될 클로저의 코드는 단 두 줄입니다. 에러 값을 출력해주고 `process::exit`를 호출합니다. `process::exit` 함수는 프로그래임 즉시 중단시키고 종료 상태 코드로 전달받은 값을 반환합니다. 이것은 항목 12-8에서 사용한 `panic!` 기반의 처리 방식과 유사해 보이지만, 더이상 필요하지 않은 출력을 하지 않죠. 해볼까요?

```
$ cargo run
Compiling greprs v0.1.0 (file:///projects/greprs)
Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
Running `target/debug/greprs`
Problem parsing arguments: not enough arguments
```

훌륭하네요! 이 출력은 우리 사용자들에게 훨씬 친화적입니다.

run 함수 추출하기

이제 환경 설정 파싱 리팩토링을 마무리 했습니다. 우리 프로그램의 로직으로 돌아갑시다. 우리가 "바이너리 프로젝트에서 핵심 기능의 분리" 절에서 논의한 과정에 따라, 우리는 `main` 함수에 구성 설정 또는 오류 처리와 관계 없는 남아있는 모든 로직들을 담고 있는 `run` 함수를 추출 할 겁니다. 이 과정이 종료되면, `main`은 간결해져 쉽게 검증할 수 있어지고, 우리는 다른 모든 로직에 대한 테스트를 작성할 수 있을 겁니다.

항목 12-11 추출된 `run` 함수를 보여줍니다. 현재 우리는 함수를 추출하여 `src/main.rs`에 함수를 정의하는 작고 점진적 개선만 수행하고 있습니다.

Filename: `src/main.rs`

```
fn main() {
    // ...snip...

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let mut f = File::open(config.filename).expect("file not found");

    let mut contents = String::new();
    f.read_to_string(&mut contents).expect("something went wrong reading the
file");

    println!("With text:\n{}", contents);
}

// ...snip...
```

항목 12-11: 남은 프로그램 로직을 `run` 함수로 추출하기

이제 `run` 함수에는 `main`에 잔존하는 파일을 읽는 것부터 나머지 모든 로직이 포함됩니다. `run` 함수는 `Config` 객체를 인수로 취합니다.

`run` 함수에서 예러 반환하기

나머지 프로그램 로직을 `main`이 아닌 `run` 함수로 분리하면, Listing 12-9의 `Config::new`처럼 예러 처리를 향상시킬 수 있습니다. `expect`를 호출하여 프로그램을 패닉 상태로 만드는 대신, `run` 함수는 무언가가 잘못되었을 때 `Result <T, E>`를 리턴 할 것입니다. 그러면 사용자 친화적인 방법으로 오류를 처리하는 로직을 `main`으로 통합 할 수 있습니다. 항목 12-12는 `run`의 선언부와 본문의 변경 사항을 보여줍니다.

Filename: src/main.rs

```
use std::error::Error;

// ...snip...

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    println!("With text:\n{}", contents);

    Ok(())
}
```

항목 12-12: `run` 함수가 `Result`를 반환하게 바꾸기

우리는 여기서 세 가지 큰 변화를 만들었습니다. 먼저, `run` 함수의 리턴 타입을 `Result <(), Box<dyn Error >>`로 바꿨습니다. 이 함수는 이전에 유닛 타입 `()`을 반환했으며, 우리는 `Ok`의 경우 반환할 값으로 이 타입을 유지합니다.

우리의 에러 타입으로, 특성 오브젝트 `Box`를 사용합니다 (그리고 상단에 `use` 문으로 `std::error::Error`를 범위 내로 임포트 해왔습니다). 우리는 특성 오브젝트들을 17장에서 다룰 것입니다. 지금 당장은, `Box<dyn Error>`는 함수가 `Error` 특성을 구현하는 타입을 반환한다는 것만 알면 되고, 특별히 어떤 타입이 반환될지에 대해서는 알 필요 없습니다. 이런 방식은 다양한 에러 상황에 다른 타입의 오류 값을 반환 할 수 있는 유연성을 확보할 수 있습니다. `dyn`은 "dynamic"의 약자입니다.

우리가 만든 두 번째 변화는 우리가 9 장에서 이야기했듯이, `?`에 대한 `expect`에 대한 호출을 제거한 것입니다. 에러 시에 `panic!`을 호출하는 것보다 현재 함수에서 에러 값을 반환하며 호출자가 처리 할 수 있도록 하였습니다.

셋째, 이 함수는 성공 사례에서 `Ok` 값을 반환합니다. 우리는 `run` 함수의 성공 타입을 선언부에서 `()`로 선언했습니다, 이것은 우리가 유닛 타입 값을 `Ok` 값으로 감쌀 필요가 있음을 의미합니다. 이 `Ok (())` 구문은 조금 이상하게 보일 수 있지만, `()`를 사용하는 것과 마찬가지로 이는 사이드이펙트 없이 `run`을 호출하는 것을 나타내는 관용적인 방법입니다. 우리가 필요로 하는 값을 반환하지 않습니다.

실행시키면, 컴파일 될텐데, 경고를 보여줍니다:

```
warning: unused result which must be used, #[warn(unused_must_use)] on by
default
--> src/main.rs:39:5
 |
39 |     run(config);
|     ^^^^^^
```

Rust는 우리 코드가 오류가 있음을 나타내는 `Result` 값을 무시한다는 것을 알려줍니다. 우리는 에러가 있는지 아닌지를 확인하지 않고 있고, 컴파일러는 우리에게 아마도 여기에 에러 처리 코드를 작성해야 한다는 것을 상기 시켜줄 것입니다! 당장 바로잡아 봅시다.

Rust는 우리 코드가 오류가 있음을 나타내는 'Result'값을 무시한다는 것을 알려줍니다. 우리는 에러가 있는지 아닌지를 확인하지 않고 있고, 컴파일러는 아마도 여기에 에러 처리 코드를 가지고 있다는 것을 상기 시켜줄 것입니다! 지금 바로 잡아 보자.

main안의 run에서 반환되는 에러 처리하기

우리는 항목 12-10의 `Config::new`를 사용하여 오류를 처리하는 방식과 비슷한 방법을 사용하여 오류를 검사하고 멋지게 처리합니다. 그러나 약간의 차이점이 있습니다.

Filename: src/main.rs

```
fn main() {
    // ...snip...

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);

        process::exit(1);
    }
}
```

우리는 `unwrap_or_else`를 호출하기보다 `if let`을 사용하여 `run`이 `Err` 값을 반환하는지 검사하고 만약 그렇다면 `process::exit(1)`을 호출합니다. `run`은 `Config::new`가 `Config` 객체를 반환하는 것처럼 우리가 `unwrap`하기를 원하는 값을 반환하지 않습니다. 왜냐하면 `run`은 성공하면 `()`를 반환하기 때문에, 우리는 에러가 발생한 경우만 신경쓰면 됩니다. 그래서 우리는 `unwrap_or_else`을 통해 포장을 벗길 필요가 없죠, 값은 무조건 `()` 일테니까요.

`if let`과 `unwrap_or_else` 함수의 내용은 동일한 경우에 동일한 동작을 합니다, 오류를 출력하고 종료 하죠.

라이브러리 크레이트로 코드를 나누기

지금까지 꽤 좋아 보인다! 이제 우리는 `src/main.rs` 파일을 나눠서 `src/lib.rs`에 몇 개의 코드를 넣어서 테스트 할 수 있고 작은 `src/main.rs` 파일을 갖게 될 것입니다.

`src/main.rs`에 파편으로 존재하는 다음 코드들을 새 파일로 옮겨봅시다. `src/lib.rs`:

- `run` 함수 정의
- 관련있는 `use` 문들
- `Config`의 정의
- `Config::new` 함수와 정의

`src/lib.rs`의 내용은 항목 12-13에서 보이는 것과 같을겁니다.

Filename: `src/lib.rs`

```

use std::error::Error;
use std::fs::File;
use std::io::prelude::*;

pub struct Config {
    pub query: String,
    pub filename: String,
}

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>>{
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    println!("With text:\n{}", contents);

    Ok(())
}

```

항목 12-13: Config과 run을 src/lib.rs로 옮기기

우리는 `Config`의 필드 및 `new` 메소드와 `run` 함수에 대해 `pub`을 자유롭게 사용했습니다. 이제 우리가 테스트 할 수 있는 공개 API를 가진 라이브러리 크레이트가 생겼습니다.

바이너리 크레이트에서 라이브러리 크레이트 호출하기

이제 우리는 `src/main.rs`에 있는 바이너리 크레이트의 범위에 `src/lib.rs`로 옮긴 코드를 `extern crate greprs`를 사용하여 가져와야 합니다. 이후 `use greprs::Config` 행을 추가하여 `Config` 타입을 범위로 가져오고 항목 12-14와 같이 크레이트 이름으로 `run` 함수 앞에 접두사를 붙입니다.

Filename: `src/main.rs`

```

extern crate greprs;

use std::env;
use std::process;

use greprs::Config;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = greprs::run(config) {
        println!("Application error: {}", e);

        process::exit(1);
    }
}

```

항목 12-14: `greprs` 크레이트를 `src/main.rs` 범위로 연결하기

라이브러리 크레이트를 바이너리 크레이트에 가져 오려면 `extern crate greprs`을 사용합니다. 그런 다음 `greprs::Config` 줄을 추가하여 `Config` 타입을 범위로 가져오고 `run` 함수 접두어에 크레이트 이름을 붙입니다. 이를 통해 모든 기능이 연결되어 있어야 하며 작동해야 합니다. `cargo run`을 실행하여 모든 것이 올바르게 연결되어 있는지 확인하십시오.

아오! 빽시게 작업했네요, 우리의 미래를 우리 스스로가 성공의 방향으로 설정했습니다. 이제 에러를 처리가 훨씬 쉬워졌고, 우리의 코드를 보다 모듈화하였습니다. 거의 모든 작업은 여기 `src/lib.rs`에서 수행될 겁니다.

새롭게 확보한 모듈성을 통해 이전의 코드로는 하지 못했을 무언가를 쉽게 할 수 있는 이점을 확보했습니다: 몇 개의 테스트를 작성해봅시다!

테스트 주도 개발로 라이브러리의 기능 개발하기

`src/lib.rs`으로 로직을 추출하고 `src/main.rs`에 인수 수집 및 에러 처리를 남겨 두었으므로 우리의 핵심 기능 코드에 대한 테스트를 작성하는 것이 훨씬 쉬워졌습니다. 커맨드라인에서 바이너리를 실행할 필요없이 다양한 인수를 사용하여 함수를 직접 호출하고 반환 값을 확인할 수 있습니다. 자신이 만든 `Config::new`와 `run` 함수의 기능에 대해 몇 가지 테스트를 작성하면서 자유도를 느껴보세요.

이 섹션에서는 TDD(Test Driven Development) 프로세스에 따라 `minigrep`에 검색 로직을 추가합니다. 해당 소프트웨어 개발 기법은 다음의 단계를 따릅니다:

1. 실패할 테스트를 작성하고, 의도한 대로 실패하는지 실행해보세요.
2. 새 테스트를 통과하기 충분할 정도로 코드를 작성하거나 수정하세요.
3. 추가하거나 수정하는 정도의 리팩토링을 해보고, 여전히 테스트를 통과하는지 확인해보세요.
4. 1단계로 반복!

이것은 소프트웨어를 작성하는 여러 가지 방법 중 하나지만 TDD는 코드 설계를 좋은 상태로 유지시켜 줍니다. 코드를 작성하기 전에 테스트를 작성하고 테스트를 통과시키면 높은 테스트 범위를 유지하는데 도움이 됩니다. 테스트 패스를 작성하는 코드를 작성하기 전에 테스트를 작성하면 프로세스 전체에서 높은 테스트 적용 범위를 유지하는 데 도움이 됩니다.

우리는 실제로 파일 내용에서 쿼리 문자열을 검색하고 쿼리와 일치하는 줄의 목록을 생성하는 기능의 구현을 테스트 주도로 개발해 볼 겁니다. 이 기능을 `search`라는 함수에 추가 할 것입니다.

실패 테스트 작성하기

더 이상 필요하지 않으므로 프로그램의 동작을 확인하는 데 사용했던 `src/lib.rs` 및 `*src/main.rs *`에서 `println!` 문을 제거해 봅시다. 그런 다음 `src/lib.rs`에 11 장에서 했던 것처럼 `test` 함수가 있는 `test` 모듈을 추가 할 것입니다. `test` 함수는 `search` 함수에 필요한 동작을 지정합니다. 쿼리와 텍스트를 가져 와서 쿼리를 검색하고 쿼리를 포함하는 텍스트의 줄만 반환합니다. 항목 12-15는 아직 컴파일되지 않는 이 테스트를 보여줍니다.

Filename: `src/lib.rs`

```

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }
}

```

Listing 12-15: Creating a failing test for the `search` function we wish we had

이 테스트는 “duct.”라는 문자열을 검색합니다. 우리가 검색하는 텍스트는 세 줄로, 한 줄은 “duct.”를 포함합니다. 우리는 `search` 함수에서 반환하는 값이 우리가 예상한 줄이어야 한다고 단정했습니다(`assert`)。

테스트가 컴파일되지 않기 때문에 우리는 이 테스트를 실행할 수 없으며 `search` 함수가 아직 존재하지 않습니다! 이제 우리는 항목 12-16에서 보듯이 항상 빈 벡터를 반환하는 `search` 함수의 정의를 추가하여 컴파일과 실행하기에 충분한 코드를 추가 할 것입니다. 빈 벡터가 `"safe, fast, productive."` 줄을 포함하는 벡터와 일치하지 않기 때문에 테스트는 컴파일되지만 실패해야 합니다.

Filename: src/lib.rs

```

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}

```

항목 12-16: 우리 테스트를 컴파일 하기 위해 필요한 `search` 정의.

`search`의 선언부에는 필요한 명시적인 라이프타임 `'a`가 `contents` 인자, 그리고 반환 값과 함께 사용됩니다. 10 장에서 인자의 라이프타임으로 라이프타임 값이 매개변수로 명시된 경우 반환되는 값의 라이프타임도 연결된다고 했던 점을 상기하십시오. 이 경우 반환된 벡터는 인자로 받은 `contents`를 참조하는 문자열 조각들이 포함되어 있어야 합니다. (`query` 인자가 아니라)

다른 말로 하자면, `search` 함수로 반환되는 데이터는 `search` 함수로 전달된 `contents` 인자만큼 오래 유지될 것이라고 Rust에게 말해주는 겁니다. 이것이 중요합니다! 조각들에 의해 참조되는 데이터는 참조가 유

효한 동안 유효해야 하기 때문이죠; 만일 컴파일러가 우리가 만든 문자열 조각이 **contents**에서가 아니라 **query**에서 만들었다고 추측하면 그에 대한 안전성 검사가 제대로 수행되지 않을 겁니다.

만약 우리가 라이프타임 어노테이션을 깜빡하고 이 함수를 컴파일하려고 시도하면, 이런 에러를 얻게 될겁니다:

```
error[E0106]: missing lifetime specifier
--> src/lib.rs:5:51
   |
5 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
   |                                     ^ expected lifetime
parameter
   |
= help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `query` or `contents`
```

Rust는 두 인자 중에 우리가 필요한 쪽이 어느건지 알 수 없기 때문에, 우리가 알려줘야 합니다. **contents**가 우리의 문자들을 모두 가지고 있고 우리가 원하는 것은 그 중 일치하는 부분이기 때문에, **contents**가 라이프타임 문법을 사용하여 반환 값과 연결되어야 한다는걸 압니다.

다른 프로그래밍 언어는 인자와 반환 값을 선언부에서 연결시키라고 요구하지 않으니, 아마 이게 낯설거고, 전체적으로 좀더 쉬울겁니다. 아마 여러분은 이 예제와 10장에서 다룬 “Validating References with Lifetimes” 장의 내용을 비교하고 싶을지도 모르겠습니다.

이제 테스트를 실행해봅시다:

```
$ cargo test
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
--warnings--
  Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
  Running target/debug/deps/minigrep-abcababc

running 1 test
test test::one_result ... FAILED

failures:

---- test::one_result stdout ----
    thread 'test::one_result' panicked at 'assertion failed: `'(left ==
right)`'
left: `["safe, fast, productive."]`,
right: `[]`', src/lib.rs:48:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
  test::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

error: test failed, to rerun pass '--lib'
```

훌륭하게, 우리가 예상했던 예상대로 테스트가 실패했습니다. 테스트를 통과하게 만들어봅시다!

테스트를 통과하는 코드 작성

현재는, 우리가 늘 빈 벡터를 반환하니까 테스트가 실패하게 됩니다. 이를 수정하고 `search`를 구현하기 위해, 우리의 프로그램은 다음 단계를 따를 필요가 있습니다.

- `contents`의 각 줄에 대한 반복작업
- 해당 줄에 우리의 쿼리 문자열이 포함되어 있는지 검사
- 그렇다면, 우리가 반환할 값 목록에 추가
- 그렇지 않다면, 통과
- 일치하는 결과 목록을 반환

각 단계를 밟아가기 위해, 줄들에 대한 반복작업부터 시작합시다!

`lines` 메소드를 사용하여 줄들에 대한 반복 작업

Rust는 문자열의 줄-단위로 반복 작업을 할 수 있는 유용한 메소드가 있는데, 편리하게 이름이 `lines`이고, 항목 12-17처럼 보여주는 것처럼 동작합니다. 아직 컴파일되지 않는다는 점에 유의하세요:

Filename: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```

항목 12-17: `contents`의 각 줄마다 반복작업

`lines` 메소드는 반복자를 리턴합니다. 우리는 13장에서 반복자에 대해서 다루게 될 겁니다만, 항목 3-4에서 반복자를 사용하는 방법을 봤었다는 걸 상기시켜 드립니다. 항목 3-4에서는 반복자와 함께 `for`반복문을 사용하여 컬렉션의 각 항목에 대해 임의의 코드를 수행했습니다.

Query로 각 줄을 검색하기

다음으로 현재 줄에 쿼리 문자열이 포함되어 있는지 확인합니다. 다행스럽게도 문자열에는 유용한 '`contains`'라는 메소드가 있습니다. 항목 12-18과 같이 `search` 함수에서 `contains` 메소드에 대한 호출을 추가하십시오. 이 코드는 여전히 컴파일되지 않으니 주의하세요.

Filename: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

항목 12-18: 어느 줄이 `query` 문자열을 포함하고 있는지 보기 위한 기능 추가

일치하는 줄 보관하기

또한 쿼리 문자열이 포함된 줄을 저장할 방법이 필요합니다. 이를 위해 우리는 `for`반복문 전에 가변 벡터를 만들고 `push` 메소드를 호출하여 벡터에 `line`을 저장합니다. 항목 12-19처럼 `for`반복문이 끝난 다음에 벡터를 반환합니다.

Filename: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

항목 12-19: 일치하는 라인들을 저장하여 반환할 수 있게 만들기.

이제 `search` 함수는 `query`를 포함하는 줄들만 반환하게 되었으니 우리의 테스트는 통과되야 할 겁니다. 테스트를 실행해 봅시다:

```
$ cargo test
--snip--
running 1 test
test test::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

우리 테스트가 통과되었으니, 제대로 동작한다는 것을 알게 되죠!

이 시점에서, 우리는 동일한 기능을 유지하기 위해 테스트를 통과시키면서 `search` 함수를 리팩토링할 기회를 고려해 볼 수 있게 됐습니다. `search` 함수가 많이 나쁘지는 않지만, 반복자의 기능들이 주는 유용함을 충분히 활용하지 못하고 있습니다. 우리는 13장에서 이 예제로 돌아와 반복자에 대해서 자세히 알아보고 어떻게 개선할 수 있는지 알아볼 겁니다.

run 함수에서 search 함수를 사용하기

Using the `search` Function in the `run` Function

이제 `search` 함수는 실행되고 테스트 되었지만, 우리의 `run` 함수에서 `search`를 호출하게 해야 합니다. 우리는 `config.query` 값과 `run`으로 파일에서 읽어온 `contents`를 `search` 함수에 전달해야 합니다. 그 이후 `run`은 `search`로부터 반환된 각 줄을 출력합니다:

Filename: src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    for line in search(&config.query, &contents) {
        println!("{} {}", line);
    }

    Ok(())
}
```

우리는 아직 `search`에서 `for` 반복문을 사용해 각 줄을 반환하고 출력하고 있습니다.

이제 우리의 프로그램 전체가 동작하는 것 같습니다! 확신하기 위해, 첫째로 “frog” 단어로 Emily Dickinson의 시에서 정확히 한 줄이 반환되어야 합니다:

```
$ cargo run frog poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

좋군요! 다음으로 여러 줄에 일치할 “body” 같은 단어를 해봅시다:

```
$ cargo run body poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

그리고 마지막으로, 시의 어디서도 찾을 수 없는 단어 “monomorphization” 같은 걸 검색하면 어떤 줄도 찾을 수 없다는 걸 확인해봅시다.

```
$ cargo run monomorphization poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep monomorphization poem.txt`
```

훌륭해! 우리는 어플리케이션의 구조화를 어떻게 수행하는지에 대해 많은 것을 배우며 고전적인 도구를 우리 자체 미니 버전으로 만들어봤습니다. 또한 우리는 파일의 입력, 출력, 라이프타임, 테스팅과 커맨드라인 파싱에 대해서도 좀 알게 되었네요.

이 프로젝트를 완벽하게 하기 위해, 환경 변수를 다루고 표준 에러를 출력하는 방법을 간단히 시연하려고 하는데, 모두 커맨드라인 프로그램을 작성하는데 유용할 겁니다.

환경 변수들을 활용하기

우리는 추가 기능을 구현하여 `minigrep`을 향상시키려고 합니다. 대소문자를 구분하여 검색할지를 선택할 수 있는 기능인데, 사용자가 환경 변수를 사용하여 키고 끌 수 있게 할 수 있도록 하려 합니다. 우리는 해당 기능을 명령줄 옵션으로 구현하고 사용자가 원할때마다 해당 옵션을 기입하게 만들 수도 있지만, 대신 환경 변수를 사용하게 할 수도 있습니다. 이를 통해 사용자가 한번 환경변수를 설정하는 것을 통해 현재 터미널 세션에서 하는 모든 검색이 대소문자를 구분하게 만듭니다.

대소문자를 구분하는 `search` 함수의 실패 케이스 작성하기

우리는 새로운 `search_case_insensitive` 함수를 추가하고, 환경 변수가 적용되어 있으면 호출하고자 합니다. 우리는 TDD 절차를 따르고자 하니, 우리는 먼저 실패 테스트를 작성해야 합니다. 우리는 새 테스트를 새 `search_case_insensitive`를 위해 작성하고 예전에 작성한 테스트 `one_result`를 `case_sensitive`로 이름을 바꿔 두 테스트 간의 차이점을 명확하게 합니다. 항목 12-20에서는 이를 보여줍니다.

Filename: src/lib.rs

```

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\R
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\R
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}

```

항목 12-20: 새로운 실패 테스트를 우리가 추가할 대소문자 구분 함수를 위해 추가

우리가 예전 테스트의 `contents`도 바꿨음을 주의하세요. 우리는 “`Duct tape`”라는 대문자 D로 시작되는 새로운 문자를 추가해 대소문자 구분 시에 쿼리 “`duct`”으로는 검색되지 않도록 하였습니다. 이러한 방식으로 이전 테스트를 변경하면 이미 구현한 대소문자 구분 검색 기능을 실수로 손상시키지 않게됩니다. 이 테스트는 지금 통과해야하며 우리가 작업을 마친 이후에도 대소문자를 구분하지 않는 검색 시에 통과되어야 합니다.

대소문자를 구분하지 않는 검색을 위해 새로 추가된 테스트는 “`rUsT`”를 쿼리로 사용합니다. 우리가 추가할 함수 `search_case_insensitive`는 “`rUsT`”가 대문자 R이 포함된 “`Rust:`”에 그리고 “`Trust me.`”처럼 쿼리와 다른 경우에도 일치될 겁니다. 이건 우리가 만든 `search_case_insensitive` 함수의 실패 테스트이고, 우리가 아직 함수를 추가하지 않았기 때문에 컴파일은 실패할 겁니다. 우리는 `search`` 함수를 추가할

때와 비슷한 방식으로 빈 벡터를 반환하는 뼈대를 자유롭게 추가하면 됩니다. 항목 12-16에서 테스트의 결과와 실패를 볼 수 있습니다.

search_case_insensitive 함수 구현하기

항목 12-21에서 보여주는 `search_case_insensitive`는 `search` 함수와 거의 같습니다. 유일하게 다른 점은 `query`와 각 `line`을 소문자로 만들어 인자의 대소문자 여부와 무관하게 동일한 문자가 각 라인에 존재하는지 검사할 수 있게 만든겁니다:

Filename: src/lib.rs

```
fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

항목 12-21: `search_case_insensitive` 함수를 정의해 `query`와 `line`을 `query`와 `line`을 비교하기 전에 소문자로 변경.

첫 째, 소문자화 한 `query` 문자열을 동일한 이름을 가진 그림자 변수에 보관합니다. `to_lowercase`를 쿼리에서 호출하면 사용자의 쿼리가 “rust”, “RUST”, “Rust”, 혹은 “rUsT”인지 구분할 필요가 없어지고, 우리는 사용자 쿼리가 “rust”로 간주하고 대소문자 구문을 하지 않을 겁니다.

`to_lowercase` 호출은 기존 데이터를 참조하는 것이 아니라 새로운 데이터를 생성기 때문에 `query`는 문자열 슬라이스가 아닌 `String`입니다. 예로 들었던 쿼리 “rUsT” 문자열 slice에는 우리가 사용할 “u” 또는 “t” 소문자가 없으므로 “rust”가 포함된 새 `String`을 할당해야 합니다. 우리가 `contains` 메소드에 인자로 `query`를 전달할 때 `contains`의 선언이 문자열 slice를 인자로 받게 정의되어 있으니 앤퍼샌드(&)를 추가해야합니다.

다음으로, 우리는 각 `line`에 모두 소문자로 이뤄진 `query`가 존재하는지 검사하기 전에 `to_lowercase`를 호출합니다. 이제 `line`과 `query`를 모두 소문자로 변경했으니, 대소문자 구분없이 매치되는 문자열을 검색할 수 있습니다.

해당 구현이 테스트들을 통과하는지 한번 보시죠.

```
running 2 tests
test test::case_insensitive ... ok
test test::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

시원하게 통과했습니다. 이제 `run` 함수에서 신상 `search_case_insensitive`를 호출해보자구요. 먼저 `Config` 구조체에 검색을 시에 대소문자를 구분할지 설정 옵션을 추가부터 하구요. 근데 이 필드를 추가하면 컴파일러가 필드 값을 초기화 하지 않았다고 에러를 내게 되요.

Filename: src/lib.rs

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

우리는 불린 값을 갖는 `case_sensitive`를 추가했어요. 다음으로, 우리는 `run` 함수를 실행해서 `case_sensitive` 필드의 값을 확인한 뒤에 `search` 함수와 `search_case_insensitive` 함수 중에 어느 쪽을 호출 할 것인지 결정하면 되요, 항목 12-22처럼 말이죠. 아직도 컴파일은 안되욧!

Filename: src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

항목 12-22: `config.case_sensitive`의 값을 기준으로
`search` 혹은 `search_case_insensitive`이 호출됩니다.

마지막으로, 우리는 환경 변수를 검사해야 해요. 환경 변수를 다루기 위한 함수들은 `env` 모듈이 있는 표준 라이브러리에 있어요, 그래서 우리는 `use std::env;`을 `src/lib.rs`의 최상단에 추가해서 현재 범위로 끌어오려고 해요. 그러면 우리는 `env`에 있는 `var` 메소드를 사용하여 `CASE_INSENSITIVE`란 이름의 환경변수를 검사할 수 있죠. 항목 12-23에서 보이듯 말이에요.

Filename: `src/lib.rs`

```
use std::env;

// --snip--

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

항목 12-23: `CASE_INSENSITIVE`란 이름의 환경변수 검사하기

여기서 우리는 `case_sensitive`라는 새 변수를 만들어요. 그의 값을 설정하려고, `env::var` 함수를 호출하고 `CASE_INSENSITIVE`란 환경변수의 이름을 전달하죠. `env::var` 메소드는 `Result`를 반환하는데, 만약 환경변수가 설정된 상태라면 환경 변수의 값을 포함한 성공한 `Ok` 변형체가, 만약 설정되지 않았다면 `Err` 변형체를 반환하게 됩니다.

우리는 `Result`의 `is_err` 메소드를 에러이며 설정되지 않은 상태라서 대소문자를 구분하는 검색을 해야 하는지 확인하고자 사용합니다. 만약 `CASE_INSENSITIVE` 환경 변수에 뭐라도 설정이 되었으면, `is_err`는 `false`를 반환하고 대소문자 구분 검색을 수행하게 될겁니다. 우리는 환경변수의 내용은 신경쓰지 않고, 그저 그게 설정이 되어있는지만을, `is_err`로 검사하며 `unwrap`, `expect`나 `Result`에 존재하는 다른 메소드는 사용하지 않았어요.

항목 12-22에서 구현했던 것처럼 `case_sensitive` 변수의 값을 `Config` 인스턴스에 전달하여 `run` 함수가 해당 값을 읽고 `search_case_insensitive` 또는 `search`를 호출할지 여부를 결정할 수 있도록

합니다.

이제 돌려보죠! 처음에는 프로그램을 환경변수 설정없이 “to” 쿼리와 함께 실행하면, 소문자 “to”를 포함하는 모든 줄이 일치되게 됩니다.

```
$ cargo run to poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

잘 동작하고 있네요! 이제, 프로그램을 `CASE_INSENSITIVE`를 1로 설정하지만 쿼리는 동일한 “to”로 실행해볼까요.

PowerShell을 사용하는 경우 환경 변수를 설정하고 둘로 나눈 명령으로 프로그램을 실행해야합니다.

```
$ $env:CASE_INSENSITIVE=1
$ cargo run to poem.txt
```

대소문자 “to”가 포함된 줄을 가져와야 합니다.

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

훌륭하게, “To”가 포함 된 줄도 있습니다! 우리의 `minigrep` 프로그램은 이제 환경변수를 통해 대소문자를 구분하지 않고 검색 할 수 있습니다. 이제 커맨드라인 인수나 환경변수를 사용하여 설정 옵션을 관리하는 방법을 알게 되었네요!

일부 프로그램은 동일 설정에 대해 인수, 그리고 환경변수를 모두 허용합니다. 이 경우 프로그램은 둘 중 하나의 우선 순위를 결정합니다. 또 다른 독자 연습의 일환으로, 커맨드라인 인수와 환경변수를 통해 대소문자 구분을 제어 해보세요. 프로그램이 하나는 대소문자를 구분하고 다른 하나는 구분하지 않도록 설정되어 실행된다면 커맨드라인 인자와 환경변수 중에 어느쪽에 우선순위를 둘지 결정해보세요.

`std::env` 모듈에는 환경 변수를 다루는 데 유용한 여러 가지 기능이 있으니 사용 가능한 내용을 보려면 문서를 확인하세요.

표준출력 대신 표준에러로 에러메시지 출력하기

지금까지 우리는 모든 출력을 `println!`을 사용하여 터미널에 출력했습니다. 대부분의 터미널은 두 가지 방식의 출력을 지원합니다: 표준 출력(`stdout`)은 일반적인 정보전달용이고 표준 에러(`stderr`)는 에러 메시지용입니다. 이렇게 구분지음으로 인해 사용자는 프로그램의 출력을 직접 파일에 작성하면서도 여전히 에러메시지를 화면에 출력할 수 있습니다.

`println!` 함수는 오직 표준출력만 사용할 수 있으므로, 우리는 표준에러에 출력을 위한 다른 것을 알아보겠습니다.

에러가 어디에 출력될지 검사

먼저, `minigrep`의 출력 내용이 어떻게 표준출력에 작성되는지를 후에 우리가 표준에러로 바꾸려는 에러메시지를 염두하며 살펴봅시다. 에러가 발생할 것을 인지한채로 우리는 표준출력 스트림을 파일로 변경하고자 합니다. 표준에러 스트림은 변경하지 않을 것이므로, 표준에러로 보내진 모든 출력내용은 화면에 표시될 겁니다.

커맨드라인 프로그램들은 에러메시지들이 표준에러로 전달되는 것을 상정하고 있기 때문에 표준출력 스트림을 파일로 변경하더라도 우리는 에러메시지가 출력되는 것을 여전히 볼 수 있습니다. 우리 프로그램은 정상 동작하고 있지 않습니다 : 오류메시지 출력이 파일로 저장되고 있거든요!

이런 동작을 시연하는 방법은 프로그램의 실행시킬때 `>`과 표준출력 스트림을 향하게 할 파일이름을 주면 됩니다. 에러가 발생할 여지가 있는 인자는 주지 않습니다.

```
$ cargo run > output.txt
```

`>` 문법은 쉘에게 표준출력의 내용을 화면이 아닌 `*output.txt`에 출력하게끔 하는 것입니다. 우리가 기대했던 에러메시지의 화면 출력은 보지 못했으니 이것은 파일 마지막에 기록됐을 겁니다. 다음인 `output.txt`의 내용입니다.

```
Problem parsing arguments: not enough arguments
```

역시, 우리의 에러메시지는 표준출력으로 출력되었네요. 이런 에러메시지가 표준에러로 출력된다면 훨씬 유용하고 우리가 같은 방법으로 표준출력을 변경했을 때 오직 성공적 실행에 관련된 데이터만 저장할 수 있게 될 겁니다. 지금 바꿔봅시다.

에러를 표준에러로 출력하기

우리는 항목 12-24의 코드를 출력되는 에러메시지들을 변경하는데 사용하고자 합니다. 이번 장 진입부에서 리팩토링한 결과 모든 에러메시지는 하나의 함수 `main`에서 출력되고 있습니다. 표준라이브러리에 존재하는 표준에러에 출력해주는 매크로 `eprintln!`를 사용하여 `println!`을 사용하여 에러를 출력하던 두 부분을 `eprintln!`을 사용하도록 변경해봅시다.

Filename: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);

        process::exit(1);
    }
}
```

항목 12-24: 표준출력에 에러메시지를 출력하던 것을 `eprintln!`을 사용하여 표준에러로 변경하기

`println!`을 `eprintln!`으로 변경한 후에, 같은 방식으로 `>`을 사용해 표준출력을 변경하는 것 외에 다른 인자를 주지 않고 프로그램을 다시 실행시켜 봅시다.

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

이제 우리는 에러를 화면에서 볼 수 있고, 우리가 커맨드라인 프로그램에서 기대한 대로 `output.txt`는 비어있습니다.

이번에는 에러를 발생시키지 않게 인자와 함께 프로그램을 실행시키면서 표준출력을 파일로 변경해봅시다.

```
$ cargo run to poem.txt > output.txt
```

터미널에는 아무것도 출력되지 않고, `output.txt`가 보관하게 됩니다.

Filename: output.txt

```
Are you nobody, too?
How dreary to be somebody!
```

이번 시연은 우리가 표준출력에 성공적출력을 표준에러에 에러출력을 의도한 대로 수행하고 있음을 보여줍니다.

종합

이번 장에서는 지금까지 우리가 배웠던 몇 가지 주요 개념을 되짚어보고 Rust 문법에서 범용 I/O 작업수행을 하는 방법을 알아봤습니다. 커맨드라인 인자, 파일, 환경변수, 그리고 `eprintln!` 매크로로 에러출력을 사용하여 당신은 이제 커맨드라인 응용프로그램을 작성할 준비가 됐습니다. 이전 장들의 개념을 활용하여, 당신의 코드는 잘 구조화되고, 적합한 데이터 구조를 사용하여 효율적으로 데이터를 저장하며, 에러를 보기좋게 관리하며, 잘 테스트 할 수 있게 됐습니다.

다음으로, 우리는 함수형 언어의 영향을 받은 Rust의 기능 몇가지를 알아보겠습니다 : 클로저와 반복자.

함수형 언어의 특성들: 반복자들과 클로저들

러스트의 디자인은 많은 기존 언어들과 기술들에서 영감을 얻었으며, 중요한 영향 중에 하나는 **함수형 프로그래밍**입니다. 함수형 스타일의 프로그래밍은 자주 함수를 값처럼 인자로 넘기는 것, 다른 함수들에서 결과값으로 함수들을 돌려주는 것, 나중에 실행하기 위해 함수를 변수에 할당하는 것 등을 포함합니다. 이번 장에서는, 무엇이 함수형 프로그래밍이고 그렇지 않은지에 대해 논의하는 대신, 다른 언어에서 자주 함수형으로 언급되는 특성들과 유사한 러스트의 특성들에 대해 논의할 것입니다.

더 구체적으로, 이것들을 다룹니다:

- **클로저들**, 변수에 저장할 수 있는 함수와 유사한 구조.
- **반복자들**, 일련의 요소들을 처리할 수 있는 방법.
- 이 두가지 특성들을 사용해서 12장의 I/O 프로젝트를 향상시킬 수 있는 방법.
- 이 두 특성들의 성능 (스포일러 있음: 생각보다 빠릅니다!)

다른 장에서 다른 패턴 매칭이나 열거형과 같은 다른 러스트의 특성들도 역시 함수형 스타일의 영향을 받았습니다. 클로저들과 반복자들을 정복하는 것은 자연스러우면서도 빠른 러스트 코드를 작성하는데 중요한 부분입니다, 그래서 이번 장 전체에서 이것들을 다룹니다.

클로저: 환경을 캡처할 수 있는 익명 함수

러스트의 클로저는 변수에 저장하거나 다른 함수에 인자로 넘길 수 있는 익명 함수입니다. 한 곳에서 클로저를 만들고 다른 문맥에서 그것을 평가하기 위해 호출할 수 있습니다. 함수와 다르게 클로저는 그들이 호출되는 스코프로부터 변수들을 캡처할 수 있습니다. 이 클로저 특성이 코드 재사용과 동작 사용자 정의를 어떤 식으로 허용하는지 예를 들어 보여줄 것입니다.

클로저로 행위를 추상화 하기

클로저를 나중에 실행하기 위해 저장하는 것이 유용한 상황에 대한 예제로 작업해 봅시다. 따라가다 보면, 클로저 문법과 타입 추론, 트레이트에 대해 이야기할 것입니다.

이런 가상의 상황을 생각해 봅시다: 우리는 맞춤 운동 계획을 생성하는 앱을 만드는 스타트업에서 일합니다. 백엔드는 러스트로 작성되어 있고, 운동 계획을 생성하는 알고리즘은 앱 사용자의 나이, 체질량 지수, 선호도, 최근 운동들과 그들이 지정한 강도 숫자와 같은 많은 다른 요소들을 고려합니다. 이 예제에서 사용되는 실제 알고리즘은 중요하지 않습니다; 중요한 것은 이 알고리즘이 몇 초가 걸린다는 것입니다. 이 알고리즘을 우리가 필요할 때 한 번만 호출하기를 원하고, 그래서 사용자가 필요 이상으로 기다리지 않게 만들고 싶습니다.

우리는 리스트 13-1에 보이는 `simulated_expensive_calculation` 함수를 사용해서 이 가상의 알고리즘 호출을 실험할 것입니다. 이 함수는 `calculating slowly...`을 출력하고, 2초를 기다린 다음, 인자로 넘어온 어떤 값이든 돌려줍니다:

파일명: src/main.rs

```
use std::thread;
use std::time::Duration;

fn simulated_expensive_calculation(intensity: u32) -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

리스트 13-1: 실행시간이 2초 걸리는 가상의 계산을 대신하는 함수

다음은 이 예제에서 중요한 운동 앱의 일부를 담고 있는 `main` 함수입니다. 이 함수는 사용자가 운동 계획을 물어볼 때 앱이 호출 할 코드를 나타냅니다. 앱의 프론트엔드와의 상호작용은 클로저를 사용하기에 적합하지 않기 때문에, 우리 프로그램에 대한 입력을 나타내는 값을 코드상에 넣어두고 결과를 출력 할 것입니다.

필요한 입력들은:

- 사용자로 부터의 강도 숫자, 이것은 그들이 운동을 요청할 때 지정되며, 낮은 강도 운동을 원하는지 혹은 고강도 운동을 원하는지를 나타냅니다.
- 임의의 숫자는 몇 가지 다양한 운동 계획들을 생성할 것입니다.

결과는 추천 운동 계획이 될 것입니다. 리스트 13-2에 우리가 사용할 `main` 함수가 있습니다:

파일이름: src/main.rs

```
fn main() {  
    let simulated_user_specified_value = 10;  
    let simulated_random_number = 7;  
  
    generate_workout(  
        simulated_user_specified_value,  
        simulated_random_number  
    );  
}
```

리스트 13-2: 사용자 입력과 임의의 숫자 생성을 시뮬레이션 하기 위한 `main` 함수와 하드코딩된 값

단순함을 위해서 `simulated_user_specified_value` 변수의 값을 10으로하고 `simulated_random_number` 변수의 값을 7로 하드코딩했습니다; 실제 프로그램에서, 강도 숫자를 앱 프론트엔드에서 얻고 2장의 추리게임에서 그랬던 것처럼, 임의의 숫자 생성을 위해 `rand` 크레이트를 사용합니다. `main` 함수는 `generate_workout` 함수를 모의의 입력값으로 호출 합니다.

이제 상황이 만들어 졌으니, 알고리즘으로 넘어가겠습니다. 리스트 13-3에 있는 `generate_workout` 함수는 이 예제에서 가장 신경써야 할 앱의 비즈니스 로직을 포함하고 있습니다. 이 예제에서 나머지 코드를 변경 사항은 이 함수에 적용 됩니다:

파일이름: src/main.rs

```

fn generate_workout(intensity: u32, random_number: u32) {
    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            simulated_expensive_calculation(intensity)
        );
        println!(
            "Next, do {} situps!",
            simulated_expensive_calculation(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                simulated_expensive_calculation(intensity)
            );
        }
    }
}

```

리스트 13-3: 입력값과 `simulated_expensive_calculation` 함수 호출에 근거해서 운동 계획을 출력하는 비즈니스 로직

리스트 13-3의 코드는 느린 계산 함수에 대해 여려번 호출을 합니다. 첫번째 `if` 블럭은 `simulated_expensive_calculation` 함수를 두번 호출하고, 바깥 `else`의 안쪽에 있는 `if` 문에서는 전혀 호출하지 않으며, 두번째 `else` 문의 경우는 한번 호출 합니다.

`generate_workout` 함수의 바람직한 행위는 먼저 사용자가 저강도 운동(25보다 작은 수로 표시) 혹은 고강도 운동(25 혹은 더 큰수)을 원하는지 체크하는 것입니다.

저강도 운동 계획은 우리가 시뮬레이션 하는 복잡한 알고리즘에 근거에서 푸쉬업과 싯업의 수를 추천 할 것입니다.

사용자가 고강도 운동을 원한다면, 약간의 추가 로직이 있습니다: 앱에 의해 생성된 임의의 숫자가 30이면, 앱은 휴식과 수분 섭취를 추천합니다. 그렇지 않다면, 사용자는 복잡한 알고리즘을 기반으로 몇 분의 달리기를 안내 받을 것입니다.

데이터 과학팀은 앞으로 알고리즘 호출 방식을 일부 변경해야 한다고 알렸습니다. 이러한 변경이 발생 했을 때 업데이트를 단순화 하기 위해서, 이 코드를 리팩토링 하여 `simulated_expensive_calculation` 함수를 단지 한 번만 호출하도록 하려고 합니다. 또한 현재 프로세스에서 해당 함수에 대한 다른 호출을 추가하지 않고 불필요하게 함수를 두 번 호출하는 위치를 없애고 싶습니다. 즉, 결과가 필요없다면 함수를 호출하고 싶지 않고, 여전히 그것을 한 번만 호출하고 싶습니다.

함수를 사용해서 리팩토링 하기

우리는 여러 방향으로 운동 프로그램을 다시 구조화 할 수 있습니다. 우선, 리스트 13-4에 보이는 것처럼, 중복된 `expensive_calculation` 함수 호출을 하나의 변수로 추출 해볼 것입니다:

파일이름: src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_result =
        simulated_expensive_calculation(intensity);

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result
        );
        println!(
            "Next, do {} situps!",
            expensive_result
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result
            );
        }
    }
}
```

리스트 13-4: `simulated_expensive_calculation`에 대한 호출들을 한 곳으로 추출하고 결과를 `expensive_result` 변수에 저장하기.

이 변경은 `simulated_expensive_calculation`에 대한 모든 호출들을 하나로 합치고 첫번째 `if` 문에서 불필요하게 이 함수를 여러번 호출하던 문제를 해결 합니다. 불행하게도, 이제 모든 경우에 대해서 이 함수를 호출하고 결과를 기다리며, 이 결과를 전혀 사용하지 않는 안쪽 `if` 블럭도 해당됩니다.

우리는 프로그램에서 한곳에서 코드를 정의하고, 실제로 결과가 필요한 곳에서만 그 코드를 실행하고 싶습니다. 이것이 클로저의 유스 케이스입니다.

코드를 저장하기 위해 클로저를 사용해서 리팩토링 하기.

`if` 블럭 전에 항상 `simulated_expensive_calculation` 함수를 호출하는 대신, 리스트 13-5에 보이

는 것처럼, 클로저를 정의하고 변수에 결과를 저장하기 보단 클로저를 변수에 저장 할 수 있습니다. 여기서 소개하는 것처럼 실제로 클로저 안에 `simulated_expensive_calculation` 의 전체 내용을 옮길 수 있습니다.

Filename: src/main.rs

```
let expensive_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

리스트 13-5: 클로저를 정의하고 `expensive_closure` 변수에 저장하기

클로저 정의는 변수 `expensive_closure` 에 그것을 할당하기 위해 `=` 다음에 옵니다. 클로저를 정의하기 위해, 수직의 파이프 (`|`) 한쌍으로 시작하며, 그 사이에 클로저에 대한 파라미터를 기술합니다; 이 문법은 스몰토크와 루비에서 클로저 정의와의 유사성 때문에 선택 되었습니다. 이 클로저는 `num` 이라는 하나의 파라미터를 갖습니다: 하나 이상의 파라미터를 갖는다면, `|param1, param2|` 와 같이 콤마로 구분합니다.

파라미터들 다음에, 클로저의 바디를 포함하는 중괄호를 넣습니다—클로저 바디가 하나의 표현식이라면 이것은 선택적입니다. 중괄호 다음에 클로저의 끝에는 `let` 문을 완성하기 위해 세미콜론이 필요합니다. 클로저 바디에서 마지막 줄로부터 반환되는 값인 (`num`) 은 그것이 호출되었을 때 클로저로 부터 반환되는 값이 될 것입니다, 왜냐하면 그 줄은 함수 본문처럼 세미콜론으로 끝나지 않기 때문입니다.

`let` 문은 `expensive_closure` 가 익명함수의 정의를 포함하며, 익명함수를 호출한 결과 값을 포함하지 않는다는 것에 유의 하세요. 우리가 클로저를 사용하는 이유는 호출할 코드를 한 곳에서 정의하고, 그 코드를 저장하며, 이후 다른 곳에서 그것을 호출하길 원하기 때문이라는 것을 상기하세요; 우리가 호출하고자 하는 코드가 이제 `expensive_closure` 에 저장되었습니다.

클로저를 정의하면서, 저장된 코드를 실행하고 결과값을 얻기 위하여 `if` 블록 안의 코드를 클로저 호출 방식으로 변경할 수 있습니다. 우리는 함수를 호출하는 것처럼 클로저를 호출 합니다: 리스트 13-6에 보이는 것처럼, 클로저 정의를 갖고 있는 변수명을 쓰고 다음엔 사용할 인자값을 포함하는 괄호가 따라 옵니다:

파일명: src/main.rs

```

fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_closure(intensity)
            );
        }
    }
}

```

리스트 13-6: 우리가 정의한 `expensive_closure` 호출하기

이제 비용이 큰 계산은 단 한곳에서만 호출 되고, 우리가 결과가 필요한 곳에서만 그 코드를 실행 합니다.

그러나, 리스트 13-3에 있는 문제중 하나를 다시 소개합니다: 우리는 여전히 첫번째 `if` 블럭에서 클로저를 두번 호출하는데, 이는 비용이 큰 코드를 두번 호출하고 사용자가 실행시간 만큼 긴시간을 두번 기다리게 합니다. 우리는 그 `if` 블럭안에 클로저 호출의 결과를 저장하는 로컬 변수를 만들어서 그 문제를 해결할 수 있지만, 클로저는 다른 해결책을 제공합니다. 우리는 그 해결책에 대해 조금 이야기할 것입니다. 그러나 우선 클로저 정의에 탑입 어노테이션이 없는 이유와 클로저와 연관된 트레잇에 대해 이야기 합시다.

클로저 탑입 추론과 어노테이션

클로저는 `fn` 함수처럼 파라미터나 반환값의 탑입을 명시할 것을 요구하지 않습니다. 탑입 어노테이션은 사용자에게 노출되는 명시적인 인터페이스의 일부이기 때문에 함수에 필요 합니다. 이 인터페이스를 엄격하게 정의하는 것은 함수가 어떤 탑입의 값을 사용하고 반환하는지에 대해 모두가 합의 한다는 것을 보장하는데 중요 합니다. 그러나 클로저는 이와 같이 노출된 인터페이스에 사용되지 않습니다: 변수에 저장되고 이름없이

우리의 라이브러리 사용자들에게 노출되지 않고 사용 됩니다.

추가적으로, 클로저는 보통 짧고 임의의 시나리오 보다 좁은 문맥 안에서만 관련이 있습니다. 이런 제한된 문맥 안에서만, 컴파일러는 안정적으로 파라미터와 리턴타입을 추론할 수 있으며, 이는 대부분의 변수 타입을 추론 할 수 있는 방법과 비슷 합니다.

프로그래머들에게 이런 작고 익명의 함수들에 타입을 달도록하는 것은 짜증나고 컴파일러가 이미 사용할수 있는 정보와 대개는 중복 됩니다.

변수처럼, 엄밀하게 필요한 것 이상으로 자세히 표현하는 비용을 지불하고서라도 명확성과 명료성을 높이고 싶다면 타입 어노테이션(혹은 타입 명시)를 추가할 수 있습니다; 리스트 13-4 에 정의한 클로저에 타입을 명시하는 것은 리스트 13-7 에 보이는 것과 같을 것입니다:

파일명: src/main.rs

```
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

리스트 13-7: 클로저에 파라미터와 반환값 타입에 대한 선택적 인 타입 어노테이션 추가하기

타입 어노테이션이 있으면 클로저와 함수의 문법은 더 비슷해 보입니다. 다음은 파라미터에 1을 더하는 함수 정의와 동일한 행위를 하는 클로저를 수직으로 비교한 것입니다. 관련 있는 부분들을 정렬하기 이해 약간의 공백을 추가했습니다. 이것은 파이프를 사용하는 것과 선택적인 문법의 양을 제외하고 클로저 문법과 함수 문법이 얼마나 비슷한지 보여줍니다:

```
fn add_one_v1(x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x| { x + 1 };
let add_one_v4 = |x| x + 1;
```

첫번째 줄은 함수 정의를 보여주고, 두번째 줄은 타입을 모두 명기한 클로저 정의를 보여 줍니다. 세번째 줄은 클로저 정의에서 타입 어노테이션을 지웠고, 네번째 줄은 선택적인 중괄호를 지웠는데, 클로저 보디가 단 하나의 표현식을 갖기 때문입니다. 이것은 모두 호출 했을 때 동일한 행위를 수행하는 유효한 정의들입니다.

클로저 정의는 각 파라미터들과 그들의 반환값에 대해 단 하나의 추론된 구체적인 타입을 갖을 것입니다. 예를 들면, 리스트 13-8 은 파라미터로 받은 값을 그대로 반환하는 짧은 클로저의 정의를 보여줍니다. 이 클로저는 이 예제의 목적 이에외는 유용하지 않습니다. 정의에 타입 어노테이션을 추가하지 않았다는 것에 유의하세요: 클로저를 두번 호출하는데, 첫번째는 `String` 을 인자로 사용하고 두번째는 `u32` 을 사용한다면 에러가 발생합니다:

파일명: src/main.rs

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

리스트 13-8: 두개의 다른 타입으로 추론된 타입을 갖는 클로저 호출 해보기

컴파일러는 이런 에러를 줍니다:

```
error[E0308]: mismatched types
--> src/main.rs
|
| let n = example_closure(5);
|           ^ expected struct `std::string::String`, found
integral variable
|
= note: expected type `std::string::String`
        found type `{integer}`
```

처음 `String` 값으로 `example_closure` 을 호출하면, 컴파일러는 `x` 의 타입과 클로저의 반환 타입을 `String` 으로 추론합니다. 이 타입들은 그다음에는 `example_closure` 에 있는 클로저에 고정되고, 같은 클로저를 다른 타입으로 사용하려고 할 때 타입 에러를 얻게 됩니다.

제네릭 파라미터와 Fn 트레잇을 사용하여 클로저 저장하기

운동 생성 앱으로 돌아갑시다. 리스트 13-6에서, 우리의 코드는 아직도 비용이 큰 계산을 하는 클로저를 필요한 것 보다 더 많이 호출 합니다. 이 문제를 풀기위한 한가지 옵션은 비싼 비용의 클로저 결과를 재활용을 위해 변수에 저장하고 결과가 필요한 부분에서 클로저를 다시 호출하는 대신 그 변수를 사용하는 것입니다. 그러나, 이 방법은 많은 반복된 코드를 만들 수 있습니다.

운 좋게도, 다른 해결책이 있습니다. 우리는 클로저와 클로저를 호출한 결과값을 갖고 있는 구조체를 만들 수 있습니다. 그 구조체는 결과값을 필요로 할 때만 클로저를 호출 할 것이며, 결과값을 캐시에 저장해 두어 우리의 나머지 코드에서 결과를 저장하고 재사용 하지 않아도 되도록 할 것입니다. 이 패턴을 **메모이제이션** (*memoization*) 혹은 *지연 평가(lazy evaluation)*로 알고 있을 것 입니다.

구조체에서 클로저를 갖고 있도록 하기 위해, 클로저 타입을 기술 할 필요가 있는데, 구조체 정의는 각 필드의 타입을 알 필요가 있기 때문입니다. 각 클로저 인스턴스는 자신의 유일한 익명 타입을 갖습니다: 즉, 두 클로저가 동일한 타입 서명을 갖더라도 그들의 타입은 여전히 다른 것으로 간주 됩니다. 클로저를 사용하는 구조체, 열거형, 함수 파라미터를 정의하기 위해, 10장에서 설명한 것처럼 제네릭과 트레이트 바운드를 사용합니다.

`Fn` 트레잇은 표준 라이브러리에서 제공 합니다. 모든 클로저들은 다음 트레잇 중 하나를 구현 합니다: `Fn`, `FnMut`, 혹은 `FnOnce`. 환경을 캡처하는 것에 대한 다음 절에서 이 트레잇들의 차이점들에 대해 설명할 것입니다; 이 예제에서, `Fn` 트레잇 을 사용할 수 있습니다.

클로저가 이 트레잇 바운드에 맞춰야 하는 파라미터와 반환값의 타입을 표현하기 위해 `Fn` 트레잇 바운드에 타입을 추가 합니다. 이 경우, 클로저는 파라미터 타입이 `u32` 이고 `u32` 타입을 반환하므로, 명시하는 트레잇 바운드는 `Fn(u32) -> u32` 입니다.

리스트 13-9 는 `Cacher` 구조체의 정의를 보여주는데 클로저와 선택적인 반환값을 갖고 있습니다:

파일명: src/main.rs

```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```

리스트 13-9: `calculation` 에 클로저를 담고, 선택적인 결과 를 `value` 에 담는 `Cacher` 구조체 정의 하기

`Cacher` 구조체는 제너릭 타입 `T` 의 `calculation` 필드를 갖습니다. `T` 에 대한 트레잇 바운드는 `Fn` 트레잇을 사용하여 그것이 클로저라는 것을 기술 합니다. `calculation` 필드에 저장하고자 하는 클로저는 하나의 `u32` 타입 파라미터 (`Fn` 다음에 괄호안에 명시됨)를 갖고 `u32` (`->` 다음에 명시됨) 타입의 값을 반환해야 합니다.

노트: 함수는 세개의 `Fn` 트레잇도 모두 구현 합니다. 환경에서 값을 캡처할 필요 가 없다면, `Fn` 트레잇을 구현한 어떤것을 필요로 하는 곳에 클로저 대신 함수를 사용할 수 있습니다.

`value` 필드는 `Option<u32>` 타입입니다. 클로저를 실행하기 전에는 `value` 는 `None` 일 것입니다. `Cacher` 를 사용하는 코드에서 클로저의 결과를 요청할 경우, `Cacher` 는 그 때 클로저를 실행하고 결과를 `Some` variant 에 넣어서 `value` 필드에 저장 할 것입니다. 그 다음에는 코드에서 클로저의 결과를 다시 요청하면 클로저를 다시 실행하는 대신, `Cacher` 는 `Some` variant 안에 있는 결과를 돌려줄 것입니다.

방금 설명한 `value` 필드에 대한 로직은 리스트 13-10 에 정의되어 있습니다:

파일명: src/main.rs

```
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}
```

리스트 13-10: `Cacher` 의 캐싱 로직

우리는 이 필드에 있는 값을 호출하는 코드에서 잠재적으로 변경하도록 두기보다 `Cacher` 가 구조체 필드의 값을 관리하도록 하고 싶기 때문에, 이 필드는 비공개 (private)입니다.

`Cacher::new` 함수는 제네릭 파라미터 `T` 를 받는데, `Cacher` 구조체와 동일한 트레이트 바운드를 갖도록 정의 되었습니다. 그 다음 `Cacher::new` 는 `calculation` 필드에 명시된 클로저를 포함하고 클로저를 아직 실행한적이 없기 때문에 `value` 필드가 `None` 값을 갖는 `Cacher` 인스턴스를 반환 합니다.

호출하는 코드에서 클로저를 평가한 결과값을 원할때, 클로저를 직접 호출하기보다, `value` 메서드를 호출 할 것입니다. 이 메서드는 이미 `self.value` 에 결과값을 `Some` 으로 갖고 있는지 체크 합니다; 만약 그렇다면 클로저를 다시 실행하는 대신 `Some` 안에 있는 값을 반환 합니다.

만약 `self.value` 가 `None` 이라면, `self.calculation` 에 저장된 클로저를 호출하고, 나중에 재사용하기 위해 결과를 `self.value` 에 저장한 다음 그 값을 반환 합니다.

리스트 13-11 는 리스트 13-6 에 있는 `generate_workout` 함수에서 이 `Cacher` 구조체를 사용하는 방법을 보여줍니다:

파일명: src/main.rs

```

fn generate_workout(intensity: u32, random_number: u32) {
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result.value(intensity)
            );
        }
    }
}

```

리스트 13-11: 캐싱 로직을 추상화 하기 위해 `generate_workout` 함수 안에서 `Cacher` 사용하기

클로저를 변수에 직접 저장하는 대신, 클로저를 갖는 `Cacher`의 새 인스턴스를 저장했습니다. 그리고는, 결과가 필요한 각 위치에 `Cacher` 인스턴스의 `value` 메소드를 호출했습니다. 우리는 `value` 메소드를 원하는 만큼 많이 호출할 수 있고, 전혀 호출하지 않을 수도 있으며, 비싼 비용의 계산은 최대 한 번만 수행 될 것입니다.

리스트 13-2의 `main` 함수로 이 프로그램을 실행해 보세요. 다양한 `if` 와 `else` 블럭에 있는 모든 케이스들을 검증하기 위해 `simulated_user_specified_value` 와 `simulated_random_number` 변수들을 변경해 보면, `calculating slowly...` 메세지는 필요할 때 단지 한 번만 나타납니다. `Cacher`는 필요한 것 보다 더 많이 비싼 비용의 계산을 호출하지 않도록 보장하는 필요한 로직을 처리해서, `generate_workout` 가 비즈니스 로직에 집중하도록 해줍니다.

Cacher 구현의 제약사항

값을 캐싱하는 것은 일반적으로 유용한 동작이기 때문에 이와는 다른 클로저를 사용 해서 우리 코드의 다른

부분에서 적용하고 싶을 수도 있습니다. 그러나 현재 **Cacher** 구현은 다른 문맥에서 다르게 재사용하기에는 두 가지 문제가 있습니다.

첫 번째 문제는 **Cacher** 인스턴스가 **value** 메소드의 **arg** 파라미터에 대해 항상 같은 값을 얻는다는 가정을 한다는 것입니다. 즉, 이 **Cacher** 테스트는 실패 할 것 입니다:

```
#[test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);

    assert_eq!(v2, 2);
}
```

이 테스트는 인자로 받은 값을 그대로 돌려주는 클로저가 포함된 새로운 **Cacher** 인스턴스를 생성 합니다. **arg** 값을 1로 그리고 **arg** 값을 2로 해서 이 **Cacher** 인스턴스의 **value** 메소드를 호출하고, **arg** 값을 2로 **value** 를 호출 했을 때 2를 반환 할 것으로 기대 합니다.

리스트 13-9 와 13-10 에 있는 **Cacher** 구현에 대해 이 테스트를 돌리면, 테스트는 이 메세지와 함께 **assert_eq!** 에서 실패 할 것입니다:

```
thread 'call_with_different_values' panicked at 'assertion failed: `(left == right)`
  left: `1`,
  right: `2`', src/main.rs
```

문제는 처음 **c.value** 을 1로 호출 했을 때, **Cacher** 인스턴스는 **self.value** 에 **Some(1)** 을 저장 합니다. 그 후에, **value** 값으로 무엇을 넘기던, 항상 1을 반환 할 것입니다.

Cacher 이 하나의 값보다 해시맵을 사용하도록 수정해 봅시다. 해시맵의 키는 넘겨받은 **arg** 값이 될 것이고, 해시맵의 값은 그 키로 클로저를 호출한 결과가 될 것입니다. **self.value** 가 **Some** 혹은 **None** 값인지 직접 살펴보는 대신, **value** 함수는 해시맵의 **arg** 값을 살펴보고 값이 있으면 반환 할 것입니다. 값이 없으면, **Cacher** 는 클로저를 호출해서 해당 **arg** 값과 연관된 해시맵에 결과값을 저장 할 것입니다.

현재 **Cacher** 구현의 두 번째 문제는 **u32** 타입 파라미터 한 개만 받고 하나의 **u32** 을 반환한다는 것입니다. 예를 들면, 문자열 슬라이스를 넘겨주고 **usize** 값을 반환하는 클로저의 결과를 캐시에 저장하고 싶을 수도 있습니다. 이 이슈를 수정 하기 위해, **Cacher** 기능에 유연성을 높여주도록 더 중립적인 파라미터를 사용해 봅시다.

클로저로 환경 캡처 하기

운동 생성 예제에서, 우리는 클로저를 단지 인라인 익명 함수로 사용 했습니다. 그러나 클로저는 함수에 없는 추가적인 능력을 갖고 있습니다: 환경을 캡처해서 클로저가 정의된 스코프의 변수들을 접근할 수 있습니다.

`equal_to_x` 변수에 저장된 클로저가 클로저를 둘러싼 환경에 있는 `x` 변수를 사용하는 예제가 리스트 13-12에 있습니다:

파일명: src/main.rs

```
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;

    let y = 4;

    assert!(equal_to_x(y));
}
```

리스트 13-12: 둘러싼 범위에 있는 변수를 참조하는 클로저의 예

비록 `x` 가 `equal_to_x` 의 파라미터 중에 하나가 아니더라도, `equal_to_x` 는 `equal_to_x` 가 정의된 동일한 스코프에 정의된 `x` 변수를 사용하는 것이 허용 됩니다.

함수로는 이와 동일하게 할 수 없습니다; 다음 예제로 시도해 보면, 코드는 컴파일 되지 않습니다:

Filename: src/main.rs

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }

    let y = 4;

    assert!(equal_to_x(y));
}
```

에러가 발생 합니다:

```
error[E0434]: can't capture dynamic environment in a fn item; use the || {
...
} closure form instead
--> src/main.rs
|
4 |     fn equal_to_x(z: i32) -> bool { z == x }
|
```

컴파일러는 이것은 클로저에서만 동작한다고 상기시켜 주기까지 합니다!

클로저가 그것의 환경에서 값을 캡처할 때, 클로저 바디에서 사용하기 위해 그 값을 저장하기 위한 메모리를 사용 합니다. 이 메모리 사용은 환경을 캡처하지 않는 코드를 실행하길 원하는 더 흔한 상황에서는 지불하기 싫지 않은 오버헤드입니다. 왜냐하면 함수는 그들의 환경을 캡처할 수 없기 때문에, 함수를 정의하고 사용하는데 결코 이런 오버헤드는 발생하지 않을 것이기 때문입니다.

클로저는 세가지 방식으로 그들의 환경에서 값을 캡처 할 수 있는데, 함수가 파라미터를 받는 세가지 방식과 직접 연결 됩니다: 소유권 받기, 불변으로 빌려오기, 가변으로 빌려오기. 이것들은 다음과 같이 세개의 **Fn** 트레잇으로 표현 합니다:

- **FnOnce** 는 클로저의 환경으로 알고 있는, 그것을 둘러싼 환경에서 캡처한 변수들을 소비합니다. 캡처한 변수를 소비하기 위해, 클로저는 이 변수의 소유권을 가져야 하고 그것이 정의될 때 클로저 안으로 그것들을 옮겨와야 합니다. 이름의 일부인 **Once** 는 그 클로저가 동일한 변수들에 대해 한번이상 소유권을 얻을수 없다는 사실을 의미하며, 그래서 한 번만 호출 될 수 있습니다.
- **Fn** 은 그 환경으로 부터 값들을 불변으로 빌려 옵니다.
- **FnMut** 값을 가변으로 빌려오기 때문에 그 환경을 변경할 수 있습니다.

우리가 클로저를 만들때, 러스트는 클로저가 환경에 있는 값을 어떻게 사용하는지에 근거 해서 어떤 트레잇을 사용할지 추론 합니다. 리스트 13-12 에서, `equal_to_x` 클로저의 바디에서는 `x` 에 있는 값을 읽기만 하면 되기 때문에 클로저는 `x` 를 불변으로 빌려 옵니다. (그래서 `equal_to_x` 은 **Fn** 트래잇입니다)

만약 클로저가 환경으로부터 사용하는 값에 대해 소유권을 갖도록 강제하고 싶다면, 파라미터 리스트 앞에 **move** 키워드를 사용할 수 있습니다. 이 기법은 클로저를 다른 쓰레드로 넘길때 데이터를 이동시켜 새로운 쓰레드가 소유하도록 할때 대부분 유용 합니다.

16장에 병렬성에 대해 이야기 하는 부분에서 더 많은 **move** 클로저에 대한 예제가 있습니다. 지금은 리스트 13-12 의 코드에서 클로저에 **move** 키워드를 추가하고 정수 대신 벡터를 사용하도록 했는데, 정수는 이동되지 않고 복사되기 때문입니다; 이 코드는 아직 컴파일 되지 않습니다:

파일명: src/main.rs

```
fn main() {
    let x = vec![1, 2, 3];

    let equal_to_x = move |z| z == x;

    println!("can't use x here: {:?}", x);

    let y = vec![1, 2, 3];

    assert!(equal_to_x(y));
}
```

아래와 같은 에러가 발생합니다:

```
error[E0382]: use of moved value: `x`
--> src/main.rs:6:40
|
4 |     let equal_to_x = move |z| z == x;
   |                     ----- value moved (into closure) here
5 |
6 |     println!("can't use x here: {:?}", x);
   |                         ^ value used here after move
|
= note: move occurs because `x` has type `std::vec::Vec<i32>`, which does
not
    implement the `Copy` trait
```

move 키워드를 추가했기 때문에 클로저가 정의될 때 **x** 값은 클로저 안으로 이동됩니다. **x**의 소유권은 클로저가 갖게 되었고, **main**은 더 이상 **println!** 문에서 **x** 사용하도록 허용되지 않습니다. **println!**를 삭제하면 이 예제는 수정 됩니다.

Fn 트레이트 바운드 중 하나를 기술할 때 대부분의 경우, **Fn**으로 시작해보면 컴파일러는 클로저 바디에서 무슨 일을 하는지에 근거해서 **FnMut** 혹은 **FnOnce**이 필요한지 말해 줍니다.

클로저가 그들의 환경을 캡처할 수 있는 상황을 표현하는 것은 함수 파라미터로써 유용 합니다. 다음 주제로 넘어가 봅시다: 반복자.

반복자로 일련의 항목들 처리하기

반복자 패턴은 일련의 항목들에 대해 순서대로 어떤 작업을 수행할 수 있도록 해줍니다. 반복자는 각 항목들을 순회하고 언제 시퀀스가 종료될지 결정하는 로직을 담당 합니다. 반복자를 사용하면, 저런 로직을 다시 구현할 필요가 없습니다.

리스트에서, 반복자는 *게으른데*, 항목들을 사용하기 위해 반복자를 소비하는 메서드를 호출하기 전까지 반복자는 아무런 동작을 하지 않습니다. 예를 들면, 리스트 13-13 의 코드는 `Vec` 에 정의된 `iter` 메서드를 호출함으로써, 벡터 `v1` 에 있는 항목들에 대한 반복자를 생성 합니다. 이 코드 자체로는 어떤 유용한 동작을 하진 않습니다.

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
```

리스트 13-13: 반복자 생성하기

일단 반복자를 만들면, 다양한 방법으로 사용할 수 있습니다. 3장의 리스트 3-5 에서, 각 항목에 대해 어떤 코드를 수행하기 위해 `for` 루프에서 반복자를 사용 했습니다만, 지금까지 `iter` 에 대한 호출이 무엇을 했는지 대충 넘어 갔었습니다.

리스트 13-14 의 예제는 `for` 루프에서 반복자를 사용하는 부분에서 반복자 생성을 분리 했습니다. 반복자는 `v1_iter` 변수에 저장되고, 그 시점에 순회는 발생하지 않습니다. `v1_iter` 에 있는 반복자를 사용하는 `for` 루프가 호출되면, 루프 순회 마다 반복자의 각 요소가 사용되는데, 각각의 값을 출력 합니다.

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
for val in v1_iter {
    println!("Got: {}", val);
}
```

리스트 13-14: `for` 루프에서 반복자 사용하기

표준 라이브러리에서 반복자를 제공하지 않는 언어에서는, 변수를 인덱스 0으로 시작해서, 그 변수로 벡터를 색인해서 값을 가져오는데 사용하며, 루프안에서 벡터에 있는 아이템의 총 갯수까지 그 변수를 증가시키는 방식으로 동일한 기능을 작성할 수 있습니다.

반복자는 그러한 모든 로직을 대신 처리 하며, 잠재적으로 엄망이 될 수 있는 반복적인 코드를 줄여 줍니다. 반복자는 벡터처럼 색인할 수 있는 자료구조 뿐만 아니라, 많은 다른 종류의 시퀀스에 대해 동일한 로직을 사용할 수 있도록 더 많은 유연성을 제공 합니다. 반복자가 어떻게 그렇게 하는지 살펴 봅시다.

Iterator 트레이트와 `next` 메서드

모든 반복자는 표준 라이브러리에 정의된 `Iterator`라는 이름의 트레이트를 구현합니다. 트레이트의 정의는 아래와 같습니다:

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    // methods with default implementations elided
}
```

이 정의는 몇 개의 새로운 문법을 사용하는 것에 유의하세요: `type Item` 과 `Self::Item`은 이 트레이트와 연관 타입을 정의 합니다. 우리는 19장에서 연관 타입에 대해 자세히 이야기 할 것입니다. 지금 당장 알아야 할 것은 이 코드가 `Iterator` 트레이트를 구현하는 것은 `Item` 타입을 정의하는 것 또한 요구하며, 이 `Item` 타입이 `next` 메서드의 리턴 타입으로 사용된다는 것을 나타낸다는 것입니다. 다른 말로, `Item` 타입은 반복자로 부터 반환되는 타입이 될 것입니다.

`Iterator` 트레이트는 단지 구현자가 하나의 메서드를 정의하도록 요구 합니다: `next` 메서드입니다. 이 메서드는 반복자의 하나의 항목을 `Some`에 넣어서 반환하고, 반복자가 종료되면 `None`을 반환 합니다.

반복자의 `next` 메서드를 직접 호출할 수 있습니다; 리스트 13-15는 벡터로 부터 생성된 반복자에 대해 반복된 `next` 호출이 어떤 값들을 반환하는지 보여줍니다:

Filename: src/lib.rs

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

리스트 13-15: 반복자의 `next` 메서드 호출하기

`v1_iter`가 변경 가능하도록 만들 필요가 있다는 것에 유의하세요: 반복자에 대해 `next` 메서드를 호출하면 시퀀스의 어디에 있는지 추적하기 위해 반복자가 사용하는 내부 상태를 변경합니다. 다른 말로, 이 코드는 반복자를 소비합니다, 혹은 다 써 버립니다. `next`에 대한 각 호출은 반복자로 부터 하나의 항목을 소비

합니다. `for` 루프를 사용할 때는 `v1_iter` 를 변경할 수 있도록 만들 필요가 없는데, 루프가 `v1_iter` 의 소유권을 갖고 내부적으로 변경 가능하도록 만들기 때문입니다.

`next` 호출로 얻어온 값들은 벡터 안에 있는 값들에 대한 불변 참조라는 점 역시 유의하세요. `iter` 메서드는 불변 참조에 대한 반복자를 만듭니다. 만약 `v1` 의 소유권을 갖고 소유된 값을 반환하도록 하고 싶다면, `iter` 대신 `into_iter` 를 호출해야 합니다. 비슷하게, 가변 참조에 대한 반복자를 원한다면, `iter` 대신 `iter_mut` 을 호출할 수 있습니다.

반복자를 소비하는 메서드들

`Iterator` 트레이트에는 표준 라이브러리에서 기본 구현을 제공하는 다수의 다른 메서드들이 있습니다; `Iterator` 트레이트에 대한 표준 라이브러리 API 문서를 살펴 보면, 이 메서드들을 찾을 수 있습니다. 이 메서드들 중 일부는 그들의 구현에서 `next` 메서드를 호출하는데, 이것이 `Iterator` 트레이트를 구현할 때 `next` 메서드를 구현해야만 하는 이유입니다.

`next` 를 호출하는 메서드들을 **소비하는 어댑터들** 이라고 하는데, 그들을 호출하면 반복자를 써버리기 때문입니다. `sum` 메서드가 하나의 예인데, 반복자의 소유권을 가져오고 반복적으로 `next` 를 호출해서 순회함으로써 반복자를 소비 합니다. 순회해 나가면서 누적합계에 각 아이템을 더하고 순회가 완료되면 합계를 반환합니다. 리스트 13-16 은 `sum` 메서드의 사용을 보여주는 테스트입니다:

Filename: src/lib.rs

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

리스트 13-16: 반복자의 모든 항목에 대한 합계를 얻기 위해 `sum` 메서드 호출 하기

`sum` 은 호출한 반복자의 소유권을 갖기 때문에, `sum` 을 호출한 후 `v1_iter` 은 사용할 수 없습니다.

다른 반복자를 생성하는 메서드들

`Iterator` 트레이트에 정의된 다른 메서드들 중에 **반복자 어댑터들**로 알려진 메서드들은 반복자를 다른 종

류의 반복자로 변경하도록 허용 합니다. 복잡한 행위를 수행하 기 위해 읽기 쉬운 방법으로 반복자 어댑터에 대한 여러개의 호출을 연결할 수 있습 니다. 하지만 모든 반복자는 게으르기 때문에, 반복자 어댑터들로 부터 결과를 얻기 위해 소비하는 메서드들 중 하나를 호출 해야 합니다.

리스트 13-17 은 반복자 어댑터 메서드인 `map` 을 호출하는 예를 보여주는데, 새로운 반복자를 생성하기 위
해 각 항목에 대해 호출할 클로저를 인자로 받습니다. 여기서 클로저는 벡터의 각 항목에서 1이 증가된 새로
운 반복자를 만듭니다. 그러나, 이 코드는 경고를 발생 합니다:

Filename: src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];
v1.iter().map(|x| x + 1);
```

리스트 13-17: 새로운 반복자를 만들기 위해 반복자 어댑터 `map` 호출 하기

경고 메세지는 이것 입니다:

```
warning: unused `std::iter::Map` which must be used: iterator adaptors are
lazy
and do nothing unless consumed
--> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
= note: #[warn(unused_must_use)] on by default
```

리스트 13-17 의 코드는 아무것도 하지 않습니다; 인자로 넘긴 클로저는 결코 호출 되지 않습니다. 경고는 이
유를 알도록 해줍니다: 반복자 어댑터는 게으르고, 반복자를 여기서 소비할 필요가 있다.

이것을 고치고 반복자를 소비하기 위해, `collect` 메서드를 사용할 것인데, 12장의 리스트 12-1 에서
`env::args` 와 함께 사용했습니다. 이 메서드는 반복자를 소비하고 결과값을 수집 데이터 타입으로 모읍니
다.

리스트 13-18 에서, 벡터에 대한 `map` 호출로 부터 반환된 반복자를 순회하면서 결과를 모읍니다. 이 벡터는
각 항목이 원본 벡터로 부터 1씩 증가된 상태로 될 것 입니다.

Filename: src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];  
  
let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();  
  
assert_eq!(v2, vec![2, 3, 4]);
```

리스트 13-18: 새로운 반복자를 만들기 위해 `map` 메서드를 호출하고, 새로운 반복자를 소비하고 벡터를 생성하기 위해 `collect` 메서드 호출 하기

`map` 은 클로저를 인자로 받기 때문에, 각 항목에 대해 수행하기를 원하는 어떤 연산도 기술할 수 있습니다. 이것은 `Iterator` 트레이트가 제공하는 반복자 행위를 재사용하면서 클로저가 어떻게 일부 행위를 맞춤 조작할 수 있는지를 보여주는 굉장한 예제입니다.

환경을 캡처하는 클로저 사용하기

이제 반복자를 소개했으니, `filter` 반복자 어댑터를 사용해서 환경을 캡처하는 클로저의 일반적인 사용을 보여줄 수 있습니다. 반복자의 `filter` 메서드는 반복자로 부터 각 항목을 받아 Boolean 을 반환하는 클로저를 인자로 받습니다. 만약 클로저가 `true` 를 반환하면, 그 값은 `filter` 에 의해 생성되는 반복자에 포함될 것입니다. 클로저가 `false` 를 반환하면, 결과로 나오는 반복자에 포함되지 않을 것입니다.

리스트 13-19에서, `Shoe` 구조체 인스턴스들의 컬렉션을 순회하기 위해 `filter` 와 그 환경으로 부터 `shoe_size` 변수를 캡처하는 클로저를 사용합니다. 그것은 기술된 크기의 신발들만 반환 할 것입니다.

Filename: src/lib.rs

```

#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter()
        .filter(|s| s.size == shoe_size)
        .collect()
}

#[test]
fn filters_by_size() {
    let shoes = vec![
        Shoe { size: 10, style: String::from("sneaker") },
        Shoe { size: 13, style: String::from("sandal") },
        Shoe { size: 10, style: String::from("boot") },
    ];
    let in_my_size = shoes_in_my_size(shoes, 10);

    assert_eq!(
        in_my_size,
        vec![
            Shoe { size: 10, style: String::from("sneaker") },
            Shoe { size: 10, style: String::from("boot") },
        ]
    );
}

```

리스팅 13-19: `shoe_size` 를 캡처하는 클로저와 `filter` 메서드 사용하기

`shoes_in_my_size` 함수는 파라미터로 신발들의 벡터에 대한 소유권과 신발 크기를 받습니다. 그것은 지정된 크기의 신발들만을 포함하는 벡터를 반환 합니다.

`shoes_in_my_size` 의 구현부에서, 벡터의 소유권을 갖는 반복자를 생성하기 위해 `into_iter` 를 호출 합니다. 그 다음 그 반복자를 클로저가 `true` 를 반환한 요소들만 포함하는 새로운 반복자로 바꾸기 위해 `filter` 를 호출 합니다.

클로저는 환경에서 `shoe_size` 매개 변수를 캡처하고, 지정된 크기의 신발만 유지하면서 각 신발의 크기와 값을 비교합니다. 마지막으로, `collect` 를 호출하면 적용된 반복자에 의해 리턴된 값을 함수가 리턴한 벡터로 모으게됩니다.

테스트는 `shoes_in_my_size` 를 호출 했을 때, 지정된 값과 동일한 사이즈를 갖는 신발들만 돌려받는다는 것을 보여 줍니다.

Iterator 트레이트로 자신만의 반복자 만들기

벡터에 대해 `iter`, `into_iter` 혹은 `iter_mut` 을 호출해서 반복자를 생성할 수 있다는 것을 보았습니다. 해시맵과 같은 표준 라이브러리에 있는 다른 컬렉션 타입으로부터 반복자를 생성할 수 있습니다. 자신만의 타입에 대해 `Iterator` 트레이트를 구현함으로써 원하는 동작을 하는 반복자를 생성하는 것 역시 가능합니다. 이전에 언급했던 것처럼, 정의를 제공해야 하는 유일한 메서드는 `next` 메서드입니다. 그리고 나면, `Iterator` 트레이트에서 제공하는 기본구현을 갖는 다른 모든 메서드를 사용할 수 있습니다!

이것을 보여주기 위해 1부터 5까지 셀 수 있는 반복자를 만듭니다. 우선, 어떤 값을 유지하는 구조체를 만들 것입니다. 그 다음 `Iterator` 트레이트를 구현하고 그 구현에서 값을 사용함으로써 이 구조체를 반복자로 만들 것입니다.

리스트 13-20에는 `Counter` 구조체의 정의와 `Counter` 인스턴스를 생성하는 연관된 `new` 함수가 있습니다:

Filename: src/lib.rs

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

리스트 13-20: `Counter` 구조체와 `count`의 초기값 0으로 `Counter`의 인스턴스를 생성하는 `new` 함수 정의하기

`Counter` 구조체는 `count`라는 이름의 하나의 필드를 갖습니다. 이 필드는 `u32` 타입의 값을 갖는데 1부터 5까지 순회하는데 어디까지 진행했는지를 추적할 것입니다. `count` 필드는 `Counter` 구현이 그 값을 관리하기 원하기 때문에 외부로 노출되지 않습니다. `new` 함수는 항상 새로운 인스턴스가 `count` 필드에 0을 담은 채로 시작하도록 강제합니다.

다음으로, 이 반복자가 사용될 때 우리가 원하는 것을 지정하기 위해 `next` 메서드의 본문을 정의함으로써 `Counter` 타입에 대한 `Iterator` 특성을 구현할 것입니다, 리스트 13-21 참조:

Filename: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

리스트 13-21: Counter 구조체에 대해 Iterator 트레이트 구현하기

우리의 반복자를 위해 연관된 `Item` 타입을 `u32`로 지정했는데, 이는 반복자가 `u32` 값을 반환한다는 것을 의미 합니다. 다시, 아직 연관 타입에 대해 걱정하시 마세요, 19장에서 다를 것입니다.

우리는 우리의 반복자가 현재 상태에 1을 더하길 원합니다, 그래서 `count`를 0으로 초기화 했고 처음엔 1을 반환할 것입니다. `count`의 값이 6 보다 작다면, `next`는 `Some`으로 포장된 현재 값을 리턴할 것이며, `count`가 6 이거나 더 크다면, 우리의 반복자는 `None`을 반환할 것입니다.

Counter 반복자의 `next` 메서드 사용하기

`Iterator` 트레이트를 구현 했다면, 반복자를 갖게 됩니다! 리스트 13-22는 리스트 13-15에서 벡터로 부터 생성된 반복자에 했던 것 처럼, `Counter` 구조체에 직접 `next` 메서드를 호출 함으로써 반복자 기능을 사용할 수 있다는 것을 보여주는 테스트를 보여 줍니다.

Filename: src/lib.rs

```
#[test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

리스트 13-22: `next` 메서드 구현의 기능 테스트

이 테스트는 `counter` 변수에 새로운 `Counter` 인스턴스를 생성하고 `next` 를 반복적으로 호출하면서, 이 반복자가 우리가 원하는 행위를 구현했다는 것을 검증 합니다: 1 부터 5까지의 값을 반환함.

다른 `Iterator` 메서드들 사용하기

우리는 `next` 메서드를 정의함으로써 `Iterator` 트레이트를 구현했습니다, 그래서 표준 라이브러리에 정의된 `Iterator` 트레이트 메서드들의 기본 구현을 사용할 수 있는데, 그들은 모두 `next` 메서드의 기능을 사용하기 때문입니다.

예를 들면, 만약 어떤 이유에서든 `Counter` 인스턴스에 의해 생성된 값들을 얻고, 다른 `Counter` 인스턴스에 의해 생성된 값과 쌍을 이루며, 각 쌍을 함께 곱하고, 3으로 나눠지는 값들만 유지하며, 모든 결과 값을 함께 더하고 싶다면, 리스트 12-23 의 테스트에서 보여지는 것처럼, 그렇게 할 수 있습니다:

Filename: src/lib.rs

```
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
        .map(|(a, b)| a * b)
        .filter(|x| x % 3 == 0)
        .sum();
    assert_eq!(18, sum);
}
```

리스트 13-23: `Counter` 반복자에 대해 `Iterator` 트레이트의 다양한 메서드 사용하기

`zip` 은 단지 네 개의 쌍을 생성한다는데 유의 하세요; 이론적으로 다섯번째 쌍인 `(5, None)` 은 결코 생성되지 않는데, `zip` 은 입력 반복자 중 하나라도 `None` 을 반환하면 `None` 을 반환하기 때문입니다.

우리가 `next` 메서드가 어떻게 동작하는지에 대해 기술했기 때문에 이 모든 메서드 호출이 가능하며, 표준 라이브러리는 `next` 를 호출하는 다른 메서드들의 기본 구현을 제공 합니다.

I/O 프로젝트 개선하기

반복자에 대한 새로운 지식을 사용하여 12장의 I/O 프로젝트의 코드들을 더 깔끔하고 간결하게 개선할 수 있습니다. 반복자를 사용하여 어떻게 `Config::new` 함수와 `search` 함수의 구현을 개선할 수 있는지 살펴봅시다.

반복자를 사용하여 `clone` 제거하기

리스트 12-6에서, `String` 값의 슬라이스를 받고 슬라이스를 인덱싱하고 복사 함으로써 `Config` 구조체의 인스턴스를 생성하였고, `Config` 구조체가 이 값을 소유하도록 했습니다. 리스트 13-24에서는 리스트 12-23에 있던 것처럼 `Config::new` 함수의 구현을 다시 재현 했습니다:

파일명: src/lib.rs

```
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

리스트 13-24: 리스트 12-23의 `Config::new` 함수 재현

그 당시, 비효율적인 `clone` 호출에 대해 걱정하지 말라고 얘기 했으며 미래에 없앨 것이라고 했습니다. 자, 그때가 되었습니다!

`String` 요소들의 슬라이스를 `args` 파라미터로 받았지만 `new` 함수는 `args`를 소유하지 않기 때문에 `clone`이 필요했습니다. `Config` 인스턴스의 소유권을 반환하기 위해, `Config`의 `query` 와 `filename` 필드로 값을 복제 함으로써 `Config` 인스턴스는 그 값을 소유할 수 있습니다.

반복자에 대한 새로운 지식으로, 인자로써 슬라이스를 빌리는 대신 반복자의 소유권을 갖도록 `new` 함수를 변경할 수 있습니다. 슬라이스의 길이를 체크하고 특정 위치로 인덱싱을 하는 코드 대신 반복자의 기능을 사용할 것 입니다. 이것은 반복자가 값에 접근 할 것이기 때문에 `Config::new` 함수가 무엇을 하는지를 명확하게 해줄 것 입니다.

`Config::new` 가 반복자의 소유권을 갖고 빌린 값에 대한 인덱싱을 사용하지 않게 된다면, `clone` 을 호출하고 새로운 할당을 만드는 대신 `String` 값들을 반복자에서 `Config` 로 이동할 수 있습니다.

반환된 반복자를 직접 사용하기

I/O 프로젝트의 `src/main.rs` 파일을 열어보면, 아래와 같을 것 입니다:

파일명: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

우리는 리스트 12-24 에 있는 `main` 함수의 시작점을 리스트 13-25 에 있는 코드로 바꿀 것입니다. 이것은 `Config::new` 도 업데이트 해야 컴파일 됩니다.

파일명: `src/main.rs`

```
fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

리스트 13-25: `Config::new` 로 `env::args` 의 반환값 넘기기

`env::args` 함수는 반복자를 반환 합니다! 반복자의 값을 벡터로 모아서 `Config::new` 에 슬라이스를 넘기는 대신, `env::args` 에서 반환된 반복자의 소유권을 `Config::new` 로 직접 전달 합니다.

그 다음, `Config::new` 정의를 업데이트 할 필요가 있습니다. I/O 프로젝트의 `src/lib.rs` 파일에서, 리스트 13-26 처럼 `Config::new` 의 시그니처를 변경 합시다. 함수 본문을 업데이트 해야 하기 때문이 아직 컴파일 되지 않습니다.

파일명: `src/lib.rs`

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        // --snip--
```

리스트 13-26: 반복자를 받도록 `Config::new` 의 시그니처 업데이트 하기

`env::args` 함수에 대한 표준 라이브러리 문서에는 반환하는 반복자의 타입이 `std::env::Args`라고 명시되어 있습니다. `Config::new` 함수의 시그니처를 업데이트 해서 `args` 파라미터가 `&[String]` 대신 `std::env::Args` 타입을 갖도록 했습니다. `args`의 소유권을 갖고 그것을 순회하면서 `args`를 변경할 것이기 때문에, 변경 가능하도록 하기 위해 `args` 파라미터의 명세에 `mut` 키워드를 추가 할 수 있습니다.

인덱싱 대신 **Iterator** 트레이트 메서드 사용하기

다음으로, `Config::new` 의 본문을 수정 할 것입니다. 표준 라이브러리 문서에는 `std::env::Args`의 `Iterator` 트레이트를 구현하고 있다는 것 역시 언급하고 있으므로, `next` 메서드를 호출 할 수 있다는 것을 알 수 있습니다! 리스트 13-27은 리스트 12-23의 코드에서 `next` 메서드를 사용하도록 변경 합니다:

Filename: src/lib.rs

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };

        let filename = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file name"),
        };

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

리스트 13-27: 반복자 메서드들을 사용하도록 `Config::new` 의 본문 변경하기

`env::args` 반환값의 첫번째 값은 프로그램 이름이라는 것을 명심하세요. 우리는 첫번째 값을 무시하고 그

다음 값을 얻기 위해 우선 `next` 를 호출한 다음, 그 반환값으로 아무것도 하지 않았습니다. 두번째로, `Config` 의 `query` 에 원하는 값을 넣기 위해 `next` 를 호출 했습니다. `next` 가 `Some` 을 반환하면, 값을 추출하기 위해 `match` 를 사용 합니다. 만약 `None` 을 반환하면, 이것은 충분한 인자가 넘어오지 않았다는 것을 의미하고, `Err` 값과 함께 조기 반환이 합니다. `filename` 값도 동일하게 처리 합니다.

반복자 어댑터로 더 간결한 코드 만들기

I/O 프로젝트의 `search` 함수에도 반복자의 장점을 활용할 수 있습니다. 리스트 12-19 의 코드가 리스트 13-28 에 재현되어 있습니다:

파일명: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

리스트 13-28: 리스트 12-19 의 `search` 함수 구현

우리는 반복자 어댑터 메서드를 사용해서 이 코드를 더 간결한 방식으로 작성할 수 있습니다. 이렇게 함으로써 `results` 벡터가 변경 가능한 중간 상태를 갖는 것을 피할 수 있습니다. 함수형 프로그래밍 스타일은 더 깔끔한 코드를 만들기 위해 변경 가능한 상태의 양을 최소화 하는 것을 선호 합니다. 가변 상태를 제거하면 `results` 벡터에 대한 동시 접근을 관리 할 필요가 없기 때문에, 추후에 검색을 병렬로 수행하는 것과 같은 향상이 가능해 집니다. 리스트 13-29 는 이 변경을 보여줍니다:

파일명: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents.lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

리스트 13-29: `search` 함수 구현에서 반복자 어댑터 메서드 사용하기

`search` 함수의 목적은 `query` 를 포함하는 `contents` 의 모든 줄을 반환하는 것임을 기억하세요. 리스트 13-19 의 `filter` 예제와 유사하게, 이 코드는 `line.contains(query)` 이 `true` 를 반환하는 줄들만 유지하기 위해 `filter` 어댑터를 사용합니다. 그리고나서 `collect` 를 통해서 일치하는 줄들을 모아 새로운 벡터로 만듭니다. 훨씬 단순합니다! `search_case_insensitive` 도 역시 반복자 메서드들을 사용하도록 같은 변경을 자유롭게 만들어 보세요.

다음 논리적 질문은 당신의 코드에서 어떤 스타일을 선택하는 것이 좋은지와 그 이유입니다: 리스트 13-28 의 최초 구현 혹은 리스트 13-29 의 반복자를 사용하는 버전. 대부분의 러스트 프로그래머는 반복자 스타일을 선호 합니다. 처음 사용하기는 다소 어렵습니다만, 다양한 반복자 어댑터와 어떤 일을 하는지에 대해 한번 감이 온다면, 반복자들은 이해하기 쉬워질 것 입니다. 루핑과 새로운 벡터 생성과 같은 다양한 작업을 수행하는 대신, 코드는 루프의 고차원적 목표에 집중 합니다. 이것은 아주 흔한 코드의 일부를 추상화해서 제거함으로써 반복자의 각 요소가 반드시 통과 해야하는 필터링 조건과 같이 이 코드에 유일한 개념을 더 쉽게 볼 수 있도록 합니다.

그러나 두 구현은 정말 동일 할까요? 직관적으로 저수준의 루프가 더 빠르다고 가정할 수도 있습니다. 그럼 성능에 대해서 얘기해 봅시다.

성능 비교하기: 루프 vs. 반복자

루프와 반복자 중에 어떤것을 사용할지 결정하기 위해, 어떤 버전의 `search` 함수가 더 빠른지 알 필요가 있습니다: 명시적으로 `for` 루프를 사용한 버전과 반복자를 사용한 버전.

우리는 아서 코난 도일이 쓴 *셜록 홈즈의 모험*의 전체 내용을 로딩하고 내용중에 *the* 를 찾는 벤치마크를 돌렸습니다. 여기 `search` 루프와 반복자를 사용한 버전에 대한 벤치마크 결과가 있습니다:

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

반복자 버전이 약간더 빠릅니다! 여기서 벤치마크 코드에 대해 설명하진 않을 것 입니다. 왜냐하면 핵심은 두 버전이 동등하다는 것을 증명하는 것이 아니고, 이 두 구현 방법이 성능 측면에서 어떻게 다른지에 대한 상식적인 이해를 얻는 것이기 때문입니다.

더 포괄적인 벤치마크를 위해, 다양한 크기의 다양한 텍스트를 `내용` 으로 사용하고, 다른 길이의 다른 단어들을 `질의어` 로 사용해서 모든 종류의 다른 조합을 확인 하는 것이 좋습니다. 핵심은 이렇습니다: 반복자는 비록 고수준의 추상이지만, 컴파일 되면 대략 직접 작성한 저수준의 코드와 같은 코드 수준으로 내려갑니다. 반복자는 러스트의 *제로 비용 추상화* 중 하나이며, 그 추상을 사용하는 것은 추가적인 실행시간 오버헤드가 없다는 것을 의미 합니다. 최초의 C++ 디자이너 이자 구현자인 비야네 스트롭스터룹이 “Foundations of C++” (2012) 에서 *제로 오버헤드* 를 정의한 것과 유사 합니다:

일반적으로, C++ 구현은 제로-오버헤드 원리를 따릅니다: 사용하지 않는 것은, 비용을 지불하지 않습니다. 그리고 더 나아가: 사용하는 것은, 더 나은 코드를 제공할 수 없습니다.

다른 예로, 다음 코드는 오디오 디코더에서 가져왔습니다. 디코딩 알고리즘은 이전 샘플의 선형 함수에 기반해서 미래의 값을 추정하기 위해 선형 예측이라는 수학적 연산을 사용합니다. 이 코드는 반복자 체인을 사용해서 스코프에 있는 세 개의 변수로 수학 연산을 합니다: 데이터의 `buffer` 슬라이스, 12 개의 `coefficients` 배열, 그리고 데이터를 쉬프트 하기 위한 `qlp_shift` 값. 이 예제에서 변수를 선언 했지만 값은 주지 않았습니다; 이 코드는 이 문맥밖에서는 크게 의미가 없지만, 러스트가 어떻게 고수준의 개념을 저수준의 코드로 변환하는지에 대한 간결하고 실제적인 예제입니다.

```

let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
        .zip(&buffer[i - 12..i])
        .map(|(&c, &s)| c * s as i64)
        .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}

```

`prediction`의 값을 계산하기 위해, 이 코드는 `coefficients`에 있는 12개의 값을 순회하면서 각각의 계수와 `buffer`의 이전 12개의 값의 쌍을 만들기 위해 `zip` 메서드를 사용 합니다. 그런 다음, 각 쌍에 대해 값을 모두 곱하고 모든 결과를 더한 후 더한 값을 `qlp_shift` 비트 만큼 우측으로 쉬프트 합니다.

오디오 디코더와 같은 어플리케이션에서의 계산은 종종 성능에 가장 높은 우선순위를 둡니다. 여기서 우리는 두 개의 어댑터를 사용하는 반복자를 생성하고 값을 소비 했습니다. 이 러스트 코드가 컴파일 되면 어떤 어셈블리 코드가 될까요? 글쎄요, 이 글을 쓰는 시점에선 그것은 직접 손으로 작성한 것과 같은 어셈블리 코드로 컴파일 됩니다. 거기엔 `coefficients`의 값을 순회하기 위한 어떤 루프도 없습니다: 러스트는 12개의 순회가 있다는 것을 알고 있으며, 루프를 "풀어(unrolls)" 놓습니다. 언롤링(Unrolling)은 루프 제어 코드의 오버헤드를 제거하고 대신 루프의 각 순회에 해당하는 반복적인 코드를 생성하는 최적화 방법입니다.

모든 계수들은 레지스터에 저장되는데 값에 대한 접근이 매우 빠르다는 것을 뜻합니다. 실행시간에 배열 접근에 대한 경계 체크가 없습니다. 러스트가 적용할 수 있는 이런 모든 최적화들은 결과 코드를 아주 효율적으로 만듭니다. 이제 이것을 알게 되었으니, 반복자와 클로저를 공포없이 사용할 수 있습니다! 이것들은 코드를 고수준으로 보이도록 하지만, 그렇게 하기 위해 실행시간 성능 저하를 만들지 않습니다.

요약

클로저와 반복자는 함수형 프로그래밍 아이디어에서 영감을 받은 러스트의 특징들입니다. 이것들은 고수준의 개념을 저수준의 성능으로 명확하게 표현할 수 있는 러스트의 능력에 기여하고 있습니다. 클로저와 반복자의 구현들은 런타임 성능에 영향을 미치지 않습니다. 이것은 제로-비용 추상을 제공하기 위해 노력하는 러스트의 목표 중의 일부입니다.

이제 I/O 프로젝트의 표현력을 개선 했으니, 프로젝트를 세상과 공유하는데 도움을 줄 `cargo`의 몇몇 특징들을 살펴 봅시다.

Cargo 와 Crates.io 더 알아보기

지금까지 우린 빌드, 실행, 코드 테스트등 Cargo 의 가장 기본적인 기능만 사용하였지만, Cargo 는 훨씬 더 많은 일을 할 수 있습니다. 이번 장에서 다음 목록의 기능을 수행하는 고급 기능 몇가지를 알아보도록 하겠습니다.

- 릴리즈 프로필을 이용해 빌드 커스터마이징하기
- [crates.io](#) 에 라이브러리 배포하기
- 대규모 작업을 위한 작업공간 구성하기
- [crates.io](#) 에서 바이너리 설치하기
- 커스텀 명령어로 Cargo 확장하기

Cargo 는 이번 장에서 다루는 것보다 더 많은 일을 할 수 있습니다. 만약 Cargo 의 모든 기능에 대한 설명을 보고 싶으시다면 [Cargo 공식 문서](#) 를 참고하세요.

릴리즈 프로필을 이용해 빌드 커스터마이징하기

러스트에서 릴리즈 프로필(*release profiles*)은 프로그래머가 코드 컴파일에 관련된 여러가지 옵션을 제어할 수 있도록 다양한 구성으로 사전 정의되고 커스텀 가능한 프로필입니다. 각 프로필은 다른 프로필과 독립적으로 설정됩니다.

Cargo는 두 메인 프로필을 가집니다: 여러분이 `cargo build`를 실행할 때 쓰는 `dev` 프로필과 `cargo build --release`를 실행할 때 쓰는 `release` 프로필입니다. `dev` 프로필은 개발에 적합한 설정을 기본값으로 갖고, `release` 프로필은 릴리즈 빌드용 설정을 기본값으로 가집니다.

여러분은 빌드 출력에서 이 프로필들의 이름을 몇 번 보셨을 수도 있습니다.

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
$ cargo build --release
    Finished release [optimized] target(s) in 0.0 secs
```

위 출력의 `dev` 와 `release` 는 컴파일러가 다른 프로필을 사용한다는 것을 나타냅니다.

Cargo는 프로젝트의 *Cargo.toml* 파일에 `[profile.*]` 구획이 따로 없을 때 적용되는 각 프로필의 기본 설정을 가지고 있습니다. 이때 여러분은 원하는 프로필에 `[profile.*]` 구획을 추가하여 기본 설정을 덮어 씌울 수 있습니다. 여기 예시로 `dev` 와 `release` 프로필 각각의 `opt-level` 기본 설정 값을 보여드리겠습니다.

Filename: *Cargo.toml*

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

`opt-level` 설정은 러스트가 여러분의 코드에 적용할 최적화 수치이며, 0 ~ 3 사이의 값을 가집니다. 여러분이 개발을 할 때와 같이 코드를 자주 컴파일 하는 상황에서는 코드의 실행 속도가 조금 느려지는 한이 있더라도 컴파일이 빨리 되길 원합니다. 하지만 높은 최적화 수치를 적용 할 수록 컴파일에 걸리는 시간은 증가합니다. 따라서 `dev` 의 기본 `opt-level` 값은 `0` 으로 되어 있습니다. 만약 여러분이 코드를 릴리즈 하려 한다면, 컴파일에 걸리는 시간이 늘어나도 상관이 없을 겁니다. 릴리즈 할 경우 컴파일은 한 번이지만, 실행 횟수는 여러분이니까요. 따라서 릴리즈 모드에서는 컴파일 시간을 희생하는 대신 빠른 코드 실행 속도를 얻기 위해 `release` 프로필의 기본 `opt-level` 값이 `3` 으로 되어 있습니다.

이전에 말했듯, 여러분은 *Cargo.toml* 에 다른 값을 넣어서 기본 설정을 덮어 씌울 수 있습니다. 예를 들어 만

약 우리가 개발용 프로필에 0 이 아닌 1 의 최적화 수치를 적용하고 싶다면 우리 프로젝트의 *Cargo.toml* 에 다음 두 줄을 추가하면 됩니다:

Filename: Cargo.toml

```
[profile.dev]
opt-level = 1
```

이 코드는 기본 설정인 0 을 덮어 씌웁니다. 이후에 우리가 `cargo build` 를 실행하면 Cargo 는 dev 프로필의 기본값과 우리가 커스텀 한 `opt-level` 을 사용합니다. 우리가 `opt-level` 을 1 로 설정 했기 때문에 Cargo 는 릴리즈 빌드 만큼은 아니지만 기본 설정 보다 많은 최적화를 진행할 겁니다.

각 프로필의 설정 옵션 및 기본값의 전체 목록을 보시려면 [Cargo 공식 문서](#) 를 참고해 주시기 바랍니다.

Crates.io 에 크레이트 배포하기

우린 crates.io 의 패키지를 프로젝트의 의존성으로만 사용했지만 여러분이 직접 여러분의 패키지를 배포 (publish)해서 코드를 다른 사람들과 공유 할 수도 있습니다. crates.io 의 크레이트 등기소 (registry)는 여러분이 만든 패키지의 소스코드를 배포하므로, crates.io 는 주로 오픈 소스인 코드를 관리합니다.

러스트와 Cargo 는 여러분이 배포한 패키지를 사람들이 더 쉽게 찾고 사용할 수 있도록 도와주는 기능이 있습니다. 다음 내용이 바로 이런 기능들 몇개에 대한 설명과 패키지를 배포하는 방법에 대한 설명입니다.

유용한 문서화 주석 만들기

여러분의 패키지를 시간을 들여서 자세하게 문서화하는 작업은 굉장히 가치있는 일입니다. 문서는 다른 사람들이 그 패키지를 언제, 어떻게 써야할지 알게 해주는데 굉장히 도움이 되거든요. 3장에서 우린 슬래시 두 개 (//) 를 이용해 러스트 코드에 주석을 남기는 법을 배웠습니다만, 러스트에는 문서화 주석(*documentation comment*) 이라고 불리는 문서화를 위한 특별한 주석이 존재합니다. 이 주석은 HTML 문서를 생성할 수 있는데, 이 HTML 에는 여러분의 크레이트가 어떻게 구현되었는지/가 아닌 어떻게 사용하는지/에 관심 있는 프로그래머들을 위한 공개 API의 문서화 주석이 보여집니다.

문서화 주석은 슬래시 두 개가 아니라 세 개(///) 를 이용하며 텍스트 서식을 위한 마크다운 표기법을 지원 합니다. 문서화 주석은 문서화할 대상 바로 이전에 배치하면 됩니다. Listing 14-1 은 my_crate 크레이트의 add_one 함수에 대한' 문서화 주석의 예시를 보여줍니다:

Filename: src/lib.rs

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
/// assert_eq!(6, my_crate::add_one(5));
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Listing 14-1: 함수에 대한 문서화 주석

자, add_one 함수가 무슨 일을 하는지 설명을 적었고 Example 절에서 add_one 함수를 어떻게 사용하는지에 대한 예시 코드를 제공 했습니다. 이제 우린 cargo doc 을 이용해 이 문서화 주석으로부터 HTML

문서를 생성할 수 있습니다. 이 명령어는 러스트에 들어있는 `rustdoc` 툴을 실행시키고 생성된 HTML 문서를 `target/doc` 디렉토리에 저장합니다.

좀 더 편리하게, `cargo doc --open` 을 실행시키면 여러분의 현재 크레이트의 문서에 대해 (심지어 여러분의 크레이트가 가진 모든 디펜던시의 문서까지) HTML 을 생성하고 웹 브라우저에 띄워줄 겁니다. 이제 `add_one` 함수를 찾아보면 여러분은 문서화 주석의 내용이 어떻게 나타나는지 보실 수 있습니다. Figure 14-1 처럼요:



Figure 14-1: `add_one` 함수에 대한 HTML 문서화

자주 사용되는 구절

우린 Listing 14-1에서 HTML에 "Examples." 제목을 가진 구절을 만들기 위해 `# Examples` 마크다운 헤더를 사용했습니다. 이외에 크레이트의 제작자가 일반적으로 문서에 사용하는 구절은 다음과 같습니다.

- **Panics:** 문서화된 기능이 패닉을 일으킬 수 있는 시나리오입니다. 함수를 호출하는 사람들에게 "프로그램이 패닉을 일으키지 않게 하려면 이러한 상황에서는 이 함수를 호출하지 않아야 합니다"라는 내용을 알려줍니다.
- **Errors:** 해당 함수가 `Result` 를 반환할 경우에는 발생할 수 있는 에러의 종류와 해당 에러들이 발생하는 조건을 설명해 주어서 호출하는 사람이 여러 에러를 여러 방법으로 처리할 수 있도록 해야합니다.
- **Safety:** 함수가 `안전하지 않을(unsafe)` 경우에 (19장에서 다루는 내용입니다) 왜 이 함수가 안전하지 않은지와 이 함수가 호출하는 사람에게 지키길 기대하는 불변성에 대해 알려주는 구절이 있어야 합

니다.

대부분의 문서화 주석은 이 구절들이 모두 필요하진 않습니다. 하지만 여러분의 코드를 사용하는 사람들이 관심을 가지고 알아보게 될 측면에 대해 곱씹어 보게 만드는 좋은 체크리스트가 될 수 있습니다.

테스트로서의 문서화 주석

여러분의 문서화 주석에 예시 코드를 추가하는 건 여러분의 라이브러리를 어떻게 사용하는지 알려줄 수 있을 뿐더러 또 다른 효과도 있습니다: 무려 `cargo test` 를 실행하면 여러분의 문서에 들어있던 예시 코드들이 테스트로서 실행됩니다! 백문이 불여일견이라는 말이 있듯이, 예시를 포함한 문서보다 좋은 문서는 없습니다. 다만, 코드를 변경하고 문서를 업데이트하지 않아서 예시 코드가 작동하지 않는 일은 절대 있어선 안되니 주의하세요. 우리가 Listing 14-1 의 `add_one` 함수에 대한 문서로 `cargo test` 를 실행하면 다음과 같은 테스트 결과를 보실수 있습니다.

```
Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

이제 우리가 함수나 예제를 변경하고 예시 코드에서 패닉이 발생하는 상태로 `cargo test` 를 실행하면, 문서 테스트 기능이 더이상 예시 코드가 기능하지 못한다고 알려줄 겁니다.

주석을 포함하는 항목을 문서화 하기

문서화 주석의 또 다른 스타일로 `///!` 가 있습니다. 이는 주석 뒤에 오는 항목을 문서화 하는게 아닌 주석을 포함하는 항목을 문서화 합니다. 일반적으로 크레이트의 루트 파일 (관례적으로 `src/lib.rs` 입니다)이나 크레이트 혹은 모듈 전체를 문서화하는 모듈 내부에 이 문서화 주석을 작성합니다.

예시로, 만약 `add_one` 함수를 포함한 `my_crate` 크레이트를 설명하기 위한 목적으로 문서화를 진행한다면, Listing 14-2 처럼 `src/lib.rs` 에 `///!` 로 시작하는 문서화 주석을 추가할 수 있습니다.

Filename: `src/lib.rs`

```
//! # My Crate
//!
//! `my_crate` is a collection of utilities to make performing certain
//! calculations more convenient.

/// Adds one to the number given.
// --snip--
```

Listing 14-2: `my_crate` 크레이트 전체를 위한 문서화

`//!`로 시작하는 줄 중 마지막 줄에 코드가 뒤따르지 않는다는 점을 주목하세요. 우린 주석 뒤에 따라오는 항목이 아닌, 주석을 포함하는 항목을 문서화 할 것이기에 `///` 가 아니라 `//!`로 시작하는 주석을 사용했습니다. 이 경우, 주석을 포함하는 항목은 크레이트의 루트 파일인 `src/lib.rs`이며 주석은 전체 크레이트를 설명하게 됩니다.

`cargo doc --open` 을 실행하면, Figure 14-2 처럼 `my_crate` 문서 첫 페이지 내용 중 크레이트의 공개 아이템들 상단에 이 주석의 내용이 표시될 것입니다.

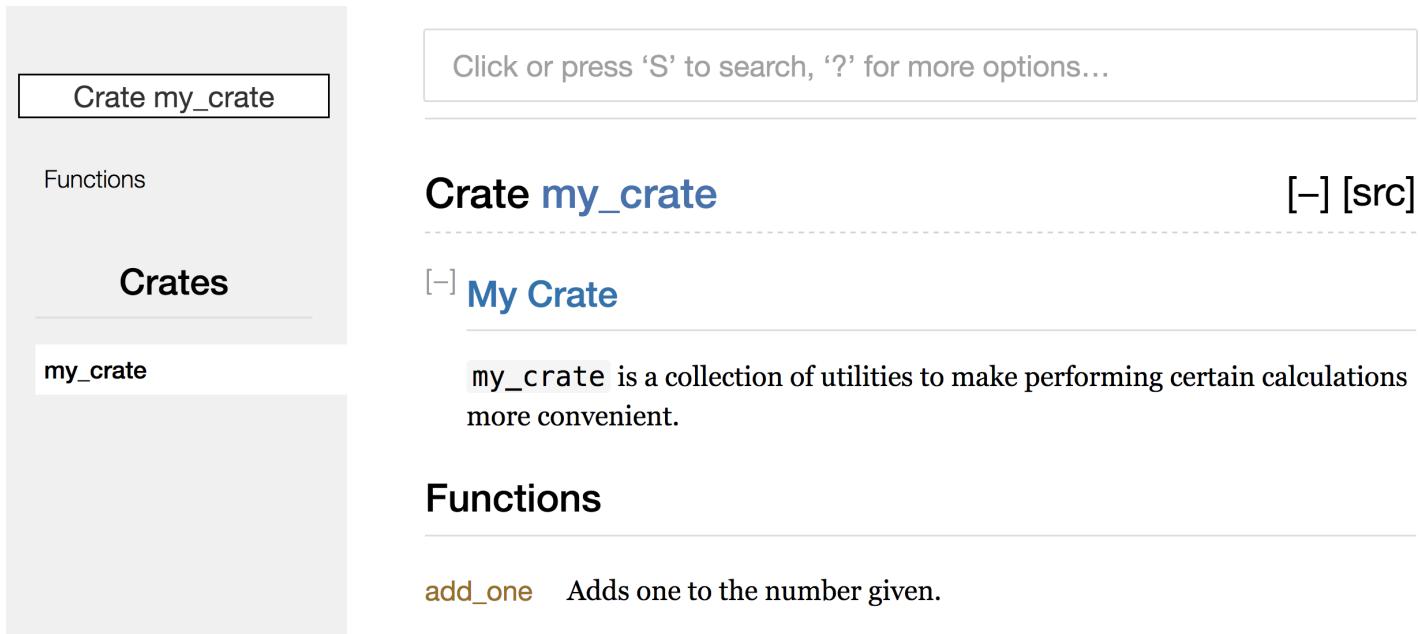


Figure 14-2: 전체 크레이트를 설명하는 주석이 포함된 `my_crate` 의 문서가 렌더링된 모습

항목 내 문서화 주석은 크레이트나 모듈을 설명하는데 유용합니다. 이를 이용해 사용자들이 크레이트의 구조를 이해할 수 있도록 크레이트의 중심 목적을 설명하세요.

pub use 를 이용해 공개 API 를 편리한 형태로 export 하기

7장에서 우린 `mod` 키워드를 이용해 우리 코드를 체계화 하는 법과, `pub` 키워드로 공개 아이템을 만드는

법, `use` 를 이용해 스코프 내로 가져오는 법을 다뤘습니다. 다만 여러분이 크레이트를 개발할 때 만들어놓은 구조는 여러분의 크레이트를 사용할 사용자들에게는 그다지 편리하지 않을 수 있습니다. 여러분은 여러 단계의 계층 구조를 이용해 크레이트를 구성하고 싶으시겠지만, 여러분이 계층 구조상에서 깊은 곳에 정의한 타입을 다른 사람들이 사용하기에는 상당히 어려움을 겪을 수 있습니다. 애초에 그런 타입이 존재하는지 알아내는 것 조차 힘들테니까요. 또한 알아내더라도 `use my_crate::UsefulType;` 가 아니라 `use my_crate::some_module::another_module::UsefulType;` 를 입력하는 일은 꽤나 짜증이 날 테죠.

공개 API 의 구조는 크레이트를 배포하는데 있어서 중요한 고려사항 중 하나입니다. 여러분의 크레이트를 이용할 사람들은 해당 구조에 있어서 여러분보다 이해도가 떨어질 것이고, 만약 여러분의 크레이트가 거대한 구조로 되어 있다면 자신들이 원하는 부분을 찾기조차 힘들 겁니다.

좋은 소식은 여러분이 만든 구조가 다른 라이브러리에서 이용하는데 편리하지 않다고 해서 굳이 내부 구조를 뒤엎을 필요는 없다는 겁니다. 대신에 여러분은 `pub use` 를 이용해 내부 항목을 다시 export(*re-export*) 하여 기존의 private 구조와 다른 public 구조를 만들 수 있다는 겁니다. 다시 export 한다는 것은 한 위치에서 공개 항목(public item)을 가져오고 이것을 마치 다른 위치에서 정의한 것처럼 공개 항목으로 만드는 것을 의미합니다.

예를 들어, 우리가 예술적인 개념을 모델링하기 위해 `art` 라는 라이브러리를 만들었다고 가정해 봅시다. 해당 라이브러리에는 두 모듈이 들어 있습니다: `kinds` 모듈은 `PrimaryColor` 과 `SecondaryColor` 열거형을 포함하고, `utils` 모듈은 `mix` 라는 이름의 함수를 포함합니다. Listing 14-3 처럼요.

Filename: src/lib.rs

```
#!/ # Art
//!
//! A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --생략--
    }
}
```

Listing 14-3: `kinds` 모듈과 `utils` 모듈로 이루어진 `art` 라이브러리

Figure 14-3 은 `cargo doc` 으로 생성된 이 크레이트 문서의 첫 화면입니다:

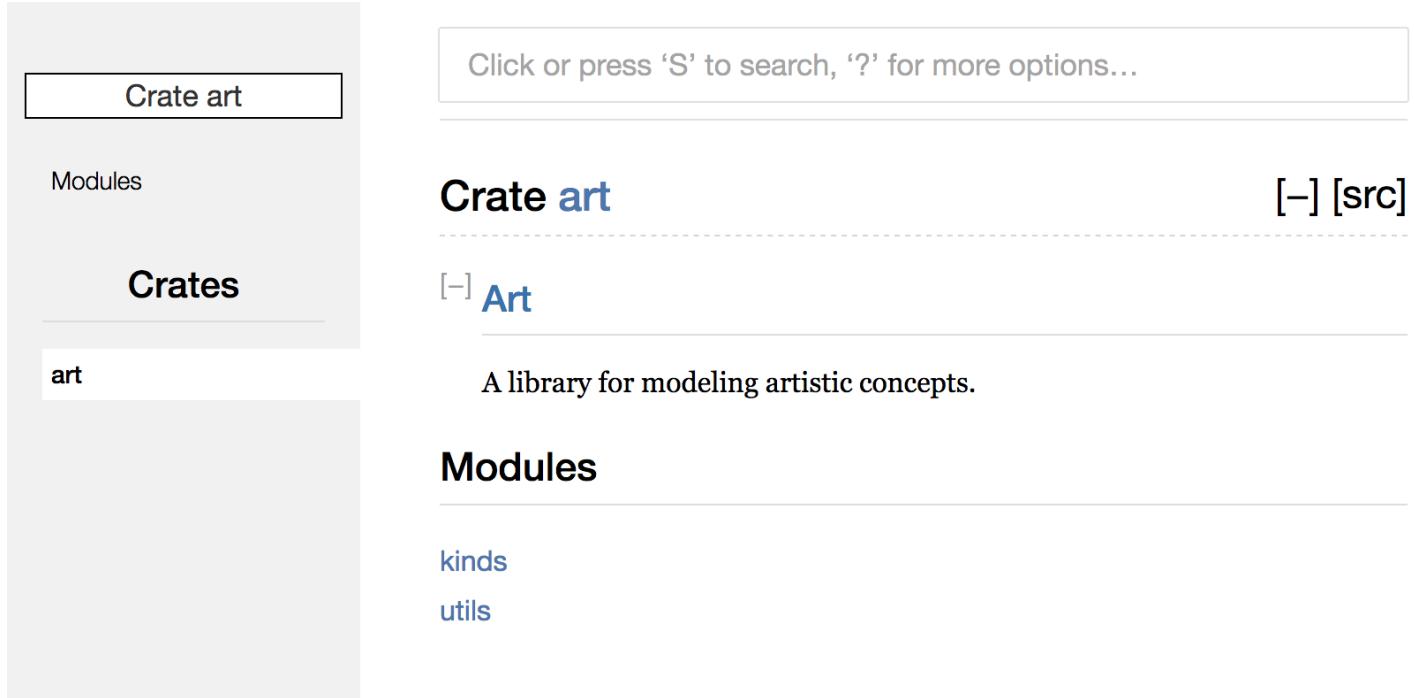


Figure 14-3: `kinds` 와 `utils` 모듈을 포함한 `art` 크레이트의 문서가 렌더링된 모습

`PrimaryColor`, `SecondaryColor` 타입들과 `mix` 함수가 첫 화면에 나오지 않는 걸 주목하세요. 이들을 보려면 각각 `kinds` 와 `utils` 를 클릭하셔야 합니다.

이 라이브러리를 의존성으로 가지고 있는 다른 크레이트에서 `use` 를 이용해 `art` 의 항목을 가져오기 위해선, 현재 정의된 `art` 모듈의 구조대로 일일이 입력해야 합니다. Listing 14-4 에서 다른 크레이트에서 `art` 크레이트의 `PrimaryColor` 과 `mix` 를 이용하는 예시를 볼 수 있습니다.

Filename: src/main.rs

```
extern crate art;

use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

Listing 14-4: `art` 크레이트의 내부 구조에 정의된 항목을 이용하는 또 다른 크레이트

Listing 14-4 의 코드를 작성한, 즉 `art` 크레이트를 사용하는 사람은 `PrimaryColor` 이 `kinds` 모듈에 들어있고 `mix` 가 `utils` 모듈에 들어 있단 걸 알아내야 합니다. 이처럼 현재 `art` 크레이트의 구조는 크레이트를 사용하는 사람보다 크레이트를 개발하는 사람에게 적합한 구조로 되어 있습니다. 내부 구조상에서

의 `kinds` 와 `utils` 모듈의 위치 같은 정보는 `art` 크레이트를 사용하는 입장에서는 전혀 필요 없는 정보이며, 또한 직접 구조상에서 자신이 찾는 것의 위치를 알아내야 하고 `use` 뒤에 모듈의 이름을 일일이 입력해야 한다는 건 혼란스럽고 불편한 일 이니까요.

공개 API로부터 내부 구조의 흔적을 제거하려면 Listing 14-3 처럼 맨 위에서 `pub use` 를 이용해 다시 `export` 하도록 `art` 크레이트의 코드를 수정해야 합니다:

Filename: src/lib.rs

```
///! # Art
///!
///! A library for modeling artistic concepts.

pub use kinds::PrimaryColor;
pub use kinds::SecondaryColor;
pub use utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

Listing 14-5: Re-export 를 위해 `pub use` 추가

`cargo doc` 를 이용해 현재 크레이트에 대한 API 문서를 생성하면 Figure 14-4 처럼 Re-exports 목록과 링크가 첫 페이지에 나타날 겁니다. 이로써 `PrimaryColor`, `Secondary` 타입과 `mix` 함수를 훨씬 더 쉽게 찾을 수 있게 되었네요.

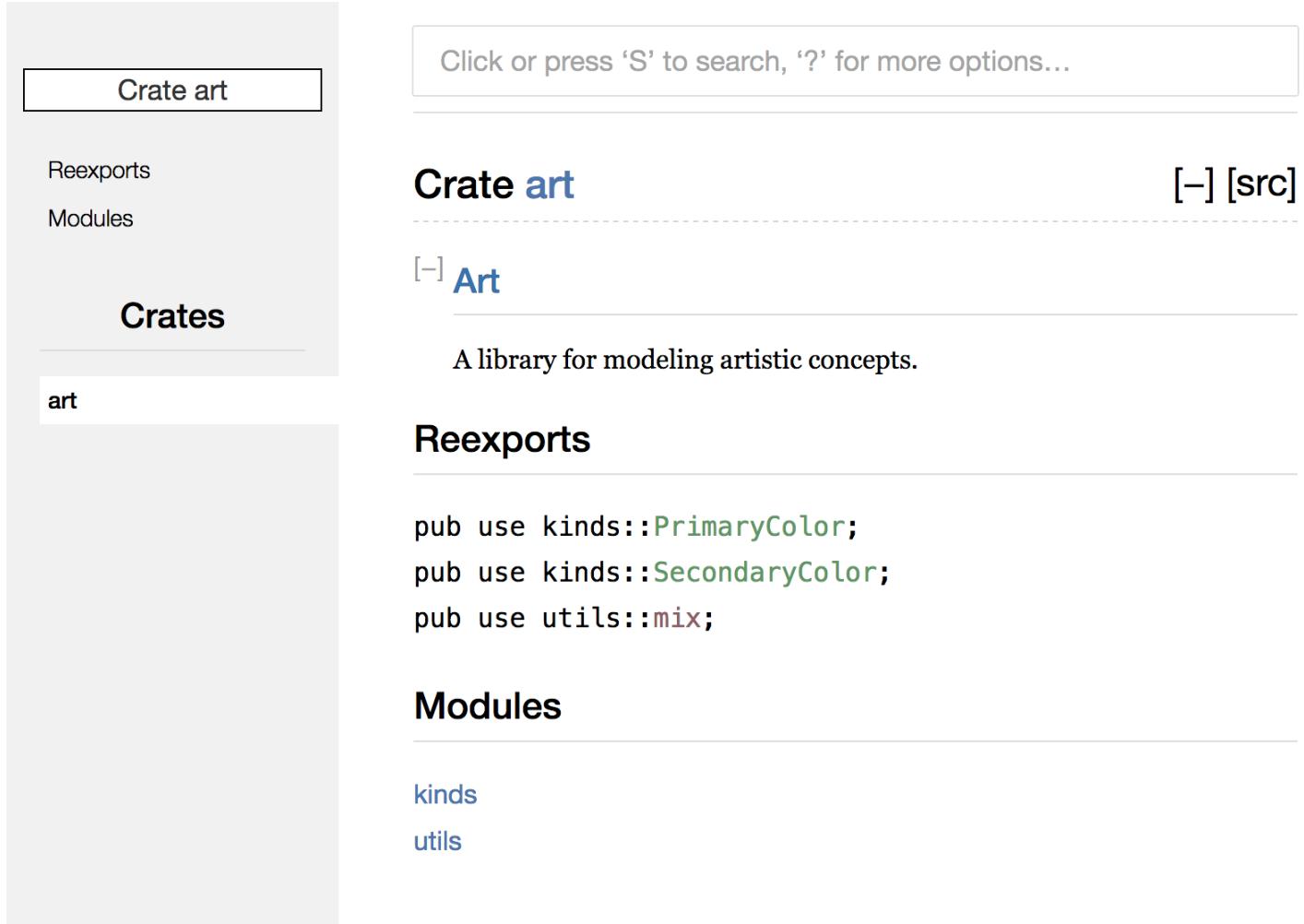


Figure 14-4: Re-exports 목록이 포함된 `art` 크레이트 문서의 첫 페이지

`art` 크레이트의 사용자는 기존의 Listing 14-3의 내부 구조를 이용하여 Listing 14-4처럼 사용하거나, 혹은 좀 더 편한 방식으로 Listing 14-5의 구조를 이용하여 Listing 14-6과 같이 사용할 수 있습니다:

Filename: src/main.rs

```
extern crate art;

use art::PrimaryColor;
use art::mix;

fn main() {
    // --생략--
}
```

Listing 14-6: `art` 크레이트의 Re-export 된 항목들을 사용하는 프로그램

만약 특정 부분에서 중첩된 모듈이 많을 경우, 모듈의 상위 계층에서 `pub use` 를 이용해 타입을 다시 export 함으로써 크레이트의 사용자들에게 더 뛰어난 경험을 제공할 수 있습니다.

쓰기 좋고 편한 형태의 공개 API 를 만드는 일은 기술보단 예술에 가까운 일입니다. 따라서 한번에 완벽한 형태를 만들려고 하기보다는 계속해서 사용자들을 위한 최적의 구조를 찾아 개선해 나가야 합니다. 이럴때 `pub use` 를 이용하면 크레이트 내부를 보다 유연하게 구조화 할 수 있고, 사용자에게 제공하는 것에서 내부 구조의 흔적을 없앨 수 있습니다. 한번 여러분이 설치한 크레이트 중에 아무거나 코드를 열어서 그의 공개 API 구조와 내부 구조를 비교해 보세요. 아마 상당히 다를걸요?

Cartes.io 계정 설정하기

여러분은 첫 크레이트를 배포하기에 앞서, [crates.io](#) 에 계정을 만들고 API 토큰을 얻어야 합니다. [crates.io](#) 홈페이지에 방문하고 GitHub 계정을 통해 로그인 해주세요. (현재는 GitHub 계정이 필수지만, 추후에 사이트에서 다른 방법을 통한 계정 생성을 지원하게 될 수 있습니다) 로그인 하셨다면 계정 설정 페이지인 <https://crates.io/me/> 로 들어가 주세요. 그리고 페이지에서 API 키를 얻어온 후에, 여러분의 API 키를 이용해 `cargo login` 명령어를 실행해 주세요. 이런식으로요:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

이 명령어는 Cargo 에게 여러분의 API 토큰을 알려주고 내부 (`~/.cargo/credentials`) 에 저장하도록 합니다. 미리 말하지만 여러분의 토큰은 남들에겐 비밀입니다: 어떤 이유로 남들에게 알려졌다면, (그 사람을 처리하거나, 혹은) [crates.io](#) 에서 기존의 토큰을 무효화하고 새 토큰을 발급받으세요.

새 크레이트에 Metadata 추가하기

계정을 만들었으니, 여러분이 크레이트를 배포하려고 한다고 가정합시다. 여러분은 배포하기 전에 `Cargo.toml` 파일에 `[package]` 구절을 추가하여 메타데이터(metadata) 를 추가해야합니다.

여러분의 크레이트명은 고유해야 합니다. 여러분이 로컬에서 작업 할 땐 문제 없지만, [crates.io](#) 에 올라갈 크레이트의 이름은 선착순으로 배정되기에, 여러분이 정한 크레이트명을 누군가 이미 쓰고 있다면 해당 크레이트명으로는 크레이트를 배포할 수 없습니다. 크레이트를 배포하기 전에 사이트에서 여러분이 사용하려는 이름을 검색해보고 해당 크레이트명이 이미 사용중인지 확인하세요. 만약 아직 사용중이지 않다면 다음과 같이 `Cargo.toml` 파일 내 `[package]` 절 아래의 이름을 수정하세요:

Filename: `Cargo.toml`

```
[package]
name = "guessing_game"
```

고유한 이름을 선택하고, 크레이트를 배포하기 위해 `cargo publish` 를 실행하면 다음과 같은 경고와 에러가 나타날 겁니다.

```
$ cargo publish
Updating registry `https://github.com/rust-lang/crates.io-index`
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
--snip--
error: api errors: missing or empty metadata fields: description, license.
```

이 에러는 중요한 정보를 몇개 입력하지 않았다는 의미입니다: 설명(description)과 라이센스(license)는 필수적인데, 이들은 각각 사람들에게 해당 크레이트가 어떤 작업을 하는지와 해당 크레이트를 이용할 수 있는 조건을 알려줍니다. 이 에러를 고치려면 이 정보들을 *Cargo.toml*에 포함시켜야 합니다.

설명은 한 문장이나 두 문장정도면 충분합니다. 크레이트를 검색 했을때의 결과에 여러분의 크레이트명과 같이 표시되거든요. **license** 필드엔 라이센스 식별자 값(*license identifier value*)을 부여해야 합니다.

[Linux Foundation's Software Package Data Exchange \(SPDX\)](#)에 여러분이 사용할 수 있는 식별자가 나열되어 있으니 참고 바랍니다. 예를 들어, 만약 여러분의 크레이트에 MIT 라이센스를 적용하고 싶으시다면, 다음과 같이 **MIT** 식별자를 추가하시면 됩니다.

Filename: *Cargo.toml*

```
[package]
name = "guessing_game"
license = "MIT"
```

SPDX에 없는 라이센스를 사용하고 싶으실 경우엔 해당 라이센스의 텍스트를 파일로 만들고 자신의 프로젝트에 해당 파일을 포함시킨 뒤, **license** 대신 **license-file**을 추가해 해당 파일의 이름을 넣으시면 됩니다.

여러분의 프로젝트에 어떤 라이센스가 적합한지에 대해 알아보는 내용은 이 책 범위 이상의 내용입니다. 다만 알아두실 건 러스트 커뮤니티의 많은 이들은 자신의 프로젝트에 러스트 자체가 쓰는 라이센스인 **MIT OR Apache-2.0** 이중 라이센스를 사용한다는 겁니다, 즉 여러분은 프로젝트의 라이선스에 **OR**을 이용해 여러 라이센스 식별자를 명시할 수 있습니다.

고유한 프로젝트명, 버전, **cargo new**로 크레이트를 생성할 때 추가된 작성자 정보, 설명, 라이센스를 모두 추가하셨다면 배포할 준비가 끝났습니다. 이때 *Cargo.toml* 파일의 모습은 다음과 같은 형태일 겁니다:

Filename: *Cargo.toml*

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
description = "A fun game where you guess what number the computer has chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

[Cargo 공식 문서](#)에 다른 사람들이 여러분의 크레이트를 좀 더 찾기 쉽게 해주고, 쓰기 편하게 해주는 나머지 메타데이터들이 설명되어 있으니, 참고 바랍니다.

Crates.io에 배포하기

계정도 만들었고, API 토큰도 얻었고, 크레이트명도 정했고, 메타데이터도 작성했으니 이제 여러분은 크레이트를 배포할 준비 만전이에요! 크레이트를 배포한다는 것은 다른 사람이 사용할 특정 버전을 [crates.io](#)에 올리는 것입니다.

크레이트를 배포할땐 주의하시기 바랍니다. 기본적으로 낙장불입이거든요. 버전은 중복될 수 없으며, 한번 올라간 코드는 수정할 수 없습니다. [crates.io](#)의 원대한 목표중 하나는 [crates.io](#)에 등록된 크레이트들에 의존하는 모든 프로젝트의 빌드가 계속 작동할 수 있도록 영구적인 코드 보관소의 역할을 맡는 것이기 때문에, 버전을 삭제하거나 수정하는 행위는 용납하지 않습니다. 만약 용납한다면 목표를 이룰 수 없으니까요. 대신 버전의 개수에 대한 제한은 없으니 버전을 올리는 것 자체는 얼마든지 가능합니다.

`cargo publish` 명령어를 재실행 해보면 이번엔 성공할 겁니다:

```
$ cargo publish
Updating registry `https://github.com/rust-lang/crates.io-index`
Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
Finished dev [unoptimized + debuginfo] target(s) in 0.19 secs
Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

축하합니다! 이제 여러분의 코드는 러스트 커뮤니티와 공유되고, 아무나 여러분의 크레이트를 자신들의 프로젝트 의존성 목록에 쉽게 추가할 수 있을 겁니다.

이미 배포한 크레이트의 버전 업데이트하기

여러분의 크레이트에 변경사항을 적용하고 새 버전을 릴리즈하려면 *Cargo.toml* 파일의 `version` 값을 새 버전으로 변경하면 됩니다. 이때 변경사항의 종류에 맞춰서 적절한 버전을 결정하는 방법은 [유의적 버전 규칙 \(Semantic Versioning rules\)](#) 을 참고하시기 바랍니다. 버전을 변경하고 나면 `cargo publish` 를 실행해 새 버전을 배포합시다.

cargo yank 를 이용해 Crates.io 에서 버전 제거하기

크레이트의 이전 버전을 제거할 순 없지만, Cargo 는 크레이트의 버전을 *yanking*(끌어내리는) 기능을 지원합니다. 이는 특정 크레이트의 버전이 어떤 이유에선가 문제가 생긴 등의 경우에 새롭게 만들어지는 프로젝트들이 해당 버전을 종속성으로 추가할 수 없도록 막아주는 주는 기능입니다. (역주: *yank* 의 사전적 의미는 **홱 당기다**입니다)

버전을 끌어내려도 해당 버전에 의존하던 기존의 프로젝트들은 계속해서 그 버전에 의존성을 가질 수 있고 해당 버전을 다운로드 받을 수도 있지만, 새로운 프로젝트들이 끌어내려진 버전을 의존성으로 가지는 시작하는 것은 불가능합니다. 근본적인 *yank* 의 의미는 *Cargo.lock* 을 가진 모든 프로젝트는 문제가 없을 것이며, 추후에 새로 생성될 *Cargo.lock* 파일은 끌어내려진 버전을 사용하지 않을 것이라 의미입니다.

크레이트의 버전을 *yank* 하기 위해서는 `cargo yank` 에 *yank* 하고자 하는 버전을 명시하고 실행하시면 됩니다:

```
$ cargo yank --vers 1.0.1
```

또한 여러분은 `--undo` 를 붙여서 *yank* 를 취소하고 다시 새 프로젝트들이 해당 버전을 의존성으로 갖는 것을 허용할 수 있습니다:

```
$ cargo yank --vers 1.0.1 --undo
```

yank 는 어떤 코드도 삭제하지 않습니다. 예를 들어, 여러분이 실수로 자신의 비밀 정보를 업로드한 상황에 대한 해결책으로 *yank* 기능을 사용하셨다면, 이는 잘못된 방법입니다. 만약 그런 일이 일어나면 비밀 정보를 재설정하셔야 합니다.

Cargo 작업공간

12 장에서 바이너리 크레이트와 라이브러리 크레이트를 포함하는 패키지를 만들어 봤습니다. 하지만 여러분이 프로젝트를 개발하다 보면, 라이브러리 크레이트가 점점 거대해져서 여러분의 패키지를 여러개의 라이브러리 크레이트로 분리하고 싶으실 겁니다. Cargo 는 이런 상황에서 사용할 수 있는 **작업공간(workspace)**이라는 기능을 제공하며, 이 기능은 함께 개발된 여러개의 관련된 패키지를 관리하는데 도움이 됩니다.

작업공간 생성

작업공간(workspace) 은 동일한 *Cargo.lock* 과 출력 디렉토리를 공유하는 패키지들의 집합입니다. 한번 이 작업공간을 이용한 프로젝트를 만들어 봅시다. 다만 작업공간의 구조에 집중할 수 있도록 간단한 코드만 사용할 겁니다. 작업공간을 구성하는 방법은 여러가지가 있지만, 일반적인 방법 중 하나를 사용하도록 하겠습니다; 작업 공간은 바이너리 하나와 두 라이브러리를 포함하도록 할 것입니다. 주요 기능을 제공할 바이너리는 두 라이브러리를 의존성으로 가지게 될 것인데, 하나는 `add_one` 함수를 제공할 것이고, 또 하나는 `add_two` 함수를 제공할 것입니다. 이 세 크레이트는 같은 작업 공간의 일부가 될 겁니다. 그럼 작업공간을 위한 새 디렉토리를 만드는 것부터 시작합시다.

```
$ mkdir add
$ cd add
```

다음은 *add* 디렉토리 내에서 전체 작업공간을 구성 할 *Cargo.toml* 파일을 생성합니다. 이 파일은 우리가 여태 다른곳에서 봤던 *Cargo.toml* 파일들과는 달리, **[package]** 절이나 메타데이터를 가지지 않습니다. 대신 **[workspace]** 로 시작하는 구절을 갖는데, 이걸 이용해 작업공간에 `members` 를 추가할 수 있습니다; 추가하는 법은 우리의 바이너리 크레이트 경로를 명시하는 것이며, 이 경우 해당 경로는 `adder` 입니다:

Filename: *Cargo.toml*

```
[workspace]

members = [
    "adder",
]
```

다음으로, *add* 디렉토리 안에서 `cargo new` 를 실행하여 `adder` 바이너리 크레이트를 생성합니다:

```
$ cargo new --bin adder
Created binary (application) `adder` project
```

이 시점에서 우린 작업 공간을 `cargo build` 로 빌드할 수 있습니다. 현재 여러분의 *add* 디렉토리의 내부

모습은 다음과 같은 형태여야 하니, 비교해 보시기 바랍니다:

```

Cargo.lock
Cargo.toml
adder
└── Cargo.toml
    └── src
        └── main.rs
target

```

작업공간은 최상위 디렉토리에 컴파일된 결과를 배치하기 위한 하나의 `target` 디렉토리를 가집니다; 따라서 `adder` 크레이트는 자신만의 `target` 디렉토리를 갖지 않습니다. 만약 `adder` 디렉토리 내에서 `cargo build` 명령어를 실행하더라도 컴파일 결과는 `add/adder/target` 이 아닌 `add/target`에 위치하게 될 겁니다. Cargo 가 작업공간 내에 이와 같이 `target` 디렉토리를 구성한 이유는, 작업공간 내의 크레이트들이 서로 의존하기로 되어있기 때문입니다. 만약 각 크레이트가 각각의 `target` 디렉토리를 갖게 된다면, 각각의 크레이트를 컴파일 할때마다 자신의 `target` 디렉토리에 컴파일 결과를 넣기 위해 다른 크레이트들을 매번 재컴파일하게 될 겁니다. 이와 같은 불필요한 재빌드를 피하기 위해, 하나의 크레이트들은 `target` 디렉토리를 공유하도록 되어 있습니다.

작업공간에 두번째 크레이트 만들기

다음은 작업공간에 `add-one` 이라고 부를 새로운 멤버 크레이트를 생성해 봅시다. `members` 목록에 `add-one` 경로를 지정하기 위해 최상위의 `Cargo.toml` 파일을 수정합시다.

Filename: `Cargo.toml`

```
[workspace]

members = [
    "adder",
    "add-one",
]
```

그리고 `add-one`이라는 새 라이브러리 크레이트를 생성합니다.

```
$ cargo new add-one
   Created library `add-one` project
```

이제 여러분의 `add` 디렉토리는 다음과 같은 디렉토리와 파일들을 갖게 될 겁니다:

```

Cargo.lock
Cargo.toml
add-one
└── Cargo.toml
    └── src
        └── lib.rs
adder
└── Cargo.toml
    └── src
        └── main.rs
target

```

add-one/src/lib.rs 파일에 `add_one` 함수를 추가합시다:

Filename: *add-one/src/lib.rs*

```

pub fn add_one(x: i32) -> i32 {
    x + 1
}

```

이제 우린 작업공간 내에 라이브러리 크레이트를 가졌으니, `adder` 바이너리 크레이트를 `add-one` 라이브러리 크레이트에 의존하도록 만들 수 있습니다. 먼저, *adder/Cargo.toml*에 `add-one`에 대한 의존성 경로를 추가합시다.

Filename: *adder/Cargo.toml*

```

[dependencies]
add-one = { path = "../add-one" }

```

Cargo는 작업공간 내 크레이트들이 서로 의존하고 있을 것이라고 추정하지 않기 때문에, 우리가 크레이트 간의 의존 관계에 대해 명시해 주어야 합니다.

다음으로 `adder` 크레이트에서 `add-one` 크레이트의 `add_one` 함수를 사용해보도록 합시다. *adder/src/main.rs* 파일을 열고 상단에 `extern crate` 행을 추가해 스코프 내로 `add-one` 라이브러리를 가져오도록 한 뒤, `main` 함수를 `add_one` 함수를 호출하도록 변경합니다. Listing 14-7 처럼요:

Filename: *adder/src/main.rs*

```
extern crate add_one;

fn main() {
    let num = 10;
    println!("Hello, world! {} plus one is {}!", num, add_one::add_one(num));
}
```

Listing 14-7: `adder` 크레이트에서 `add-one` 라이브러리 사용하기

이제 한번 최상위 `add` 디렉토리에서 `cargo build` 를 실행해 작업공간을 빌드해 봅시다!

```
$ cargo build
Compiling add-one v0.1.0 (file:///projects/add/add-one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs
```

`add` 디렉토리에서 바이너리 크레이트를 실행하기 위해선 `cargo run` 에 `-p` 옵션과 패키지 이름을 사용하여 우리가 작업공간 내에서 사용할 패키지를 명시해야 합니다:

```
$ cargo run -p adder
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

이는 `add-one` 크레이트에 의존성을 가진 `adder/src/main.rs` 코드를 실행시킵니다.

작업공간의 외부 크레이트에 의존성 갖기

작업공간은 작업공간에 있는 각각의 크레이트의 디렉토리에 `Cargo.lock` 파일을 갖는게 아닌, 작업공간의 최상위에만 단 하나의 `Cargo.lock` 파일을 갖는다는 걸 기억하세요. 이는 모든 크레이트들이 모든 의존성의 같은 버전을 사용함을 보증합니다. 만약 우리가 `rand` 크레이트를 `adder/Cargo.toml` 과 `add-one/Cargo.toml`에 추가하면 Cargo 는 둘을 모두 같은 버전을 쓰도록 결정하고 하나의 `Cargo.lock`에 기록합니다. 작업공간의 모든 크레이트들이 같은 의존성을 갖도록 한다는 의미는 작업공간 내의 크레이트들이 항상 서로 조화를 이룬다는 의미입니다. 한번 `add-one` 크레이트에서 `rand` 크레이트를 사용할 수 있도록 `add-one/Cargo.toml` 파일의 `[dependencies]` 절에 `rand` 를 추가해 봅시다:

Filename: `add-one/Cargo.toml`

```
[dependencies]
rand = "0.3.14"
```

이제 우린 `add-one/src/lib.rs` 파일에 `extern crate rand;` 를 추가할 수 있으며, `add` 디렉토리에서 `cargo build` 를 이용해 전체 작업공간을 빌드하면 `rand` 크레이트를 가져오고 컴파일 할 것입니다:

```
$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading rand v0.3.14
  --snip--
  Compiling rand v0.3.14
  Compiling add-one v0.1.0 (file:///projects/add/add-one)
  Compiling adder v0.1.0 (file:///projects/add/adder)
  Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs
```

이제 최상위 `Cargo.lock` 엔 `add-one` 의 `rand` 로의 종속성 정보가 포함되어 있습니다. 하지만, 작업공간의 어딘가에서 `rand` 를 사용하였다고 해도, 작업공간의 다른 크레이트에선 `rand` 를 자신의 `Cargo.toml` 파일에 추가하지 않는 한 사용이 불가능합니다. 예를 들어, 만약 `adder` 크레이트에서 `rand` 를 그냥 사용하기 위해 `adder/src/main.rs` 파일에 `extern crate rand;` 를 추가하면 에러가 나타납니다:

```
$ cargo build
  Compiling adder v0.1.0 (file:///projects/add/adder)
error: use of unstable library feature 'rand': use `rand` from crates.io (see
issue #27703)
--> adder/src/main.rs:1:1
 |
1 | extern crate rand;
```

이 에러를 해결하려면 `adder` 크레이트의 `Cargo.toml` 파일을 수정하여 `rand` 를 해당 크레이트의 의존성으로 나타내야합니다. 그 후 `adder` 크레이트를 빌드하면 `Cargo.lock` 의 `adder` 을 위한 의존성 목록에 `rand` 가 추가될 테지만, `rand` 가 다시 다운로드 되진 않을 겁니다. Cargo 는 `rand` 를 사용하는 작업공간 내의 크레이트는 모두 같은 버전의 `rand` 크레이트를 사용할 것임을 보장하기 때문에 같은 크레이트를 여러개의 버전으로 다운로드 받을 필요 없고, 따라서 그만큼 공간은 절약되며, 작업공간 내의 각 크레이트는 조화를 이룰 수 있습니다.

작업공간에 테스트 추가하기

또 다른 향상을 위해, `add_one` 크레이트의 `add_one::add_one` 함수에 대한 테스트를 추가해 봅시다.

Filename: `add-one/src/lib.rs`

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

이제 최상위 `add` 디렉토리에서 `cargo test` 를 실행해 봅시다:

```
$ cargo test
Compiling add-one v0.1.0 (file:///projects/add/add-one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
Running target/debug/deps/add_one-f0253159197f7841

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Running target/debug/deps/adder-f88af9d2cc175a5e

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

출력의 첫번째 절은 `add-one` 크레이트의 `it_works` 테스트가 통과했다는 의미이고, 다음 절은 `adder` 크레이트에서 테스트를 찾지 못했다는 의미이며, 마지막 절은 `add-one` 크레이트에서 문서화 테스트를 찾지 못했다는 의미입니다. 이처럼 작업공간 구조 내에서 `cargo test` 를 실행하면 작업공간 내의 모든 크레이트에 대한 테스트들이 실행됩니다.

우린 작업공간 내의 하나의 특정한 크레이트에 대한 테스트도 실행할 수 있습니다. 최상위 디렉토리에서 `-p` 플래그와 테스트 하고자 하는 크레이트명을 명시해줌으로써 말이죠:

```
$ cargo test -p add-one
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/deps/add_one-b3235fea9a156f74

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

이 출력은 `cargo test` 가 `adder` 크레이트는 테스트하지 않고 `add-one` 크레이트에 대해서만 테스트를 실행 했음을 보여줍니다.

만약 여러분이 <https://crates.io/> 에 작업공간 내의 크레이트를 배포하시려면, 각 크레이트는 분리돼서 배포되어야 합니다. `cargo publish` 명령어엔 `--all` 이나 `-p` 같은 플래그가 없어요. 따라서 여러분은 각 크레이트 디렉토리를 수정하고 `cargo publish` 를 실행해야 합니다.

추가 과제로는, 한번 이 작업공간에 `add-two` 크레이트를 추가해 보세요! `add-one` 크레이트를 추가할 때와 비슷한 방법으로 하시면 됩니다.

언젠가 여러분의 프로젝트가 커지면 작업공간을 사용하는 걸 고려해보세요: 하나의 거대한 코드보다 작은 개별 요소를 이해하는 일이 훨씬 쉽고, 작업공간에서 크레이트를 관리한다면 각 크레이트가 동시에 변경되는 경우도 쉽게 조정할 수 있습니다.

cargo install 을 이용해 Crates.io 에서 바이너리 설치하기

`cargo install` 명령어는 여러분이 로컬에서 바이너리 크레이트를 설치하고 사용할 수 있도록 해줍니다. 이는 시스템 패키지를 대체하기 위한 것이 아닌, 러스트 개발자들이 [crates.io](#)에서 공유하고 있는 툴을 편리하게 설치할 수 있도록 하기 위함입니다. 여러분은 [바이너리 타겟\(binary target\)](#)을 가진 패키지만 설치할 수 있다는 걸 알아두셔야 하는데, 이 [바이너리 타겟](#)이란 혼자서 실행될 수 없고 다른 프로그램에 포함되는 용도인 라이브러리 타겟과는 반대되는 의미로, `src/main.rs` 파일 혹은 따로 바이너리로 지정된 파일을 가진 크레이트가 생성해낸 실행 가능한 프로그램을 말합니다. 보통 해당 크레이트가 라이브러리인지, 바이너리 타겟을 갖는지, 혹은 둘 다인지에 대한 정보를 `README` 파일에 작성해둡니다.

`cargo install` 을 이용해 설치한 모든 바이너리들은 Cargo가 설치된 폴더의 `bin` 폴더에 저장됩니다. 만약 여러분이 `rustup.rs`를 이용해 러스트를 설치하셨고, 따로 설정을 건들지 않으셨다면 `$HOME/.cargo/bin` 폴더입니다. `cargo install`로 설치한 프로그램을 실행하시려면 여러분의 `$PATH` 환경변수에 해당 디렉토리가 등록되어 있는지 확인하세요.

12장에서 언급한 `grep`을 러스트로 구현한 파일 검색 툴인 `ripgrep`을 예로 들어봅시다. `ripgrep`을 설치하려면 다음과 같이 하면 됩니다:

```
$ cargo install ripgrep
Updating registry `https://github.com/rust-lang/crates.io-index`
Downloading ripgrep v0.3.2
--snip--
Compiling ripgrep v0.3.2
  Finished release [optimized + debuginfo] target(s) in 97.91 secs
Installing ~/.cargo/bin/rg
```

출력의 마지막 줄은 설치된 바이너리의 경로와 이름을 보여줍니다. `ripgrep`의 이름은 `rg`네요. 방금 앞에서 말했던 것처럼 여러분의 `$PATH` 환경변수에 설치된 폴더가 등록되어 있는 한, 여러분은 명령창에서 `rg --help`를 실행할 수 있고, 앞으로 파일을 찾을 때 더 빠르고 러스트다운 툴을 사용할 수 있습니다!

커스텀 명령어로 Cargo 확장하기

Cargo 는 여러분이 직접 Cargo 를 수정하지 않고도 새 보조 명령어를 추가할 수 있도록 되어 있습니다. 만약 여러분의 `$PATH` 내 어떤 바이너리의 이름이 `cargo-something` 이고, 해당 바이너리가 Cargo 의 보조 명령어 바이너리일 경우 `cargo something` 라는 명령어를 이용해 실행할 수 있습니다. 이와 같은 커스텀 명령어들은 `cargo --list` 를 실행 할 때의 목록에도 포함됩니다. 이런식으로 `cargo install` 을 이용해 확장 모듈을 설치하고 Cargo 의 자체 툴처럼 이용할 수 있다는 점은 Cargo 의 무척 편리한 점 중 하나입니다.

정리

Cargo 와 crates.io 를 통해 코드를 공유하는 행위는 러스트 생태계를 발전시키고, 러스트가 많은 방면에서 활약하도록 만드는데 주축이 되는 행위입니다. 러스트의 기본 라이브러리는 작고 고정되어 있지만, 크레이트들은 쉽게 공유될 수 있고, 쉽게 사용될 수 있으며 러스트 언어 자체보다 훨씬 빠른 속도로 발전합니다. 여러분에게 유용한 코드가 있다면 주저말고 crates.io 에 공유하세요; 분명 다른 누군가에게도 도움이 될 테니까요!

스마트 포인터

포인터 (pointer)는 메모리의 주소 값을 담고 있는 변수에 대한 일반적인 개념입니다. 이 주소 값은 어떤 다른 데이터를 참조합니다. 혹은 바꿔 말하면, “가리킵니다”. 러스트에서 가장 흔한 종류의 포인터는 참조자인데, 이는 여러분들이 3장에서 배웠던 것입니다. 참조자는 & 심볼에 의해 나타내지고 이들이 가리키고 있는 값을 빌립니다. 이들은 값을 참조하는 것 외에 다른 어떤 특별한 능력도 없습니다. 또한, 이들은 어떠한 오버헤드도 발생하지 않으며 우리가 가장 자주 사용하는 포인터의 한 종류입니다.

한편, 스마트 포인터 (smart pointer)는 포인터처럼 작동하지만 추가적인 메타데이터와 능력들도 가지고 있는 데이터 구조입니다. 스마트 포인터의 개념은 러스트에 고유한 것이 아닙니다: 스마트 포인터는 C++로부터 유래되었고 또한 다른 언어들에도 존재합니다. 러스트에서는, 표준 라이브러리에 정의된 다양한 종류의 스마트 포인터들이 참조자들에 의해 제공되는 것을 넘어서는 추가 기능을 제공합니다. 우리가 이번 장에서 탐구할 한 가지 예로는 참조 카운팅 (reference counting) 스마트 포인터 타입이 있습니다. 이 포인터는 소유자의 수를 계속 추적하고, 더 이상 소유자가 없으면 데이터를 정리하는 방식으로, 여러분들이 어떤 데이터에 대한 여러 소유자들을 만들 수 있게 해 줍니다.

소유권과 빌림의 개념을 가지고 있는 러스트에서, 참조자와 스마트 포인터 간의 추가적인 차이점은 참조자가 데이터를 오직 빌리기만 하는 포인터라는 점입니다; 반면, 많은 경우에서 스마트 포인터는 그들이 가리키고 있는 데이터를 소유합니다.

우리는 이미 이 책에서 8장의 `String` 과 `Vec<T>`와 같은 몇 가지 스마트 포인터들을 마주쳤습니다. 비록 그때는 이것들을 스마트 포인터라고 부르지 않았지만요. 이 두 타입 모두 스마트 포인터로 치는데 그 이유는 이들이 얼마간의 메모리를 소유하고 여러분이 이를 다루도록 허용하기 때문입니다. 그들은 또한 (그들의 용량 등의) 메타데이터와 (`String`이 언제나 유효한 UTF-8일 것임을 보장하는 것 등의) 추가 능력 혹은 보장을 갖고 있습니다.

스마트 포인터는 보통 구조체를 이용하여 구현되어 있습니다. 스마트 포인터가 일반적인 구조체와 구분되는 특성은 바로 스마트 포인터가 `Deref` 와 `Drop` 트레이잇을 구현한다는 것입니다. `Deref` 트레이잇은 스마트 포인터 구조체의 인스턴스가 참조자처럼 동작하도록 하여 참조자나 스마트 포인터 둘 중 하나와 함께 작동하는 코드를 작성하게 해 줍니다. `Drop` 트레이잇은 스마트 포인터의 인스턴스가 스코프 밖으로 벗어났을 때 실행되는 코드를 커스터마이징 가능하도록 해 줍니다. 이번 장에서는 이 두 개의 트레이잇 모두를 다루고 이들이 어째서 스마트 포인터에게 중요한지를 보여줄 것입니다.

스마트 포인터 패턴이 러스트에서 자주 사용되는 일반적인 디자인 패턴으로 주어지므로, 이번 장에서는 존재하는 스마트 포인터를 모두 다루지는 않을 것입니다. 많은 라이브러리들이 그들 자신만의 스마트 포인터를 가지고 있고, 심지어 여러분도 여러분 자신만의 것을 작성할 수 있습니다. 우리는 표준 라이브러리 내의 가장 흔한 스마트 포인터들을 다룰 것입니다:

- 값을 힙에 할당하기 위한 `Box<T>`
- 복수개의 소유권을 가능하게 하는 참조 카운팅 타입인 `Rc<T>`

- 빌림 규칙을 컴파일 타임 대신 런타임에 강제하는 타입인, `RefCell<T>`를 통해 접근 가능한 `Ref<T>` 와 `RefMut<T>`

추가로, 우리는 불변 타입이 내부 값을 변경하기 위하여 API를 노출하는 **내부 가변성 (interior mutability)** 패턴에 대해 다룰 것입니다. 또한 *참조 순환 (reference cycles)*이 어떤 식으로 메모리가 세어나가게 할 수 있으며, 이를 어떻게 방지하는지에 대해서도 논의해 보겠습니다.

함께 뛰어들어 볼까요!

Box<T>는 힙에 있는 데이터를 가리키고 알려진 크기를 갖습니다

가장 직관적인 스마트 포인터는 박스 (box) 인데, 이 타입은 `Box<T>` 라고 쓰입니다. 박스는 여러분이 데이터를 스택이 아니라 힙에 저장할 수 있도록 해줍니다. 스택에 남는 것은 힙 데이터를 가리키는 포인터입니다. 스택과 힙의 차이를 살펴보면 4장을 참조하세요.

박스는 스택 대신 힙에 데이터를 저장한다는 점 외에는, 성능적인 오버헤드가 없습니다. 하지만 여러 가지의 추가 기능 또한 가지고 있지 않습니다. 여러분은 이를 아래와 같은 상황에서 가장 자주 쓰게 될 것입니다:

- 컴파일 타임에 크기를 알 수 없는 타입을 갖고 있고, 정확한 사이즈를 알 필요가 있는 맥락 안에서 해당 타입의 값을 이용하고 싶을 때
- 커다란 데이터를 가지고 있고 소유권을 옮기고 싶지만 그렇게 했을 때 데이터가 복사되지 않을 것이라고 보장하기를 원할 때
- 어떤 값을 소유하고 이 값의 구체화된 타입을 알고 있기보다는 특정 트레이트를 구현한 타입이라는 점만 신경 쓰고 싶을 때

이 장에서는 첫 번째 상황을 보여줄 것입니다. 그러나 보여주기 전에, 나머지 두 상황에 대해 약간 더 자세히 말하겠습니다: 두 번째 경우, 방대한 양의 데이터의 소유권 옮기기는 긴 시간이 소요될 수 있는데 이는 그 데이터가 스택 상에서 복사되기 때문입니다. 이러한 상황에서 성능을 향상하기 위해서, 박스 안의 힙에 그 방대한 양의 데이터를 저장할 수 있습니다. 그러면, 작은 양의 포인터 데이터만 스택 상에서 복사되고, 데이터는 힙 상에서 한 곳에 머물게 됩니다. 세 번째 경우는 **트레이트 객체 (trait object)**라고 알려진 것이고, 17장이 이 주제만으로 전체를 쏟아부었습니다. 그러니 여러분이 여기서 배운 것을 17장에서 다시 적용하게 될 것입니다!

Box<T>을 사용하여 힙에 데이터를 저장하기

`Box<T>`에 대한 사용례를 논의하기 전에, 먼저 문법 및 `Box<T>` 내에 저장된 값과 어떻게 상호작용 하는지 다루겠습니다.

Listing 15-1은 힙에 `i32` 값을 저장하기 위해 박스를 사용하는 법을 보여줍니다:

Filename: src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Listing 15-1: 박스를 사용하여 `i32` 값을 힙에 저장하기

`5`라는 값을 가리키는 `Box`의 값을 갖는 변수 `b`를 선언했는데, 여기서 `5`는 힙에 할당됩니다. 이 프로그램은 `b = 5`를 출력할 것입니다; 이 경우, 우리는 마치 이 데이터가 스택에 있었던 것과 유사한 방식으로 박스 내의 데이터에 접근할 수 있습니다. 다른 어떤 소유한 값과 마찬가지로, `b`가 `main`의 끝에 도달하는 것처럼 어떤 박스가 스코프를 벗어날 때, 할당은 해제될 것입니다. 할당 해제는 (스택에 저장된) 박스와 이것이 가리키고 있는 (힙에 저장된) 데이터 모두에게 일어납니다.

단일 값을 힙에 집어넣는 것은 그다지 유용하지는 않으므로, 이 방식처럼 박스를 이용하는 것은 자주 쓰지 않을 것입니다. 단일한 `i32` 같은 값을 스택에 갖는 것은, 스택이 해당 값이 기본적으로 저장되는 곳이기도 하고, 대부분의 경우에서 더 적절합니다. 만일 우리가 박스를 쓰지 않는다면 허용되지 않았을 타입을 정의하도록 해주는 경우를 살펴봅시다.

박스는 재귀적 타입을 가능하게 합니다

컴파일 타임에서, 러스트는 어떤 타입이 얼마나 많은 공간을 차지하는지를 알 필요가 있습니다. 컴파일 타임에는 크기를 알 수 없는 한 가지 타입이 바로 **재귀적 타입** (*recursive type*) 인데, 이는 어떤 값이 그 일부로서 동일한 타입의 다른 값을 가질 수 있는 것을 말합니다. 이러한 값의 내포가 이론적으로는 무한하게 계속될 수 있으므로, 러스트는 재귀적 타입의 값이 얼마큼의 공간을 필요로 하는지 알지 못합니다. 하지만, 박스는 알려진 크기를 갖고 있으므로, 재귀적 타입 정의 내에 박스를 넣음으로써 이를 쓸 수 있습니다.

재귀적 타입의 예제로서, 함수형 프로그래밍 언어에서 일반적인 데이터 타입인 `cons list`를 탐험해 봅시다. 우리가 정의할 `cons list` 타입은 재귀를 제외하면 직관적입니다; 그러므로, 우리가 작업할 예제에서의 개념은 여러분이 재귀적 타입을 포함하는 더 복잡한 어떠한 경우에 처하더라도 유용할 것입니다.

Cons List에 대한 더 많은 정보

`cons list`는 Lisp 프로그래밍 언어 및 그의 파생 언어들로부터 유래된 데이터 구조입니다. Lisp에서, (“생성 함수 (construct function)”의 줄임말인) `cons` 함수는 두 개의 인자를 받아 새로운 한 쌍을 생성하는데, 이 인자는 보통 단일 값과 또 다른 쌍입니다. 이러한 쌍들을 담고 있는 쌍들이 리스트를 형성합니다.

`cons` 함수 개념은 더 일반적인 함수형 프로그래밍 용어로 나아갑니다: “to cons `x` onto `y`”는 약식으로 요소 `x`를 새로운 컨테이너에 집어넣고, 그다음 컨테이너 `y`를 넣는 식으로 새로운 컨테이너 인스턴스를 생성하는 것을 의미합니다.

`cons list` 내의 각 아이템은 두 개의 요소를 담고 있습니다: 현재 아이템의 값과 다음 아이템이지요. 리스트의 마지막 아이템은 다음 아이템 없이 `Nil`이라 불리는 값을 담고 있습니다. `cons list`는 `cons` 함수를 재귀적으로 호출함으로써 만들어집니다. 재귀의 기본 케이스를 의미하는 표준 이름이 바로 `Nil`입니다. 유효하지 않은 값 혹은 값이 없는 것을 말하는 6장의 “null” 혹은 “nil” 개념과 동일하지 않다는 점을 주의하세요.

비록 함수형 프로그래밍 언어들이 `cons list`를 자주 사용할지라도, 러스트에서는 흔히 사용되는 데이터 구조

가 아닙니다. 러스트에서 아이템의 리스트를 갖는 대부분의 경우에는, `Vec<T>`이 사용하기에 더 나은 선택입니다. 그와는 다른, 더 복잡한 재귀적 데이터 타입들은 다양한 상황들에서 유용하기는 하지만, `cons list`를 가지고 시작함으로써, 박스가 어떻게 재귀적 데이터 타입을 정의하도록 해주는지 우리의 집중을 방해하는 것들 없이 탐구할 수 있습니다.

Listing 15-2는 `cons list`를 위한 열거형 정의를 담고 있습니다. 우리가 보여주고자 하는 것인데, `List` 타입이 알려진 크기를 가지고 있지 않고 있기 때문에 이 코드는 아직 컴파일이 안된다는 점을 유의하세요:

Filename: src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```

Listing 15-2: `i32` 값의 `cons list` 데이터 구조를 표현하는 열거형 정의에 대한 첫 번째 시도

노트: 이 예제의 목적을 위해 오직 `i32` 값만 담는 `cons list`를 구현하고 있습니다. 우리가 10장에서 논의한 것처럼, 임의의 타입 값을 저장할 수 있는 `cons list` 타입을 정의하기 위해서는 제네릭을 이용해 이를 구현할 수도 있습니다.

`List` 타입을 이용하여 리스트 `1, 2, 3`을 저장하는 것은 Listing 15-3의 코드와 같이 보일 것입니다:

Filename: src/main.rs

```
use List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

Listing 15-3: `List` 열거형을 이용하여 리스트 `1, 2, 3` 저장하기

첫 번째 `Cons` 값은 `1`과 `List` 값을 갖습니다. 이 `List` 값은 `2`와 또 다른 `List` 값을 갖는 `Cons` 값입니다. 그 안의 `List` 값은 `3`과 `List` 값을 갖는 추가적인 `Cons`인데, 여기서 마지막의 `List`은 `Nil`로서, 리스트의 끝을 알리는 비재귀적인 variant입니다.

만일 Listing 15-3의 코드를 컴파일하고자 시도하면, Listing 15-4에 보이는 에러를 얻습니다:

```
error[E0072]: recursive type `List` has infinite size
--> src/main.rs:1:1
1 | enum List {
  | ^^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |             ----- recursive without indirection
|
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
make `List` representable
```

Listing 15-4: 재귀적 열거형을 정의하고자 시도했을 때 얻게 되는 에러

이 에러는 이 타입이 “무한한 크기를 갖는다”라고 말해줍니다. 그 원인은 우리가 재귀적인 variant를 이용하여 `List`를 정의했기 때문입니다: 즉 이것은 또 다른 자신을 직접 값으로 갖습니다. 결과적으로, 러스트는 `List` 값을 저장하는데 필요한 크기가 얼마나 되는지 알아낼 수 없습니다. 왜 우리가 이런 에러를 얻게 되는지 좀 더 쪼개어 봅시다: 먼저, 러스트가 비재귀적인 타입의 값을 저장하는데 필요한 용량이 얼마나 되는지 결정하는 방법을 살펴봅시다.

비재귀적 타입의 크기 계산하기

6장에서 열거형 정의에 대해 논의할 때 우리가 Listing 6-2에서 정의했던 `Message` 열거형을 상기해봅시다:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

`Message` 값을 할당하기 위해 얼마나 많은 공간이 필요한지를 결정하기 위해서, 러스트는 어떤 variant가 가장 많은 공간을 필요로 하는지를 알기 위해 각각의 variant들 내부를 봅니다. 러스트는 `Message::Quit` 가 어떠한 공간도 필요 없음을 알게 되고, `Message::Move` 는 두 개의 `i32` 값을 저장하기에 충분한 공간이 필요함을 알게 되고, 그렇게 진행됩니다. 단 하나의 variant만 사용될 것이기 때문에, `Message` 값이 필요로 하는 가장 큰 공간은 그것의 variant 중 가장 큰 것을 저장하는데 필요한 공간입니다.

러스트가 Listing 15-2의 `List` 열거형과 같은 재귀적 타입이 필요로 하는 공간을 결정하고자 시도할 때 어떤 일이 일어나는지를 이와 대조해보세요. 컴파일러는 `Cons` variant를 살펴보는 것을 시작하는데, 이는 `i32` 타입의 값과 `List` 타입의 값을 갖습니다. 그러므로, `Cons`는 `i32`의 크기에 `List` 크기를 더한 만큼의 공간을 필요로 합니다. `List` 타입이 얼마나 많은 메모리를 차지하는지 알아내기 위해서, 컴파일러는 그것의 variants를 살펴보는데, 이는 `Cons` variant로 시작됩니다. `Cons` variant는 `i32` 타입의 값과

List 타입의 값을 갖고, 이 과정은 Figure 15-1에서 보는 바와 같이 무한히 계속됩니다:

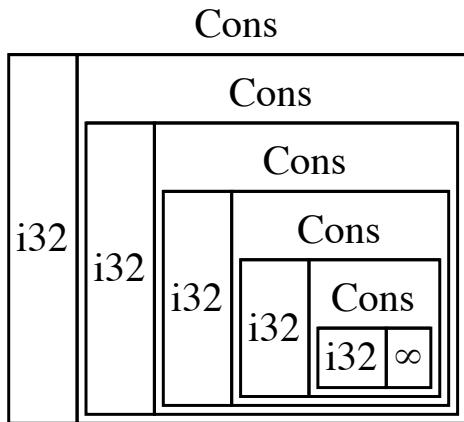


Figure 15-1: 무한한 **Cons** variant를 가지고 있는 무한한 **List**

Box<T>를 이용하여 알려진 크기를 가진 재귀적 타입 만들기

러스트는 재귀적으로 정의된 타입을 위하여 얼마큼의 공간을 할당하는지 알아낼 수 없으므로, 컴파일러는 Listing 15-4의 에러를 내줍니다. 하지만 이 에러는 아래와 같은 유용한 제안을 포함하고 있습니다:

```
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
make `List` representable
```

이 제안에서, “간접 (indirection)”은 값을 직접 저장하는 대신, 간접적으로 값의 포인터를 저장하기 위하여 데이터 구조를 바꿀 수 있음을 의미합니다.

Box<T>가 포인터이기 때문에, 러스트는 언제나 **Box<T>**가 필요로 하는 공간이 얼마인지 알고 있습니다: 포인터의 크기는 그것이 가리키고 있는 데이터의 양에 기반하여 변경되지 않습니다. 이는 우리가 **Cons** variant 내에 또 다른 **List** 값을 직접 넣는 대신 **Box<T>**를 넣을 수 있다는 뜻입니다. **Box<T>**는 **Cons** variant 안에 있기보다는 힙에 있을 다음의 **List** 값을 가리킬 것입니다. 개념적으로, 우리는 다른 리스트들을 “담은” 리스트들로 만들어진 리스트를 여전히 갖게 되지만, 이 구현은 이제 또 다른 것 안의 아이템들이 아니라 또 다른 것 옆에 있는 아이템들에 더 가깝습니다.

우리는 Listing 15-2의 **List** 열거형의 정의와 Listing 15-3의 **List** 사용법을 Listing 15-5의 코드로 바꿀 수 있는데, 이는 컴파일될 것입니다:

Filename: src/main.rs

```

enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::*;

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}

```

Listing 15-5: 알려진 크기를 갖도록 하기 위해 `Box<T>`를 이용하는 `List`의 정의

`Cons` variant는 `i32`와 박스의 포인터 데이터를 저장할 공간을 더한 크기를 요구할 것입니다. `Nil` variant는 아무런 값도 저장하지 않으므로, `Cons` variant에 비해 공간을 덜 필요로 합니다. 우리는 이제 어떠한 `List` 값이 `i32`의 크기 더하기 박스의 포인터 데이터의 크기만큼을 차지할 것인 점을 알게 되었습니다. 박스를 이용함으로써, 우리는 무한하고, 재귀적인 연결을 부수었고, 따라서 컴파일러는 `List` 값을 저장하는데 필요한 크기를 알아낼 수 있습니다. Figure 15-2는 `Cons` variant가 이제 어떻게 생겼는지를 보여주고 있습니다:

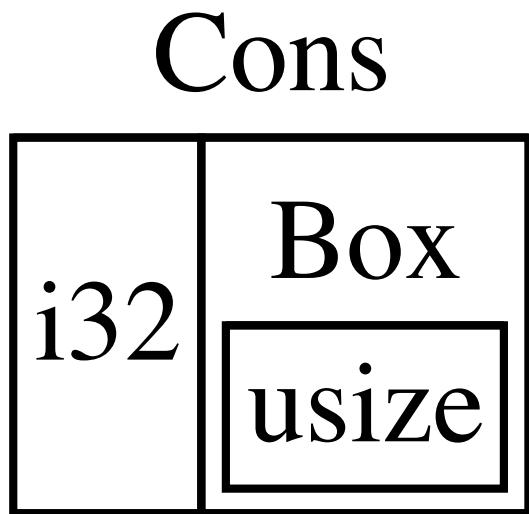


Figure 15-2: `Cons` 가 `Box` 를 들고 있기 때문에 무한한 크기가 아니게 된 `List`

박스는 단지 간접 및 힙 할당만을 제공할 뿐입니다; 이들은 다른 어떤 특별한 능력들, 우리가 다른 스마트 포인터 타입들에서 보게 될 것 같은 능력들이 없습니다. 또한 이들은 이러한 특별한 능력들이 초래하는 성능적인 오버헤드도 가지고 있지 않으므로, 우리가 필요로 하는 기능이 딱 간접 하나인 cons list와 같은 경우에 유

용할 수 있습니다. 우리는 또한 17장에서 박스에 대하여 더 많은 사용례를 살펴볼 것입니다.

`Box<T>` 타입은 스마트 포인터인데 그 이유는 이것이 `Deref` 트레잇을 구현하고 있기 때문이며, 이는 `Box<T>` 값이 참조자와 같이 취급되도록 허용해줍니다. `Box<T>` 값이 스코프 밖으로 벗어날 때, 박스가 가리키고 있는 힙 데이터도 마찬가지로 정리되는데 이는 `Drop` 트레잇의 구현 때문에 그렇습니다. 이 두 가지 트레잇에 대하여 더 자세히 탐구해 봅시다. 이 두 트레잇이 이 장의 나머지에서 다루게 될 다른 스마트 포인터 타입에 의해 제공되는 기능들보다 심지어 더 중요할 것입니다.

Deref 트레잇을 가지고 스마트 포인터를 평범한 참조자와 같이 취급하기

Deref 트레잇을 구현하는 것은 우리가 (곱하기 혹은 글롭 연산자와는 반대 측에 있는) 역참조 연산자 (*dereference operator*) `*`의 동작을 커스터마이징 하는 것을 허용합니다. 스마트 포인터가 평범한 참조자처럼 취급될 수 있는 방식으로 **Deref**를 구현함으로써, 우리는 참조자에 대해 작동하는 코드를 작성하고 이 코드를 또한 스마트 포인터에도 사용할 수 있습니다.

먼저 `*`가 보통의 참조자와 어떤 식으로 동작하는지를 살펴보고, 그런 다음 `Box<T>` 와 비슷한 우리만의 타입을 정의하는 시도를 해서 왜 `*`가 우리의 새로 정의된 타입에서는 참조자처럼 작동하지 않는지를 봅시다. 우리는 **Deref** 트레잇을 구현하는 것이 어떻게 스마트 포인터가 참조자와 유사한 방식으로 동작하는 것을 가능하게 해 주는지를 탐구할 것입니다. 그런 뒤 러스트의 역참조 강제 (*deref coercion*) 기능과 이 기능이 어떻게 참조자 혹은 스마트 포인터와 함께 동작하도록 하는지 살펴보겠습니다.

*와 함께 포인터를 따라가서 값을 얻기

보통의 참조자는 포인터 타입이며, 포인터를 생각하는 한 가지 방법은 다른 어딘가에 저장된 값을 가리키는 화살표로서 생각하는 것입니다. Listing 15-6에서는 `i32` 값의 참조자를 생성하고는 참조자를 따라가서 값을 얻기 위해 역참조 연산자를 사용합니다:

Filename: src/main.rs

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-6: 역참조 연산자를 사용하여 `i32` 값에 대한 참조자를 따라가기

변수 `x`는 `i32` 값을 가지고 있습니다. `y`에는 `x`의 참조자를 설정했습니다. 우리는 `x`가 `5`와 동일함을 단언할 수 있습니다. 하지만, 만일 `y` 안의 값에 대한 단언을 만들고 싶다면, 참조자를 따라가서 이 참조자가 가리키고 있는 값을 얻기 위해 `*y`를 사용해야 합니다 (그래서 역참조라 합니다). 일단 `y`를 역참조하면, `5`와 비교 가능한 `y`가 가리키고 있는 정수 값에 접근하게 됩니다.

대신 `assert_eq!(5, y);`라고 작성하길 시도했다면, 아래와 같은 컴파일 에러를 얻을 것입니다:

```
error[E0277]: the trait bound `'{integer}: std::cmp::PartialEq<&{integer}>` is
not satisfied
--> src/main.rs:6:5
 |
6 |     assert_eq!(5, y);
|     ^^^^^^^^^^^^^^ can't compare `'{integer}` with `&{integer}`
|
= help: the trait `std::cmp::PartialEq<&{integer}>` is not implemented for
`{integer}`
```

숫자와 숫자에 대한 참조자를 비교하는 것은 허용되지 않는데 그 이유는 이들이 서로 다른 타입이기 때문입니다. `*`를 사용하여 해당 참조자를 따라가서 그것이 가리키고 있는 값을 얻어야 합니다.

Box<T>를 참조자처럼 사용하기

Listing 15-7에서 보는 바와 같이, Listing 15-6의 코드는 참조자 대신 `Box<T>`를 이용하여 재작성될 수 있으며, 역참조 연산자는 동일한 방식으로 작동될 것입니다:

Filename: src/main.rs

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-7: Box<i32> 상에 역참조 연산자 사용하기

Listing 15-7와 Listing 15-6 사이의 차이점은 오직 `x`의 값을 가리키는 참조자보다는 `x`를 가리키는 박스의 인스턴스로 `y`를 설정했다는 것입니다. 마지막 단언문에서, 우리는 `y`가 참조자일 때 했던 것과 동일한 방식으로 박스 포인터 앞에 역참조 연산자를 사용할 수 있습니다. 다음으로, 우리만의 박스 타입을 정의함으로써 `Box<T>`가 우리에게 역참조 연산자를 사용 가능하게끔 해주는 특별함이 무엇인지 탐구해 보겠습니다.

우리만의 스마트 포인터 정의하기

어떤 식으로 스마트 포인터가 기본적으로 참조자와는 다르게 동작하는지를 경험하기 위해서, 표준 라이브러리가 제공하는 `Box<T>` 타입과 유사한 스마트 포인터를 만들어 봅시다. 그런 다음 어떻게 역참조 연산자를 사용할 수 있는 기능을 추가하는지 살펴보겠습니다.

`Box<T>` 타입은 궁극적으로 하나의 요소를 가진 튜플 구조체로 정의되므로, Listing 15-8은 `MyBox<T>` 타입을 동일한 방식으로 정의하였습니다. 또한 `Box<T>`에 정의되어 있는 `new` 함수에 맞추기 위해 `new` 함수도 정의겠습니다:

Filename: src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

Listing 15-8: `MyBox<T>` 타입 정의하기

우리는 `MyBox`라는 이름의 구조체를 정의하고 제네릭 파라미터 `T`를 선언했는데, 이는 우리의 타입이 어떠한 종류의 타입 값이든 가질 수 있길 원하기 때문입니다. `MyBox` 타입은 `T` 타입의 하나의 요소를 가진 튜플 구조체입니다. `MyBox::new` 함수는 `T` 타입인 하나의 파라미터를 받아서 그 값을 갖는 `MyBox` 인스턴스를 반환합니다.

Listing 15-7의 `main` 함수를 Listing 15-8에 추가하고 `Box<T>` 대신 우리가 정의한 `MyBox<T>`를 이용하도록 수정해봅시다. Listing 15-9는 컴파일되지 않을 것인데 그 이유는 러스트가 `MyBox`를 어떻게 역참조 하는지 모르기 때문입니다:

Filename: src/main.rs

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-9: 참조자와 `Box<T>`를 사용한 것과 동일한 방식으로 `MyBox<T>` 사용 시도하기

아래는 그 결과 발생한 컴파일 에러입니다:

```
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
--> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);           ^
   |
```

우리의 `MyBox<T>` 타입은 역참조 될 수 없는데 그 이유는 우리의 타입에 대해 해당 기능을 아직 구현하지 않았기 때문입니다. ★ 연산자로 역참조를 가능케 하기 위해서, 우리는 `Deref` 트레잇을 구현합니다.

Deref 트레잇을 구현하여 임의의 타입을 참조자처럼 다루기

10장에서 논의한 바와 같이, 트레잇을 구현하기 위해서는 트레잇의 요구 메소드들에 대한 구현체를 제공할 필요가 있습니다. 표준 라이브러리가 제공하는 `Deref` 트레잇은 우리에게 `self`를 빌려서 내부 데이터에 대한 참조자를 반환하는 `deref`라는 이름의 메소드 하나를 구현하도록 요구합니다. Listing 15-10은 `MyBox`의 정의에 덧붙여 `Deref`의 구현을 담고 있습니다:

Filename: src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

Listing 15-10: `MyBox<T>` 상의 `Deref` 구현

`type Target = T;` 문법은 `Deref` 트레잇이 사용할 연관 타입 (associated type)을 정의합니다. 연관 타입은 제네릭 파라미터를 정의하는 것과 약간 다른 방식이지만, 여러분은 지금 이를 걱정할 필요는 없습니다; 우리는 이를 19장에서 더 자세히 다룰 것입니다.

우리는 `deref` 메소드의 본체를 `&self.0`로 채웠으므로 `deref`는 우리가 ★ 연산자를 이용해 접근하고자 하는 값의 참조자를 반환합니다. `MyBox<T>` 값에 대하여 ★을 호출하는 Listing 15-9의 `main` 함수는 이제 컴파일되고 단언문은 통과됩니다!

`Deref` 트레잇 없이, 컴파일러는 오직 `&` 참조자들만 역참조 할 수 있습니다. `deref` 메소드는 컴파일러에게 `Deref`를 구현한 어떠한 타입의 값을 가지고 `&` 참조자를 가져오기 위해서 어떻게 역참조 하는지 알고 있는 `deref` 메소드를 호출하는 기능을 부여합니다.

Listing 15-9의 ★에 들어설 때, 무대 뒤에서 러스트는 실제로 아래의 코드를 실행했습니다:

```
*(y.deref())
```

러스트는 ★ 연산자에 `deref` 메소드 호출 후 보통의 역참조를 대입하므로 프로그래머로서 우리는 `deref`

메소드를 호출할 필요가 있는지 혹은 없는지를 생각하지 않아도 됩니다. 이 러스트의 기능은 우리가 보통의 참조자를 가지고 있는 경우 혹은 **Deref** 를 구현한 타입을 가지고 있는 경우에 대하여 동일하게 기능하는 코드를 작성하도록 해 줍니다.

deref 메소드가 값의 참조자를 반환하고 `*(y.deref())` 에서의 괄호 바깥의 평범한 역참조가 여전히 필요한 이유는 소유권 시스템 때문입니다. 만일 **deref** 메소드가 값의 참조자 대신 값을 직접 반환했다면, 그 값은 **self** 바깥으로 이동될 것입니다. 위의 경우 및 우리가 역참조 연산자를 사용하는 대부분의 경우에서 우리는 `MyBox<T>` 내부의 값에 대한 소유권을 얻길 원치 않습니다.

우리의 코드에 `*` 를 한번 타이핑할 때마다, `*` 는 **deref** 함수의 호출 후 `*` 를 한번 호출하는 것으로 대체된다는 점을 기억하세요. `*` 의 대입이 무한히 재귀적으로 실행되지 않기 때문에, 우리는 결국 `i32` 타입의 데이터를 얻는데, 이는 Listing 15-9의 `assert_eq!` 내의 5와 일치합니다.

함수와 메소드를 이용한 암묵적 역참조 강제

역참조 강제(*deref coercion*) 는 러스트가 함수 및 메소드의 인자에 수행하는 편의성 기능입니다. 역참조 강제는 **Deref** 를 구현한 어떤 타입의 참조자를 **Deref** 가 본래의 타입으로부터 바꿀 수 있는 타입의 참조자로 바꿔줍니다. 역참조 강제는 우리가 특정 타입의 값에 대한 참조자를 함수 혹은 메소드의 인자로 넘기는 중정의된 파라미터 타입에는 맞지 않을 때 자동적으로 발생합니다. 일련의 **deref** 메소드 호출은 우리가 제공한 타입을 파라미터가 요구하는 타입으로 변경해 줍니다.

역참조 강제가 러스트에 도입되어서 함수와 메소드 호출을 작성하는 프로그래머들은 `&` 와 `*` 를 이용한 많은 수의 명시적 참조 및 역참조를 추가하지 않아도 됩니다. 역참조 강제 기능은 또한 우리가 참조자나 스마트 포인터 둘 중 어느 경우라도 작동할 수 있는 코드를 더 많이 작성할 수 있도록 해 줍니다.

역참조 강제가 실제 작동하는 것을 보기 위해서, 우리가 Listing 15-8에서 정의했던 `MyBox<T>` 과 Listing 15-10에서 추가했던 **Deref** 의 구현체를 이용합시다. Listing 15-11은 스트링 슬라이스 파라미터를 갖는 함수의 정의를 보여줍니다:

Filename: src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

Listing 15-11: 타입 `&str` 의 `name` 이라는 파라미터를 갖는 `hello` 함수

우리는 예를 들면 `hello("Rust");` 와 같이 스트링 슬라이스를 인자로 하여 `hello` 함수를 호출할 수 있습니다. Listing 15-12에서 보는 바와 같이, 역참조 강제는 `MyBox<String>` 타입의 값에 대한 참조자를 이용하여 `hello` 를 호출하는 것을 가능하게 해 줍니다:

Filename: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

Listing 15-12: 역참조 강제 때문에 작동되는, `MyBox<String>` 값에 대한 참조자로 `hello` 호출하기

여기서 우리는 `hello` 함수를 호출하는 인자로서 `&m`를 이용했는데, 이는 `MyBox<String>`의 참조자입니다. 우리가 Listing 15-10에서 `MyBox<T>`의 `Deref` 트레잇을 구현했기 때문에, 러스트는 `deref`를 호출하여 `&MyBox<String>`을 `&String`으로 바꿀 수 있습니다. 표준 라이브러리는 스트링 슬라이스를 반환하는 `String`의 `Deref` 구현체를 제공하는데, 이는 `Deref`에 대한 API 문서에도 있습니다. 러스트는 `deref`를 다시 한번 호출하여 `&String`을 `&str`로 변환하고, 이는 `hello` 함수의 정의와 일치하게 됩니다.

만일 러스트가 역참조 강제 기능을 구현하지 않았다면, 우리는 `&MyBox<String>` 타입의 값을 가지고 `hello` 함수를 호출하는 데 있어 Listing 15-12의 코드 대신 Listing 15-13의 코드를 작성해야 했을 것입니다:

Filename: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

Listing 15-13: 만일 러스트에 역참조 강제가 없었다면 우리가 작성했어야 했을 코드

`(*m)`은 `MyBox<String>`을 `String`로 역참조해 줍니다. 그런 다음 `&`과 `[..]`은 `hello` 시그니처와 일치되도록 전체 스트링과 동일한 `String`의 스트링 슬라이스를 얻습니다. 역참조 강제가 없는 코드는 이러한 모든 기호들이 수반된 상태에서 읽기도, 쓰기도, 이해하기도 더 힘들어집니다. 역참조 강제는 러스트가 우리를 위해 이러한 변환을 자동적으로 다룰 수 있도록 해 줍니다.

`Deref` 트레잇이 관련된 타입에 대해 정의될 때, 러스트는 해당 타입을 분석하여 파라미터의 타입에 맞는 참조자를 얻기 위해 필요한 수만큼의 `Deref::deref`를 사용할 것입니다. `Deref::deref`가 삽입될 필요가 있는 횟수는 컴파일 타임에 분석되므로, 역참조 강제의 이점을 얻는 데에 관해서 어떠한 런타임 페널티도 없습니다!

역참조 강제가 가변성과 상호작용 하는 법

불변 참조자에 대한 `*`를 오버 라이딩하기 위해 `Deref` 트레이잇을 이용하는 방법과 비슷하게, 러스트는 가변 참조자에 대한 `*`를 오버 라이딩하기 위한 `DerefMut` 트레이잇을 제공합니다.

러스트는 다음의 세 가지 경우에 해당하는 타입과 트레이잇 구현을 찾았을 때 역참조 강제를 수행합니다:

- `T: Deref<Target=U>` 일때 `&T`에서 `&U`로
- `T: DerefMut<Target=U>` 일때 `&mut T`에서 `&mut U`로
- `T: Deref<Target=U>` 일때 `&mut T`에서 `&U`로

첫 두 가지 경우는 가변성 부분만 제외하고는 동일합니다. 첫 번째 경우는 만일 여러분이 `&T`를 가지고 있고, `T`가 어떤 타입 `U`에 대한 `Deref`를 구현했다면, 여러분은 명료하게 `&U`를 얻을 수 있음을 기술하고 있습니다. 두 번째 경우는 동일한 역참조 강제가 가변 참조자에 대해서도 발생함을 기술합니다.

세 번째 경우는 좀 더 교묘합니다: 러스트는 가변 참조자를 불변 참조자로 강제할 수도 있습니다. 하지만 그 역은 불가능합니다: 불변 참조자는 가변 참조자로 결코 강제되지 않을 것입니다. 빌림 규칙 때문에, 만일 여러분이 가변 참조자를 가지고 있다면, 그 가변 참조자는 해당 데이터에 대한 유일한 참조자임에 틀림없습니다 (만일 그렇지 않다면, 그 프로그램은 컴파일되지 않을 것입니다). 가변 참조자를 불변 참조자로 변경하는 것은 결코 빌림 규칙을 깨트리지 않을 것입니다. 불변 참조자를 가변 참조자로 변경하는 것은 해당 데이터에 대한 단 하나의 불변 참조자가 있어야 한다는 요구를 하게 되고, 이는 빌림 규칙이 보장해줄 수 없습니다. 따라서, 러스트는 불변 참조자를 가변 참조자로 변경하는 것이 가능하다는 가정을 할 수 없습니다.

Drop 트레잇은 메모리 정리 코드를 실행시킵니다

스마트 포인터 패턴에서 중요한 두 번째 트레잇은 `Drop` 인데, 이는 값이 스코프 밖으로 벗어나려고 할 때 어떤 일이 발생될지를 커스터마이징하게끔 해줍니다. 우리는 어떠한 타입이 든 간에 `Drop` 트레잇을 위한 구현을 제공할 수 있고, 우리가 특정한 코드는 파일이나 네트워크 연결 같은 자원을 해제하는 데에 사용될 수 있습니다. 우리는 스마트 포인터의 맥락 안에서 `Drop`을 소개하고 있는데 그 이유는 `Drop` 트레잇의 기능이 언제나 대부분 스마트 포인터를 구현할 때에 사용되기 때문입니다. 예를 들면, `Box<T>`는 박스가 가리키고 있는 힙 상의 공간을 할당 해제하기 위해 `Drop`을 커스터마이징 합니다.

몇몇 언어들에서, 프로그래머는 스마트 포인터의 인스턴스 사용을 종료하는 매번 마다 메모리 혹은 자원을 해제하기 위해 코드를 호출해야 합니다. 만일 이를 잊어버리면, 그 시스템은 과부하가 걸리거나 멈출지도 모릅니다. 러스트에서는 값이 스코프 밖으로 벗어날 때마다 실행되어야 하는 특정한 코드 조각을 특정할 수 있고, 컴파일러는 이 코드를 자동으로 삽입해줄 것입니다. 결과적으로, 우리는 프로그램 내에서 특정한 타입의 인스턴스가 종료되는 곳마다 정리 코드를 집어넣는 것에 관한 걱정을 할 필요가 없지만, 여전히 자원 누수는 발생하지 않을 것입니다!

`Drop` 트레잇을 구현함으로써 값이 스코프 밖으로 벗어났을 때 실행될 코드를 특정합니다. `Drop` 트레잇은 `self`에 대한 가변 참조자를 파라미터로 갖는 `drop`이라는 이름의 하나의 메소드를 구현하도록 우리에게 요구합니다. 러스트가 언제 `drop`을 호출하는지 보기 위해서, 지금은 `println!` 구문과 함께 `drop`을 구현해봅시다.

Listing 15-4는 인스턴스가 스코프 밖으로 벗어났을 때 `Dropping CustomSmartPointer!`를 출력하는 커스텀 기능만을 갖춘 `CustomSmartPointer` 구조체를 보여주고 있습니다. 이 예제는 러스트가 `drop` 함수를 실행시키는 때를 보여줍니다:

Filename: src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer { data: String::from("my stuff") };
    let d = CustomSmartPointer { data: String::from("other stuff") };
    println!("CustomPointers created.");
}
```

Listing 15-14: 우리의 정리 코드를 넣을 수 있는 `Drop` 트레이잇을 구현한 `CustomSmartPointer` 구조체

`Drop` 트레이잇은 프렐루드에 포함되어 있으므로, 이를 가져오지 않아도 됩니다. 우리는 `CustomSmartPointer` 상에 `Drop` 트레이잇을 구현하였고, `println!`을 호출하는 `drop` 메소드 구현을 제공했습니다. `drop` 함수의 본체는 여러분이 만든 타입의 인스턴스가 스코프 밖으로 벗어났을 때 실행시키고자 하는 어떠한 로직이라도 위치시킬 수 있는 곳입니다. 우리는 여기서 러스트가 `drop`을 호출하게 될 때를 보여주기 위해서 어떤 텍스트를 출력하는 중입니다.

`main`에서는 두 개의 `CustomSmartPointer` 인스턴스를 만든 다음 `CustomSmartPointers created.`를 출력합니다. `main`의 끝에서, 우리의 `CustomSmartPointer` 인스턴스는 스코프 밖으로 벗어날 것이고, 러스트는 우리가 `drop` 메소드 내에 집어넣은 코드, 즉 우리의 마지막 메시지를 출력하는 코드를 호출할 것입니다. 우리가 `drop` 메소드를 명시적으로 호출할 필요가 없다는 점을 주의하세요.

이 프로그램을 실행시켰을 때, 다음과 같은 출력을 보게 될 것입니다:

```
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!
```

러스트는 우리의 인스턴스가 스코프 밖으로 벗어났을 때 우리를 위하여 `drop`을 호출했고, 우리가 특정한 그 코드를 호출하게 됩니다. 변수들은 만들어진 순서의 역순으로 버려지므로, `d`는 `c` 전에 버려집니다. 이 예제는 여러분에게 `drop` 메소드가 어떻게 동작하는지에 대한 시각적인 가이드만을 제공하지만, 여러분은 보통 메시지 출력보다는 여러분의 타입이 실행할 필요가 있는 정리 코드를 특정할 것입니다.

`std::mem::drop`을 이용하여 값을 일찍 버리기

불행하게도, 자동적인 `drop` 기능을 비활성화하는 것은 직관적이지 않습니다. `drop` 비활성화는 보통 필요가 없습니다; `Drop` 트레이잇의 전체적 관점은 자동적으로 다루어진다는 것입니다. 가끔, 여러분은 값을 일찍 정리하기를 원할지도 모릅니다. 한 가지 예는 락을 관리하는 스마트 포인터를 이용할 때입니다: 여러분은 실행할 락을 해제하는 `drop` 메소드를 강제로 실행시켜서 같은 스코프 내의 다른 코드가 락을 얻을 수 있길 원할지도 모릅니다. 러스트는 우리가 수동으로 `Drop` 트레이잇의 `drop` 메소드를 호출하도록 해주지 않습니다; 대신 우리가 스코프 밖으로 벗어나기 전에 값이 강제로 버려질 원한다면 표준 라이브러리에서 제공하는 `std::mem::drop` 함수를 호출해야 합니다.

Listing 15-14의 `main` 함수를 Listing 15-15처럼 수정하여 `Drop` 트레이잇의 `drop` 메소드를 호출하려고 하면 어떤 일이 벌어지는지 봅시다:

Filename: src/main.rs

```
fn main() {
    let c = CustomSmartPointer { data: String::from("some data") };
    println!("CustomSmartPointer created.");
    c.drop();
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Listing 15-15: 메모리 정리를 일찍 하기 위해 `Drop` 트레이트로부터 `drop` 메소드를 호출 시도하기

이 코드의 컴파일을 시도하면, 다음과 같은 에러를 얻게 됩니다:

```
error[E0040]: explicit use of destructor method
--> src/main.rs:14:7
   |
14 |     c.drop();
   |     ^^^^^ explicit destructor calls not allowed
```

이 에러 메시지는 우리가 `drop`를 명시적으로 호출하는 것이 허용되지 않음을 기술하고 있습니다. 에러 메시지는 소멸자 (*destructor*)라는 용어를 사용하는데, 이는 인스턴스를 정리하는 함수에 대한 일반적인 프로그래밍 용어입니다. 소멸자는 인스턴스를 생성하는 생성자 (*constructor*)와 비슷합니다. 러스트 내의 `drop` 함수는 특정한 형태의 소멸자입니다.

러스트는 우리가 `drop`을 명시적으로 호출하도록 해주지 않는데 이는 러스트가 `main`의 끝에서 값에 대한 `drop` 호출을 여전히 자동적으로 할 것이기 때문입니다. 이는 러스트가 동일한 값을 두 번 메모리 정리를 시도할 수 있기 때문에 중복 해제 (*double free*) 에러가 될 수 있습니다.

우리는 값이 스코프 밖으로 벗어났을 때 자동적인 `drop` 추가를 비활성화할 수 없고, `drop` 메소드를 명시적으로 호출할 수도 없습니다. 따라서, 값이 일찍 메모리 정리되도록 강제하길 원한다면, `std::mem::drop` 함수를 이용할 수 있습니다.

`std::mem::drop` 함수는 `Drop` 트레이트 내에 있는 `drop` 메소드와 다릅니다. 우리가 일찍 버리도록 강제하길 원하는 값을 인자로 넘김으로써 이를 호출할 수 있습니다. 이 함수는 프렐루드에 포함되어 있으므로, 우리는 Listing 15-15의 `main`을 Listing 15-16에서 보는 것처럼 수정할 수 있습니다:

Filename: src/main.rs

```
fn main() {
    let c = CustomSmartPointer { data: String::from("some data") };
    println!("CustomSmartPointer created.");
    drop(c);
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Listing 15-16: 값이 스코프 밖으로 벗어나기 전에 명시적으로 버리기 위한 `std::mem::drop` 호출하기

이 코드의 실행은 다음을 출력할 것입니다:

```
CustomSmartPointer created.  
Dropping CustomSmartPointer with data `some data`!  
CustomSmartPointer dropped before the end of main.
```

`Dropping CustomSmartPointer with data `some data`!`라는 텍스트가 `CustomSmartPointer created.` 와 `CustomSmartPointer dropped before the end of main.` 사이에 출력되는데, 이는 `c`를 그 지점에서 버리기 위해 `drop` 메소드 코드가 호출되었음을 보여줍니다.

우리는 메모리 정리를 편리하고 안전하게 하기 위하여 `Drop` 트레이트 구현체 내에 특정된 코드를 다양한 방식으로 이용할 수 있습니다: 예를 들면, 이것을 우리만의 고유한 메모리 할당자를 만들기 위해 사용할 수도 있습니다! `Drop` 트레이트와 러스트의 소유권 시스템을 이용하면, 러스트가 메모리 정리를 자동적으로 수행하기 때문에 메모리 정리를 기억하지 않아도 됩니다.

우리는 또한 계속 사용 중인 값이 뜻하지 않게 정리되는 것을 걱정하지 않아도 되는데, 그런 것은 컴파일 에러를 야기할 것이기 때문입니다: 참조자가 항상 유효하도록 확실히 해주는 소유권 시스템은 또한 값이 더 이상 사용되지 않을 때 `drop`이 오직 한 번만 호출될 것을 보장합니다.

지금까지 `Box<T>` 와 스마트 포인터의 몇 가지 특성을 시험해 보았으니, 표준 라이브러리에 정의되어 있는 다른 몇 가지의 스마트 포인터를 살펴봅시다.

Rc<T>, 참조 카운팅 스마트 포인터

대부분의 경우에서, 소유권은 명확합니다: 여러분은 어떤 변수가 주어진 값을 소유하는지 정확히 압니다. 그러나, 하나의 값이 여러 개의 소유자를 가질 수도 있는 경우가 있습니다. 예를 들면, 그래프 데이터 구조에서, 여러 에지가 동일한 노드를 가리킬 수도 있고, 그 노드는 개념적으로 해당 노드를 가리키는 모든 에지들에 의해 소유됩니다. 노드는 어떠한 에지도 이를 가리키지 않을 때까지는 메모리 정리가 되어서는 안됩니다.

복수 소유권을 가능하게 하기 위해서, 러스트는 **Rc<T>** 라 불리우는 타입을 가지고 있습니다. 이 이름은 참조 카운팅 (*reference counting*)의 약자인데, 이는 어떤 값이 계속 사용되는지 혹은 그렇지 않은지를 알기 위해 해당 값에 대한 참조자의 갯수를 계속 추적하는 것입니다. 만일 값에 대한 참조자가 0개라면, 그 값은 어떠한 참조자도 무효화하지 않고 메모리 정리될 수 있습니다.

Rc<T>를 거실의 TV로 상상해보세요. 만일 한 사람이 TV를 보러 들어온다면, TV를 킁니다. 다른 사람들은 거실로 들어와서 TV를 볼 수 있습니다. 마지막 사람이 거실을 나선다면, TV는 더 이상 사용되지 않으므로 이를 끍니다. 만일 다른 사람들이 여전히 TV를 보고 있는 중에 누군가가 이를 끈다면, 남은 TV 시청자들로부터 엄청난 소란이 있을 것입니다!

우리 프로그램의 여러 부분에서 읽을 데이터를 힙에 할당하고 싶고, 어떤 부분이 그 데이터를 마지막에 이용하게 될지 컴파일 타임에는 알 수 없는 경우 **Rc<T>** 타입을 사용합니다. 만일 어떤 부분이 마지막으로 사용하는지 알 수 있다면, 우리는 그냥 그 해당 부분을 데이터의 소유자로 만들면 되고, 컴파일 타임에 집행되는 보통의 소유권 규칙이 효과를 발생시킬 것입니다.

Rc<T>가 오직 단일 스레드 시나리오 상에서만 사용 가능하다는 점을 주의하세요. 16장에서 동시성 (concurrency)을 논의할 때, 다중 스레드 프로그램에서는 어떻게 참조 카운팅을 하는지 다루겠습니다.

Rc<T>를 사용하여 데이터 공유하기

Listing 15-5의 cons list 예제로 돌아가 봅시다. 우리는 **Box<T>**를 이용해서 이것을 정의했던 것을 상기하세요. 이번에는 세 번째 리스트의 소유권을 둘 다 공유하는 두 개의 리스트를 만들 것인데, 이는 개념적으로 Figure 15-3과 유사하게 보일 것입니다:

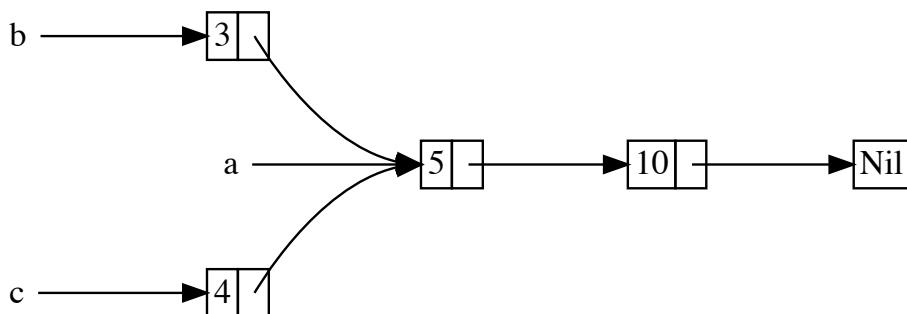


Figure 15-3: 세 번째 리스트 **a**의 소유권을 공유하는 두 리스트 **b**와 **c**

우리는 5와 10을 담은 리스트 **a**를 만들 것입니다. 그런 다음 두 개의 리스트를 더 만들 것입니다: 3으로 시작하는 **b**와 4로 시작하는 **c**입니다. 그리고 나서 **b**와 **c** 리스트 둘 모두 5와 10을 담고 있는 첫번째 **a** 리스트로 계속되게 할 것입니다. 바꿔 말하면, 두 리스트 모두 5와 10을 담은 첫 리스트를 공유하게 될 것입니다.

Listing 15-17에서 보시는 것처럼, 우리가 **Box<T>**를 가지고 정의한 **List**를 이용하여 이 시나리오를 구현하는 시도는 작동하지 않을 것입니다:

Filename: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::*;

fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));

    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

Listing 15-17: **Box<T>**를 이용한 두 리스트가 세 번째 리스트의 소유권을 공유하는 시도는 허용되지 않음을 보이는 예

이 코드를 컴파일하면, 다음과 같은 에러를 얻습니다:

```
error[E0382]: use of moved value: `a`
--> src/main.rs:13:30
   |
12 |     let b = Cons(3, Box::new(a));
      |             - value moved here
13 |     let c = Cons(4, Box::new(a));
      |             ^ value used here after move
   |
= note: move occurs because `a` has type `List`, which does not implement
the `Copy` trait
```

Cons variant는 이것이 가지고 있는 데이터를 소유하므로, 우리가 **b** 리스트를 만들 때, **a**는 **b** 안으로 이동되고 **b**는 **a**를 소유합니다. 그 뒤, **c**를 생성할 때 **a**를 다시 이용하는 시도를 할 경우, 이는 **a**가 이동되

었으므로 허용되지 않습니다.

우리는 `Cons` 가 대신 참조자를 갖도록 정의를 변경할 수도 있지만, 그러면 라이프타임 파라미터를 명시해야 할 것입니다. 라이프타임 파라미터를 명시함으로써, 리스트 내의 모든 요소들이 최소한 전체 리스트만큼 오래 살아있도록 명시될 것입니다. 빌림 검사기는 예를 들면 `let a = Cons(10, &Nil);` 을 컴파일되도록 하지 않게 할텐데, 이는 일시적인 `Nil` 값은 `a` 가 그에 대한 참조자를 가질 수도 있는 시점 이전에 버려질 것이기 때문입니다.

대신, 우리는 Listing 15-18과 같이 `Box<T>` 의 자리에 `Rc<T>` 를 이용하여 `List` 의 정의를 바꿀 것입니다. 각각의 `Cons` variant는 이제 어떤 값과 `List` 를 가리키는 `Rc<T>` 를 갖게 될 것입니다. `b` 를 만들때는 `a` 의 소유권을 얻는 대신, `a` 를 가지고 있는 `Rc<List>` 를 클론할 것인데, 이는 참조자의 갯수를 하나에서 둘로 증가시키고 `a` 와 `b` 가 `Rc<List>` 안에 있는 값을 공유하게 해줍니다. 우리는 또한 `c` 를 만들때도 `a` 를 클론할 것인데, 이는 참조자의 갯수를 둘에서 셋으로 늘립니다. 우리가 `Rc::clone` 을 호출하는 매번마다, 해당 `Rc<List>` 가 가지고 있는 데이터에 대한 참조 카운트는 증가할 것이고, 그 데이터는 참조자가 0 개가 되지 않으면 메모리 정리되지 않을 것입니다:

Filename: src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Listing 15-18: `Rc<T>` 를 이용하는 `List` 정의

`Rc<T>` 는 프렐루드에 포함되어 있지 않으므로 우리는 이를 가져오기 위해 `use` 구문을 추가할 필요가 있습니다. `main` 내에서, 우리는 5와 10을 가지고 있는 리스트를 만들어서 이를 `a` 의 새로운 `Rc<List>` 에 저장합니다. 그 다음 `b` 와 `c` 를 만들 때, 우리는 `Rc::clone` 함수를 호출하고 `a` 의 `Rc<List>` 에 대한 참조자를 인자로서 넘깁니다.

`Rc::clone(&a)` 보다는 `a.clone()` 을 호출할 수도 있지만, 위의 경우 러스트의 관례는 `Rc::clone` 를 이용하는 것입니다. `Rc::clone` 의 구현체는 대부분의 타입들의 `clone` 구현체들이 하는 것처럼 모든 데이터의 깊은 복사 (deep copy) 를 만들지 않습니다. `Rc::clone` 의 호출은 오직 참조 카운트만 증가시키는데, 이는 큰 시간이 들지 않습니다. 데이터의 깊은 복사는 많은 시간이 걸릴 수 있습니다. 참조 카운팅을 위

해 `Rc::clone`을 이용함으로써, 우리는 깊은 복사 종류의 클론과 참조 카운트를 증가시키는 종류의 클론을 시각적으로 구별할 수 있습니다. 코드 내에서 성능 문제를 찾고 있다면, 깊은 복사 클론만 고려할 필요가 있고 `Rc::clone` 호출은 무시할 수 있는 것입니다.

`Rc<T>`의 클론 생성은 참조 카운트를 증가시킵니다

Listing 15-18의 동작 예제를 변경하여 `a` 내부의 `Rc<List>`에 대한 참조자가 생성되고 드롭될 때 참조 카운트의 변화를 볼 수 있도록 해봅시다.

Listing 15-19에서는 `main`을 변경하여 리스트 `c`를 감싸고 있는 내부 스코프를 갖도록 하겠습니다; 그런 다음 우리는 `c`가 스코프 밖으로 벗어났을 때 참조 카운트가 어떻게 변하는지 볼 수 있습니다. 프로그램 내 참조 카운트가 변하는 각 지점에서, 우리는 참조 카운트 값을 출력할텐데, 이는 `Rc::strong_count` 함수를 호출함으로써 얻을 수 있습니다. 이 함수는 `count` 보다는 `strong_count`라는 이름을 갖고 있는데 이는 `Rc<T>` 타입이 `weak_count`도 갖고 있기 때문입니다; `weak_count`가 무엇을 위해 사용되는지는 “참조 순환 (reference cycles) 방지하기”절에서 볼 것입니다.

Filename: src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

Listing 15-19: 참조 카운트 출력하기

이 코드는 다음을 출력합니다:

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

우리는 `a`의 `Rc<List>` 가 초기 참조 카운트로서 1을 갖는 것을 볼 수 있습니다; 그 다음 우리가 `clone`을 호출하는 매번마다, 카운트는 1씩 증가합니다. `c`가 스코프 밖으로 벗어날 때, 카운트는 1만큼 감소합니다. 우리는 참조 카운트를 증가시키기 위해서 `Rc::clone`를 호출해야 하는 것과 같이 참조 카운트를 감소시키

기 위해 어떤 함수를 호출하지 않아도 됩니다: `Rc<T>` 값이 스코프 밖으로 벗어나면 `Drop` 트레이트의 구현체가 자동으로 참조 카운트를 감소시킵니다.

이 예제에서 볼수 없는 것은 `main`의 끝에서 `b`와 그 다음 `a`가 스코프 밖을 벗어나서, 카운트가 0이 되고, 그 시점에서 `Rc<List>`가 완전히 메모리 정리되는 때입니다. `Rc<T>`를 이용하는 것은 단일값이 복수 개의 소유자를 갖도록 허용해주고, 이 카운트는 소유자중 누구라도 여전히 존재하는 한 값이 계속 유효함을 확실히 해줍니다.

불변 참조자를 통하여, `Rc<T>`는 읽기 전용으로 우리 프로그램의 여러 부분 사이에서 데이터를 공유하도록 허용해줍니다. 만일 `Rc<T>`가 또한 복수개의 가변 참조자도 갖는 것을 허용한다면, 우리는 4장에서 논의했던 빌림 규칙 중 하나를 위반할지도 모릅니다: 동일한 위치에 대한 복수개의 가변 빌림은 데이터 레이스 및 데이터 불일치를 야기할 수 있다는 것입니다. 하지만 데이터의 변형을 가능하게 하는 것은 매우 유용하죠! 다음 절에서는 내부 가변성 (interior mutability) 패턴과 이러한 불변성 제약과 함께 동작하기 위해 `Rc<T>`와 같이 결합하여 사용할 수 있는 `RefCell<T>` 타입에 대해 논의할 것입니다.

RefCell<T>와 내부 가변성 패턴

내부 가변성 (*interior mutability*)은 어떤 데이터에 대한 불변 참조자가 있을 때라도 여러분이 데이터를 변형할 수 있게 해주는 러스트의 디자인 패턴입니다: 보통 이러한 동작은 빌림 규칙에 의해 허용되지 않습니다. 그렇게 하기 위해서, 이 패턴은 변형과 빌림을 지배하는 러스트의 통상적인 규칙을 구부리기 위하여 데이터 구조 내에서 **unsafe** (안전하지 않은) 코드를 사용합니다. 우리는 아직 안전하지 않은 코드를 다루지 않습니다; 이는 19장에서 다룰 것입니다. 우리가 런타임에 빌림 규칙을 따를 것임을 보장할 수 있을 때라면, 심지어 컴파일러가 이를 보장하지 못하더라도 내부 가변성 패턴을 이용하는 타입을 사용할 수 있습니다. 포함되어 있는 **unsafe** 코드는 안전한 API로 감싸져 있고, 외부 타입은 여전히 불변입니다.

내부 가변성 패턴을 따르는 **RefCell<T>** 타입을 살펴보는 것으로 이 개념을 탐구해 봅시다.

RefCell<T>을 가지고 런타임에 빌림 규칙을 집행하기

Rc<T> 와는 다르게, **RefCell<T>** 타입은 가지고 있는 데이터 상에 단일 소유권을 나타냅니다. 그렇다면, **Box<T>** 와 같은 타입에 비교해 **RefCell<T>** 의 다른 부분은 무엇일까요? 여러분이 4장에서 배웠던 빌림 규칙을 상기해보세요:

- 어떠한 경우이든 간에, 여러분은 다음의 둘 다는 아니고 둘 중 하나만 가질 수 있습니다: 하나의 가변 참조자 혹은 임의 개수의 불변 참조자들을요.
- 참조자는 항상 유효해야 합니다.

참조자와 **Box<T>** 를 이용할 때, 빌림 규칙의 불변성은 컴파일 타임에 집행됩니다. **RefCell<T>** 를 이용할 때, 이 불변성은 런타임에 집행됩니다. 참조자를 가지고서 여러분이 이 규칙을 어기면 컴파일러 에러를 얻게 될 것입니다. **RefCell<T>** 를 가지고서 여러분이 이 규칙을 어기면, 여러분의 프로그램은 **panic!** 을 일으키고 종료될 것입니다.

컴파일 타임에 빌림 규칙을 검사하는 것은 개발 과정에서 에러를 더 일찍 잡을 수 있다는 점, 그리고 이 모든 분석이 사전에 완료되기 때문에 런타임 성능에 영향이 없다는 점에서 장점을 가집니다. 이러한 까닭에, 대부분의 경우 컴파일 타임에서 빌림 규칙을 검사하는 것이 가장 좋은 선택이고, 이것이 러스트의 기본 설정인 이유이기도 합니다.

대신 런타임에 빌림 규칙을 검사하는 것은 컴파일 타임 검사에 의해서는 허용되지 않는, 특정한 메모리 안정성 시나리오가 허용된다는 잇점이 있습니다. 러스트 컴파일러와 같은 정적 분석은 태생적으로 보수적입니다. 어떤 코드 속성은 코드의 분석을 이용해서는 발견이 불가능합니다: 가장 유명한 예제는 정지 문제 (halting problem) 인데, 이는 이 책의 범위를 벗어나지만 연구하기에 흥미로운 주제입니다.

몇몇 분석이 불가능하기 때문에, 만일 코드가 소유권 규칙을 준수한다는 것을 러스트 컴파일러가 확신할 수 없다면, 컴파일러는 올바른 프로그램을 거부할지도 모릅니다; 이렇게 하여, 컴파일러는 보수적입니다. 만일 러스트가 올바르지 않은 프로그램을 받아들이면, 사용자들은 러스트가 보장하는 것을 신뢰할 수 없을 것입니다.

다. 하지만, 만일 러스트가 올바른 프로그램을 거부한다면, 프로그래머는 불편해할 것이지만, 어떠한 재앙도 일어나지 않을 수 있습니다. `RefCell<T>` 타입은 여러분의 코드가 빌림 규칙을 따르는 것을 여러분이 확신 하지만, 컴파일러는 이를 이해하고 보장할 수 없을 경우 유용합니다.

`Rc<T>` 와 유사하게, `RefCell<T>` 은 단일 스레드 시나리오 내에서만 사용 가능하고, 만일 여러분이 이를 다중 스레드 맥락 내에서 사용을 시도할 경우 여러분에게 컴파일 타임 에러를 줄 것입니다. `RefCell<T>` 의 기능을 다중 스레드 프로그램 내에서 사용하는 방법에 대해서는 16장에서 이야기할 것입니다.

`Box<T>`, `Rc<T>`, 혹은 `RefCell<T>` 을 선택하는 이유의 요점은 다음과 같습니다:

- `Rc<T>` 는 동일한 데이터에 대해 복수개의 소유자를 가능하게 합니다; `Box<T>` 와 `RefCell<T>` 은 단일 소유자만 갖습니다.
- `Box<T>` 는 컴파일 타임에 검사된 불변 혹은 가변 빌림을 허용합니다; `Rc<T>` 는 오직 컴파일 타임에 검사된 불변 빌림만 허용합니다; `RefCell<T>` 는 런타임에 검사된 불변 혹은 가변 빌림을 허용합니다.
- `RefCell<T>` 이 런타임에 검사된 가변 빌림을 허용하기 때문에, `RefCell<T>` 이 불변일 때라도 `RefCell<T>` 내부의 값을 변경할 수 있습니다.

불변값 내부의 값을 변경하는 것을 **내부 가변성 패턴**이라고 합니다. 내부 가변성이 유용한 경우를 살펴보고 이것이 어떻게 가능한지 조사해 봅시다.

내부 가변성: 불변값에 대한 가변 빌림

빌림 규칙의 결과로 인해 우리는 불변값을 가지고 있을 때 이를 변경 가능하게 빌릴 수 없습니다. 예를 들면, 다음 코드는 컴파일되지 않을 것입니다:

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

이 코드의 컴파일을 시도하면, 다음과 같은 에러를 얻을 것입니다:

```
error[E0596]: cannot borrow immutable local variable `x` as mutable
--> src/main.rs:3:18
   |
2 |     let x = 5;
   |         - consider changing this to `mut x`
3 |     let y = &mut x;
   |             ^ cannot borrow mutably
```

하지만, 값이 자신의 메소드 내부에서 변경되지만 다른 코드에서는 불변인 것으로 보이는 것이 유용할 수 있는 경우가 있습니다. 그 값의 메소드 바깥의 코드는 값을 변경할 수 없을 것입니다. `RefCell<T>`을 이용하는 것은 내부 가변성의 기능을 얻는 한가지 방법입니다. 그러나 `RefCell<T>`은 빌림 규칙을 완벽하게 피하는 것은 아닙니다: 컴파일러 내의 빌림 검사기는 이러한 내부 가변성을 허용하고, 빌림 규칙은 대신 런타임에 검사됩니다. 만일 이 규칙을 위반하면, 우리는 컴파일러 에러 대신 `panic!`을 얻을 것입니다.

불변 값을 변경하기 위해 `RefCell<T>`를 이용할 수 있는 실질적인 예제를 살펴보고 이것이 왜 유용한지를 알아봅시다.

내부 가변성에 대한 용례: 목(mock) 객체

테스트 더블 (*test double*)은 테스트하는 동안 또 다른 타입을 대신하여 사용되는 타입을 위한 일반적인 프로그래밍 개념입니다. 목 객체 (*mock object*)는 테스트 중 어떤 일이 일어났는지 기록하여 정확한 동작이 일어났음을 단언할 수 있도록 하는 테스트 더블의 특정한 타입입니다.

러스트는 다른 언어들이 객체를 가지는 것과 동일한 의미의 객체를 가지고 있지 않고, 러스트는 몇몇 다른 언어들이 제공하는 것 같은 표준 라이브러리에 미리 만들어진 목 객체 기능이 없습니다. 하지만, 우리는 목 객체와 동일한 목적을 제공할 구조체를 당연히 만들 수 있습니다.

다음은 우리가 테스트할 시나리오입니다: 우리는 최대값에 맞서 값을 추적하고 현재 값이 최대값에 얼마나 근접한지를 기반으로 메세지를 전송하는 라이브러리를 만들 것입니다. 이 라이브러리는 예를 들면 한 명의 유저에게 허용되고 있는 API 호출수의 허용량을 추적하는데 사용될 수 있습니다.

우리의 라이브러리는 오직 어떤 값이 최대값에 얼마나 근접한지를 추적하고 어떤 시간에 어떤 메세지를 보내야 할지 정하는 기능만을 제공할 것입니다. 우리의 라이브러리를 사용하는 어플리케이션이 메세지를 전송하는 것에 대한 메카니즘을 제공할 예정입니다: 그 어플리케이션은 메세지를 어플리케이션 내에 집어넣거나, 이메일을 보내거나, 문자 메세지를 보내거나, 혹은 기타 다른 것을 할 수 있습니다. 라이브러리는 그런 자세한 사항을 알 필요가 없습니다. 필요한 모든 것은 우리가 제공할 `Messenger`라는 이름의 트레잇을 구현하는 것입니다. Listing 15-20는 라이브러리 코드를 보여줍니다:

Filename: src/lib.rs

```

pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: 'a + Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where T: Messenger {
    pub fn new(messenger: &T, max: usize) -> LimitTracker<T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 0.75 && percentage_of_max < 0.9 {
            self.messenger.send("Warning: You've used up over 75% of your
quota!");
        } else if percentage_of_max >= 0.9 && percentage_of_max < 1.0 {
            self.messenger.send("Urgent warning: You've used up over 90% of
your quota!");
        } else if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        }
    }
}

```

Listing 15-20: 어떤 값이 최대값에 얼마나 근접하는지를 추적하고 특정 수준에 값이 있으면 경고해주는 라이브러리

이 코드에서 한가지 중요한 부분은 `Messenger` 트레이트의 `self`에 대한 불변 참조자와 메세지의 텍스트를 인자로 갖는 `send`라는 이름의 하나의 메소드를 갖고 있다는 것입니다. 이는 우리의 목 객체가 가질 필요가 있는 인터페이스입니다. 그 외에 중요한 부분은 우리가 `LimitTracker` 상의 `set_value` 메소드의 동작을 테스트하고 싶어한다는 점입니다. 우리는 `value` 파라미터에 대해 어떤 것을 넘길지 바꿀 수 있지만, `set_value`는 우리가 단언을 하기 위한 어떤 것도 반환하지 않습니다. 우리는 `Messenger` 트레이트를 구현한 무언가와 `max`에 대한 특정값과 함께 `LimitTracker`를 만든다면, `value`에 대해 다른 숫자들을 넘겼을 때 메신저가 적합한 메세지를 보낸다고 말하고 싶습니다.

우리는 `send` 를 호출했을 때 메일이나 텍스트 메세지를 보내는 대신 보냈다고 언급하는 메세지만 추적할 목 객체가 필요합니다. 우리는 목 객체의 새로운 인스턴스를 만들고, 이 목 객체를 사용하는 `LimitTracker` 를 만들고, `LimitTracker` 상의 `set_value` 메소드를 호출하고, 그 다음 목 객체는 우리가 기대했던 메세지를 가지고 있는지를 검사할 수 있습니다. Listing 15-21은 바로 이런 일을 하지만 빌림 검사기가 허용하지는 않을 목 객체 구현 시도를 보여주고 있습니다:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: vec![] }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

Listing 15-21: 빌림 검사기가 허용하지 않는 `MockMessenger` 구현 시도

이 테스트 코드는 보내질 메세지들을 추적하기 위한 `String` 값의 `Vec` 인 `sent_messages` 필드를 갖는 `MockMessenger` 구조체를 정의하고 있습니다. 우리는 또한 빈 메세지 리스트로 시작하는 새로운 `MockMessenger` 값을 생성하기 쉽게 하기 위해 연관 함수 `new` 를 정의하였습니다. 그런 다음에는 `MockMessenger` 를 `LimitTracker` 에 넘겨줄 수 있도록 `MockMessenger` 를 위한 `Messenger` 트레이잇을 구현하였습니다. `send` 메소드의 정의 부분에서는 파라미터로서 넘겨진 메세지를 가져와서

`MockMessenger` 내의 `sent_messages` 리스트에 저장합니다.

테스트 내에서는 `max` 값의 75 퍼센트 이상의 무언가가 `value`로 설정되었을 때 `LimitTracker`는 어떤 메세지를 듣는지를 테스트하고 있습니다. 먼저 우리는 새로운 `MockMessenger`를 만드는데, 이는 비어있는 메시지 리스트로 시작할 것입니다. 그 다음에는 새로운 `LimitTracker`를 만들고 여기에 새로운 `MockMessenger`의 참조자와 `max` 값 100을 파라미터로 넘깁니다. 우리는 `LimitTracker` 상의 `set_value` 메소드를 80 값으로 호출하였습니다. 그 다음 우리는 `MockMessenger`가 추적하고 있는 메세지 리스트가 이제 한 개의 메세지를 가지고 있는지를 검사합니다.

하지만, 아래에서 보는 것과 같이 이 테스트에 한가지 문제점이 있습니다:

```
error[E0596]: cannot borrow immutable field `self.sent_messages` as mutable
--> src/lib.rs:52:13
   |
51 |         fn send(&self, message: &str) {
   |             ^----- use `&mut self` here to make mutable
52 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^ cannot mutably borrow immutable field
```

우리는 메세지를 추적하기 위해 `MockMessenger`를 수정할 수 없는데 그 이유는 `send` 메소드가 `self`의 불변 참조자를 파라미터로 갖기 때문입니다. 우리는 또한 여러 메세지로부터 `&mut self`를 대신 사용하라는 제안도 얻을 수 없는데, 그렇게 되면 `send`의 시그니처가 `Messenger` 트레이트의 정의에 있는 시그니처와 일치하지 않을 것이지 때문입니다 (마음 편하게 한번 시도해보고 어떤 에러가 나오는지 보세요).

이는 내부 가변성이 도움을 줄 수 있는 상황입니다! 우리는 `sent_messages`를 `RefCell<T>` 내에 저장할 것이고, 그러면 `send` 메소드는 우리가 보게 되는 메세지를 저장하기 위해 `sent_message`를 수정할 수 있을 것입니다. Listing 15-22는 이것이 어떤 형태인지를 보여줍니다:

Filename: src/lib.rs

```

#[cfg(test)]
mod tests {
    use super::*;

    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: RefCell::new(vec![]) }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}

```

Listing 15-22: `RefCell<T>`를 사용하여 바깥쪽에서는 불변으로 간주되는 동안 내부의 값을 변경하기

`sent_messages` 필드는 이제 `Vec<String>` 대신 `RefCell<Vec<String>>` 타입입니다. `new` 함수 내에서, 우리는 빈 벡터를 감싼 새로운 `RefCell<Vec<String>>` 인스턴스를 생성합니다.

`send` 메소드의 구현부에 대하여, 첫번째 파라미터는 여전히 `self`의 불변 빌림 형태인데, 이는 트레이트의 정의와 일치합니다. 우리는 `self.sent_messages` 내의 `RefCell<Vec<String>>` 상에 있는 `borrow_mut`를 호출하여 `RefCell<Vec<String>>` 내의 값에 대한 가변 참조자를 얻는데, 이는 벡터입니다. 그런 다음에는 그 벡터에 대한 가변 참조자 상의 `push`를 호출하여 테스트하는 동안 보내진 메세지를 추적할 수 있습니다.

마지막으로 우리가 변경한 부분은 단언 부분 내에 있습니다: 내부 벡터 내에 몇개의 아이템이 있는지 보기 위해서, 우리는 `RefCell<Vec<String>>` 상의 `borrow`를 호출하여 벡터에 대한 불변 참조자를 얻습니다.

이제 여러분이 `RefCell<T>`를 어떻게 사용하는지 보았으니, 이것이 어떤 식으로 동작하는지 파고 들어 봅시다!

RefCell<T>는 런타임에 빌림을 추적합니다

불변 및 가변 참조자를 만들 때, 우리는 각각 `&` 및 `&mut` 문법을 사용합니다. `RefCell<T>`을 이용할 때는 `borrow` 와 `borrow_mut` 메소드를 사용하는데, 이들은 `RefCell<T>`가 소유한 안전한 API 중 일부입니다. `borrow` 메소드는 스마트 포인터 타입인 `Ref<T>`를 반환하고, `borrow_mut`는 스마트 포인터 타입 `RefMut<T>`을 반환합니다. 두 타입 모두 `Deref`를 구현하였으므로 우리는 이들을 보통의 참조자처럼 다룰 수 있습니다.

`RefCell<T>`는 현재 활성화된 `Ref<T>` 와 `RefMut<T>` 스마트 포인터들이 몇개나 있는지 추적합니다. 우리가 `borrow`를 호출할 때마다, `RefCell<T>`는 불변 참조자가 활성화된 갯수를 증가시킵니다. `Ref<T>` 값이 스코프 밖으로 벗어날 때, 불변 빌림의 갯수는 하나 감소합니다. 컴파일 타임에서의 빌림 규칙과 똑같이, `RefCell<T>`는 우리가 어떤 시점에서든 여러 개의 불변 빌림 혹은 하나의 가변 빌림을 가질 수 있도록 합니다.

만일 이 규칙들을 위반한다면, `RefCell<T>`의 구현체는 우리가 참조자들을 가지고 했을 때처럼 컴파일 에러를 내기보다는 런타임에 `panic!`을 일으킬 것입니다. Listing 15-23은 Listing 15-22의 `send` 구현의 수정을 보여줍니다. 우리는 `RefCell<T>`가 런타임에 두 개의 활성화된 가변 빌림을 같은 스코프에 만드는 일을 하는 것을 막아주는 것을 보여주기 위해서 의도적으로 그런 시도를 하는 중입니다:

Filename: src/lib.rs

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

Listing 15-23: `RefCell<T>`이 패닉을 일으킬 것을 보기 위한 같은 스코프 내에 두 개의 가변 참조자 만들기

우리는 `borrow_mut`로부터 반환된 `RefMut<T>` 스마트 포인터를 위한 `one_borrow` 변수를 만들었습니다. 그런 다음 또 다른 가변 빌림을 같은 방식으로 `two_borrow` 변수에 만들어 넣었습니다. 이는 같은 스코프에 두 개의 가변 참조자를 만드는데, 이는 허용되지 않습니다. 우리가 우리의 라이브러리를 위한 테스트를 실행할 때, Listing 15-23의 코드는 어떠한 에러 없이 컴파일될 것이지만, 테스트는 실패할 것입니다:

```
---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'tests::it_sends_an_over_75_percent_warning_message' panicked at
'already borrowed: BorrowMutError', src/libcore/result.rs:906:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

코드가 `already borrowed: BorrowMutError`라는 메세지와 함께 패닉을 일으켰음을 주목하세요. 이것이 바로 `RefCell<T>` 가 런타임에 빌림 규칙의 위반을 다루는 방법입니다.

빌림 에러를 컴파일 타임보다 런타임에 잡는다는 것은 개발 과정 이후에 우리 코드의 실수를 발견할 것이라 의미이고, 심지어는 우리 코드가 프로덕션으로 배포될 때 까지도 발견되지 않을 가능성도 있습니다. 또한, 우리 코드는 컴파일 타임 대신 런타임에 빌림을 추적하는 결과로서 약간의 런타임 성능 페널티를 초래할 것입니다. 그러나, `RefCell<T>` 를 이용하는 것은 우리가 오직 불변 값만 허용하는 콘텍스트 내에서 그것이 본 메세지를 추적하기 위해서 스스로를 변경할 수 있는 목 객체를 작성하도록 해줍니다. 우리는 일반적인 참조자가 우리에게 제공하는 것보다 더 많은 기능을 얻기 위해서 트레이드 오프에도 불구하고 `RefCell<T>` 를 이용 할 수 있습니다.

`Rc<T>`와 `RefCell<T>`를 조합하여 가변 데이터의 복수 소유자 만들기

`RefCell<T>` 를 사용하는 일반적인 방법은 `Rc<T>` 와 함께 조합하는 것입니다. `Rc<T>` 이 어떤 데이터에 대해 복수의 소유자를 허용하지만, 그 데이터에 대한 불변 접근만 제공하는 것을 상기하세요. 만일 우리가 `RefCell<T>` 을 들고 있는 `Rc<T>` 를 갖는다면, 우리가 변경 가능하면서 복수의 소유자를 갖는 값을 가질 수 있습니다!

예를 들면, Listing 15-18에서 우리가 어떤 리스트의 소유권을 공유하는 여러 개의 리스트를 가질 수 있도록 하기 위해 `Rc<T>` 를 사용했던 `cons` 리스트 예제를 상기해보세요. `Rc<T>` 가 오직 불변의 값만을 가질 수 있기 때문에, 우리가 이들을 일단 만들면 리스트 안의 값들을 변경하는 것은 불가능했습니다. 이 리스트 안의 값을 변경하는 능력을 얻기 위해서 `RefCell<T>` 를 추가해 봅시다. Listing 15-24는 `Cons` 정의 내에 `RefCell<T>` 를 사용함으로써 우리가 모든 리스트 내에 저장된 값을 변경할 수 있음을 보여줍니다:

Filename: src/main.rs

```

#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}

```

Listing 15-24: `Rc<RefCell<i32>>`을 사용하여 변경 가능한 `List` 생성하기

우리는 `Rc<RefCell<i32>>`의 인스턴스인 값을 생성하고 `value`라는 이름의 변수 안에 이를 저장하여 나중에 이를 직접 접근할 수 있게 했습니다. 그런 다음 우리는 `value`를 가지고 있는 `Cons` variant와 함께 `a`에다 `List`를 생성하였습니다. `value`에서 `a`로 소유권이 이전되거나 `value`로부터 빌린 `a` 보다는 `a`와 `value` 둘다 내부의 `5` 값에 대한 소유권을 얻기 위해서는 `value`를 클론할 필요가 있습니다.

리스트 `a`는 `Rc<T>`로 감싸져서 우리가 `b`와 `c` 리스트를 만들때, 이 리스트들은 둘다 `a`를 참조할 수 있는데, 이는 Listing 15-18에서 해본 것입니다.

`a`, `b`, 그리고 `c` 리스트를 생성한 이후, 우리는 `value` 내의 값에 10을 더했습니다. `value` 상의 `borrow_mut`를 호출함으로써 수행되었는데, 이는 내부의 `RefCell<T>` 값을 가리키는 `Rc<T>`를 역참조하기 위해서 우리가 5장에서 논의했던 자동 역참조 기능을 사용한 것입니다 ("`->` 연산자는 어디로 갔나요?" 절을 보세요). `borrow_mut` 메소드는 `RefMut<T>` 스마트 포인터를 반환하고, 우리는 여기에 역참조 연산자를 사용한 다음 내부 값을 변경합니다.

`a`, `b`, 그리고 `c`를 출력할때, 우리는 이 리스트들이 모두 5가 아니라 변경된 값 15를 가지고 있는 것을 볼 수 있습니다:

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

이 기술은 매우 깔끔합니다! `RefCell<T>`을 이용함으로써, 우리는 표면상으로는 불변인 `List`를 갖고 있습니다. 하지만 우리는 내부 가변성 접근을 제공하여 우리가 원할때 데이터를 변경시킬 수 있는 `RefCell<T>` 내의 메소드를 사용할 수 있습니다. 빌림 규칙의 런타임 검사는 데이터 레이스로부터 우리를 지켜주고, 우리 데이터 구조의 이러한 유연성을 위해서 약간의 속도를 맞거래하는 것이 때때로 가치있습니다.

표준 라이브러리는 내부 가변성을 제공하는 다른 타입을 가지고 있는데, 이를 테면 `Cell<T>`는 내부 값의 참조자를 주는 대신 값이 복사되어 `Cell<T>` 밖으로 나오는 점만 제외하면 비슷합니다. 또한 `Mutex<T>`가 있는데, 이는 스레드들을 건너가며 사용해도 안전한 내부 가변성을 제공합니다; 이것의 사용법은 16장에서 다룰 것입니다. 이 타입들의 차이점에 대해 더 자세히 알고 싶다면 표준 라이브러리 문서를 참조하세요.

순환 참조는 메모리 릭을 발생시킬 수 있습니다

러스트의 메모리 안정성 보장은 (메모리 릭 (*memory leak*)이라고도 알려져 있는) 뜻하지 않게 해제되지 않는 메모리를 생성하기 힘들게 하지만, 그게 불가능한 것은 아닙니다. 메모리 릭을 완전히 방지하는 것은 컴파일 타임에 데이터 레이스를 허용하지 않는 것과 마찬가지로 러스트가 보장하는 것들 중 하나가 아닌데, 이는 메모리 립도 러스트에서는 메모리 안정성에 포함됨을 의미합니다. 러스트가 `Rc<T>` 및 `RefCell<T>`를 사용하여 메모리 립을 허용하는 것을 우리는 알 수 있습니다: 즉 아이템들이 서로를 순환 참조하는 참조자를 만드는 것이 가능합니다. 이는 메모리 립을 발생시키는데, 그 이유는 순환 고리 안의 각 아이템들의 참조 카운트는 결코 0이 되지 않을 것이고, 그러므로 값들은 버려지지 않을 것이기 때문입니다.

순환 참조 만들기

Listing 15-25의 `List` 열거형과 `tail` 메소드 정의를 가지고서 어떻게 순환 참조가 생길 수 있고, 이를 어떻게 방지하는지 알아봅시다:

Filename: src/main.rs

```
use std::rc::Rc;
use std::cell::RefCell;
use List::{Cons, Nil};

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match *self {
            Cons(_, ref item) => Some(item),
            Nil => None,
        }
    }
}
```

Listing 15-25: `RefCell<T>`를 가지고 있어서 `Cons` variant가 참조하는 것을 변경할 수 있는 `cons` 리스트 정의

우리는 Listing 15-5의 `List` 정의의 또 다른 변형을 이용하고 있습니다. 이제 `Cons` variant 내의 두번째 요소는 `RefCell<Rc<List>>`인데, 이는 Listing 15-24에서 우리가 했던 것처럼 `i32` 값을 변경하는 능력을 갖는 대신, `Cons` variant가 가리키고 있는 `List` 값을 변경하길 원한다는 의미입니다. 또한 `Cons`

variant를 갖고 있다면 두번째 아이템에 접근하기 편하도록 `tail` 메소드를 추가하고 있습니다.

Listing 15-26에서 우리는 Listing 15-25의 정의를 사용하는 `main` 함수를 추가하고 있습니다. 이 코드는 `a`에 리스트를 만들고 `b`에는 `a`의 리스트를 가리키고 있는 리스트를 만들어 넣었습니다. 그 다음 `a`의 리스트가 `b`를 가리키도록 수정하는데, 이것이 순환 참조를 생성합니다. 이 과정 내의 다양한 지점에서 참조 카운트가 얼마인지를 보기 위해 곳곳에 `println!` 구문들이 있습니다.

Filename: src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

Listing 15-26: 두 개의 `List` 값이 서로를 가리키는 순환 참조 생성하기

우리는 초기값 리스트 `5, Nil`를 가진 `List` 값을 갖는 `Rc<List>` 인스턴스를 만들어 `a` 변수에 넣었습니다. 그런 다음 값 10과 `a`의 리스트를 가리키는 또다른 `List` 값을 갖는 `Rc<List>` 인스턴스를 만들어서 `b` 변수에 넣었습니다.

우리는 `a`를 수정하여 이것이 `Nil` 대신 `b`를 가리키도록 하였습니다. `a` 내의 `RefCell<Rc<List>>`에 대한 참조자를 얻어오기 위해 `tail` 메소드를 사용했는데, 이 참조자는 `link` 변수에 집어넣습니다. 그런 다음 `RefCell<Rc<List>>`의 `borrow_mut` 메소드를 사용하여 `Nil` 값을 가지고 있는 `Rc<List>` 내부의 값을 `b`의 `Rc<List>`로 바꾸었습니다.

지금 잠깐동안 마지막 `println!` 문이 들어가지 않도록 주석처리하고 이 코드를 실행시킬 때, 아래와 같은

출력을 얻을 것입니다:

```
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

a의 리스트가 **b**를 가리키도록 변경한 이후 **a**와 **b**의 **Rc<List>** 인스턴스의 참조 카운트는 둘 다 2입니다. **main**의 끝에서, 러스트는 **b**를 먼저 버리는 시도를 할 것인데, 이는 **a**와 **b**의 각각의 **Rc<List>** 인스턴스 내의 카운트를 1로 줄일 것입니다.

하지만 **a**가 여전히 **b** 내에 있는 **Rc<List>**를 참조하는 상태기 때문에, 이 **Rc<List>**는 0이 아니라 1의 카운트를 갖게 되고, 따라서 **Rc<List>**가 힙에 가지고 있는 메모리는 버려지지 않을 것입니다. 그 메모리는 참조 카운트 1을 가진 채로 영원히 그 자리에 그냥 있을 것입니다. 이러한 순환 참조를 시각화하기 위해 Figure 15-4의 다이어그램을 만들었습니다.

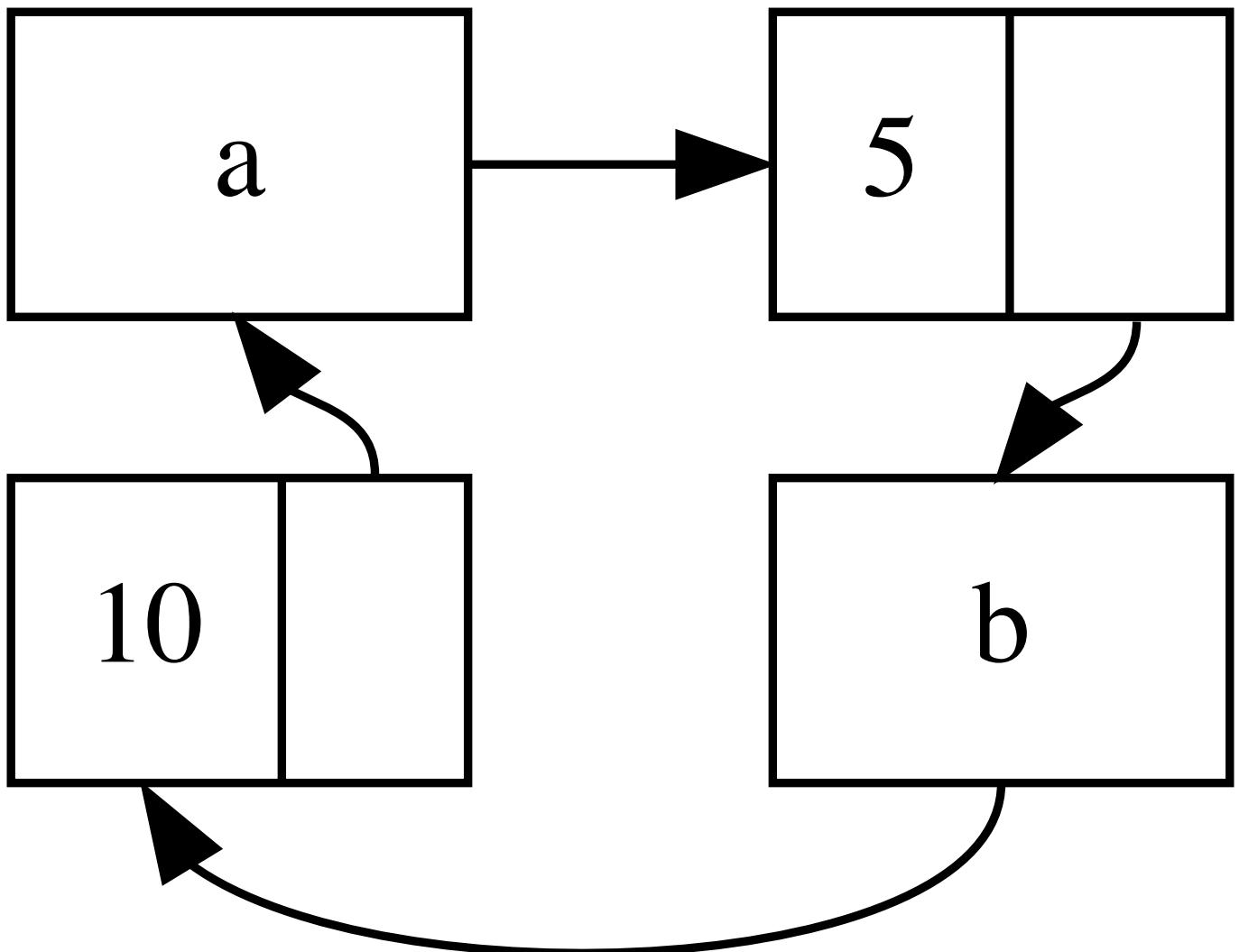


Figure 15-4: 리스트 `a`와 `b`가 서로를 가리키고 있는 순환 참조

만일 여러분이 마지막 `println!`의 주석을 해제하고 프로그램을 실행해보면, 러스트는 `a`를 가리키고 있는 `b`를 가리키고 있는 `a`를 가리키고 있는... 과 같은 식으로 스택 오버플로우가 날 때까지 이 순환을 출력하려 할 것입니다.

이 경우, 우리가 순환 참조를 만든 직후, 프로그램은 종료됩니다. 위의 순환의 결과는 그렇게까지 심각하지는 않습니다. 하지만, 만일 좀더 복잡한 프로그램이 많은 메모리를 순환 형태로 할당했고 오랫동안 이를 유지했더라면, 프로그램은 필요한 것보다 더 많은 메모리를 사용하게 되고, 사용 가능한 메모리를 둉나게 하여 시스템을 멈추게 했을지도 모릅니다.

순환 참조를 만드는 것은 쉽게 이루어지지는 않지만, 불가능한 것도 아닙니다. 만일 여러분이 `Rc<T>` 값을 가지고 있는 `RefCell<T>` 혹은 내부 가변성 및 참조 카운팅 기능이 있는 타입들로 유사한 조합을 사용한다면, 여러분은 순환을 만들지 않음을 보장해야 합니다; 이 순환들을 찾아내는 것을 러스트에 의지할 수는 없습니다. 순환 참조를 만드는 것은 여러분이 자동화된 테스트, 코드 리뷰, 그 외 소프트웨어 개발 연습 등을 이용하여 최소화해야 할 프로그램 내의 논리적 버그입니다.

순환 참조를 피하는 또 다른 해결책은 여러분의 데이터 구조를 재구성하여 어떤 참조자는 소유권을 갖고 어떤 참조자는 그렇지 않도록 하는 것입니다. 결과적으로 여러분은 몇 개의 소유권 관계와 몇 개의 소유권 없는 관계로 이루어진 순환을 가질 수 있으며, 소유권 관계들만이 값을 버릴지 말지에 관해 영향을 주게 됩니다.

Listing 15-25에서 우리는 `Cons` variant가 언제나 리스트를 소유하기를 원하므로, 데이터 구조를 재구성하는 것은 불가능합니다. 언제 소유권 없는 관계가 순환 참조를 방지하는 적절한 방법이 되는 때인지를 알기 위해서 부모 노드와 자식 노드로 구성된 그래프를 이용하는 예제를 살펴봅시다.

참조 순환 방지하기: `Rc<T>`를 `Weak<T>`로 바꾸기

이제까지 우리는 `Rc::clone`을 호출하는 것이 `Rc<T>` 인스턴스의 `strong_count`를 증가시키고, `Rc<T>` 인스턴스는 이것의 `strong_count`가 0이 된 경우에만 제거되는 것을 보았습니다. 여러분은 또한 `Rc::downgrade`를 호출하고 여기에 `Rc<T>`에 대한 참조자를 넘겨서 `Rc<T>` 인스턴스 내의 값을 가리키는 약한 참조 (*weak reference*)를 만들 수 있습니다. 여러분이 `Rc::downgrade`를 호출하면, 여러분은 `Weak<T>` 타입의 스마트 포인터를 얻게 됩니다. `Rc<T>` 인스턴스의 `strong_count`를 1 증가시키는 대신, `Rc::downgrade`는 `weak_count`를 1 증가시킵니다. `Rc<T>` 타입은 몇 개의 `Weak<T>` 참조가 있는지 추적하기 위해서 `strong_count`와 유사한 방식으로 `weak_count`를 사용합니다. 차이점은 `Rc<T>` 인스턴스가 제거되기 위해서 `weak_count`가 0일 필요는 없다는 것입니다.

강한 참조는 여러분이 `Rc<T>` 인스턴스의 소유권을 공유할 수 있는 방법입니다. 약한 참조는 소유권 관계를 표현하지 않습니다. 이것은 순환 참조를 야기하지 않는데 그 이유는 몇몇의 약한 참조를 포함하는 순환이라도 강한 참조의 카운트가 0이 되고 나면 깨지게 될 것이기 때문입니다.

`Weak<T>`가 참조하고 있는 값이 이미 버려졌을지도 모르기 때문에, `Weak<T>`가 가리키고 있는 값을 가지고 어떤 일을 하기 위해서는 그 값이 여전히 존재하는지를 반드시 확인해야 합니다. 이를 위해 `Weak<T>`의 `upgrade` 메소드를 호출하는데, 이 메소드는 `Option<Rc<T>>`를 반환할 것입니다. 만일 `Rc<T>` 값이 아직 버려지지 않았다면 여러분은 `Some` 결과를 얻게 될 것이고 `Rc<T>` 값이 버려졌다면 `None` 결과값을 얻게 될 것입니다. `upgrade`가 `Option<T>`를 반환하기 때문에, 러스트는 `Some`의 경우와 `None`의 경우가 반드시 처리되도록 할 것이고, 따라서 유효하지 않은 포인터는 없을 것입니다.

예제로서 어떤 아이템이 오직 다음 아이템에 대해서만 알고 있는 리스트를 이용하는 것보다는 자식 아이템 그리고 부모 아이템에 대해 모두 알고 있는 아이템을 갖는 트리를 만들어 보겠습니다.

트리 데이터 구조 만들기: 자식 노드를 가진 `Node`

자신의 자식 노드에 대해 알고 있는 노드를 갖는 트리를 만드는 것으로 시작해 보겠습니다. 우리는 `i32` 같은 물론 자식 `Node`들의 참조자들 또한 가지고 있는 `Node`라는 이름의 구조체를 만들 것입니다:

Filename: src/main.rs

```
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>,
}
```

우리는 `Node`가 자신의 자식들을 소유하기를 원하고, 이 소유권을 공유하여 트리의 각 `Node`에 직접 접근할 수 있도록 하기를 원합니다. 이를 하기 위해서 `Vec<T>` 아이템이 `Rc<Node>` 타입의 값이 되도록 정의하였습니다. 또한 우리는 어떤 노드가 다른 노드의 자식이 되도록 수정하기를 원하므로, `Vec<Rc<Node>>` 를 `RefCell<T>`로 감싼 `children`을 갖도록 하였습니다.

그 다음, Listing 15-27에서 보시는 것처럼 이 구조체 정의를 이용하여 3의 값과 자식 노드가 없는 `leaf`라는 이름의 `Node` 인스턴스, 그리고 5의 값과 `leaf`를 자식으로 갖는 `branch`라는 이름의 인스턴스를 만들도록 하겠습니다:

Filename: src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

Listing 15-27: 자식이 없는 `leaf` 노드와 이 `leaf`를 자식 중 하나로 갖는 `branch` 노드 만들기

`leaf` 내의 `Rc<Node>`를 클론하여 이를 `branch` 내에 저장했는데, 이는 `leaf` 내의 `Node`가 이제 두 소유권자를 가지게 되었다는 의미입니다. 우리는 `branch.children`를 통하여 `branch`에서부터 `leaf` 까지 접근할 수 있게 되었지만, `leaf`에서부터 `branch`로 접근할 방법은 없습니다. 그 이유는 `leaf`가 `branch`에 대한 참조자를 가지고 있지 않아서 이들간의 연관성을 알지 못하기 때문입니다. 우리는 `leaf`로 하여금 `branch`가 그의 부모임을 알도록 하기를 원합니다. 이걸 다음에 해보겠습니다.

자식으로부터 부모로 가는 참조자 추가하기

자식 노드가 그의 부모를 알도록 만들기 위하여, `parent` 필드를 우리의 `Node` 구조체 정의에 추가할 필요

가 있습니다. 문제는 `parent`의 타입이 어떤 것이 되어야 하는지를 결정하는 중에 발생합니다. 이것이 `Rc<T>`를 담을 수 없음을 우리는 알고 있는데, 그렇게 하게 되면 `branch`를 가리키고 있는 `leaf.parent` 와 `leaf`를 가리키고 있는 `branch.children`를 가지고 있는 순환 참조를 만들게 되며, 이것들의 `strong_count` 값을 결코 0이 안되도록 하는 일을 야기하기 때문입니다.

이 관계들을 다른 방식으로 생각해보면, 부모 노드는 그의 자식들을 소유해야 합니다: 만일 부모 노드가 버려지게 되면, 그의 자식 노드들도 또한 버려져야 합니다. 하지만, 자식은 그의 부모를 소유해서는 안됩니다: 만일 우리가 자식 노드를 버리면, 그 부모는 여전히 존재해야 합니다. 이것이 바로 약한 참조를 위한 경우에 해당됩니다!

따라서 `Rc<T>` 대신 `Weak<T>`를 이용하여, 특별히 `RefCell<Weak<Node>>`를 이용하여 `parent`의 타입을 만들겠습니다. 이제 우리의 `Node` 구조체 정의는 아래와 같이 생기게 되었습니다:

Filename: src/main.rs

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

노드는 그의 부모 노드를 참조할 수 있게 되겠지만 그 부모를 소유하지는 않습니다. Listing 15-28에서, 우리는 이 새로운 정의를 사용하도록 `main` 을 업데이트하여 `leaf` 노드가 그의 부모인 `branch`를 참조할 수 있는 방법을 갖도록 할 것입니다:

Filename: src/main.rs

```

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}

```

Listing 15-28: 부모 노드 `branch`의 약한 참조를 갖는 `leaf` 노드

`leaf` 노드를 만드는 것은 `parent` 필드를 제외하고는 Listing 15-27에서 `leaf` 노드를 만드는 방법과 비슷해 보입니다: `leaf`는 부모없이 시작되어서, 새로운 비어있는 `Weak<Node>` 참조자 인스턴스를 생성하였습니다.

이 시점에서, 우리가 `upgrade` 메소드를 사용하여 `leaf`의 부모에 대한 참조자를 얻는 시도를 했을 때, 우리는 `None` 값을 얻습니다. 첫번째 `println!` 구문에서는 아래와 같은 출력을 보게됩니다:

```
leaf parent = None
```

`branch` 노드를 생성할 때, 이 또한 `parent` 필드에 새로운 `Weak<Node>` 참조자를 가지도록 하는데, 이는 `branch`가 부모 노드를 가지지 않기 때문입니다. 우리는 여전히 `leaf`를 `branch`의 자식 중 하나로서 가지게 됩니다. 일단 `branch` 내의 `Node` 인스턴스를 가지게 되면, `leaf`에게 그의 부모에 대한 `Weak<Node>` 참조자를 가지도록 수정할 수 있습니다. 우리는 `leaf`의 `parent` 필드 내의 `RefCell<Weak<Node>>` 상의 `borrow_mut` 메소드를 사용하고, 그런 다음 `Rc::downgrade` 함수를 이용하여 `branch`의 `Rc<Node>`로부터 `branch`에 대한 `Weak<Node>` 참조자를 생성하였습니다.

`leaf`의 부모를 다시한번 출력할 때, 이번에는 `branch`를 가지고 있는 `Some` variant를 얻게될 것입니다: 이제 `leaf`는 그의 부모에 접근할 수 있습니다! `leaf`를 출력할 때, 우리는 또한 Listing 15-26에서 발생했던 것과 같이 궁극적으로 스택 오버플로우로 끝나버리는 순환을 피하게 되었습니다: `Weak<Node>` 참조자는 `(Weak)`로 출력됩니다:

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },  
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak)  
},  
children: RefCell { value: [] } }] } })
```

무한 출력이 없다는 것은 이 코드가 순환 참조를 생성하지 않는 것을 나타냅니다. 이것은 또한 `Rc::strong_count` 와 `Rc::weak_count` 를 호출함으로써 얻은 값을 살펴보는 것으로도 알 수 있습니다.

strong_count와 **weak_count**의 변화를 시각화하기

새로운 내부 스코프를 만들고 `branch`의 생성을 이 스코프로 옮기는 것으로 `Rc<Node>` 인스턴스의 `strong_count` 와 `weak_count` 값이 어떻게 변하는지 살펴보기로 합시다. 그렇게 함으로써, 우리는 `branch` 가 만들어질 때와 그 다음 스코프 밖으로 벗어났을 때 어떤 일이 생기는지 알 수 있습니다. 수정본은 Listing 15-29와 같습니다:

Filename: src/main.rs

```

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );
    }

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}

println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
println!(
    "leaf strong = {}, weak = {}",
    Rc::strong_count(&leaf),
    Rc::weak_count(&leaf),
);
}

```

Listing 15-29: `branch`를 내부 스코프에서 만들고 강한 참조 및 약한 참조 카운트를 시험하기

`leaf`가 생성된 다음, 이것의 `Rc<Node>`는 강한 참조 카운트 1개와 약한 참조 카운트 0개를 갖습니다. 내부 스코프에서 `branch`를 만들고 `leaf`와 연관짓게 되는데, 이때 우리가 카운트를 출력하면 `branch`의 `Rc<Node>`는 강한 참조 카운트 1개와 (`Weak<Node>`를 가지고 `branch`를 가리키는 `leaf.parent`에 대

한) 약한 참조 카운트 1개를 갖고 있을 것입니다. 이제 `branch` 가 `leaf` 의 `Rc<Node>` 클론을 `branch.children`에 저장해 두었으므로, `leaf`의 카운트를 출력해보면 강한 참조 카운트는 2개가 되지만, 약한 참조 카운트는 여전히 0개일 것입니다.

내부 스코프가 끝나게 되면, `branch` 는 스코프 밖으로 벗어나게 되며 `Rc<Node>` 의 강한 참조 카운트는 0 으로 줄어들게 되므로, 이것의 `Node` 는 버려지게 됩니다. `leaf.parent`로부터 발생된 1개의 약한 참조 카운트는 `Node` 가 버려질지 말지에 대한 어떠한 영향도 주지 않으므로, 아무런 메모리 릭도 발생하지 않았습니다!

만일 우리가 이 스코프의 끝 이후에 `leaf` 의 부모에 접근하기를 시도한다면, 우리는 다시 `None` 을 얻게 될 것입니다. 프로그램의 끝 부분에서, `leaf` 의 `Rc<Node>` 는 강한 참조 카운트 1개와 약한 참조 카운트 0개 를 갖고 있는데, 그 이유는 `leaf` 변수가 이제 다시 `Rc<Node>` 에 대한 유일한 참조자이기 때문입니다.

참조 카운트들과 버리는 값들을 관리하는 모든 로직은 `Rc<T>` 와 `Weak<T>`, 그리고 이들의 `Drop` 트레이트에 대한 구현부에 만들어져 있습니다. 자식으로부터 부모로의 관계가 `Node` 의 정의 내에서 `Weak<T>` 참조자로 되어야 함을 특정함으로서, 여러분은 순환 참조와 메모리 릭을 만들지 않고도 자식 노드를 가리키는 부모 노드 혹은 그 반대의 것을 가지게 될 수 있습니다.

정리

이번 장에서는 러스트가 일반적인 참조자를 가지고 기본적으로 보장하는 것들과는 다른 보장 및 트레이드 오프를 만들어내기 위해 스마트 포인터를 사용하는 방법을 다루었습니다. `Box<T>` 타입은 알려진 크기를 갖고 있고 힙에 할당된 데이터를 가리킵니다. `Rc<T>` 타입은 힙에 있는 데이터에 대한 참조자의 개수를 추적하여 그 데이터가 여러 개의 소유자들을 가질 수 있도록 합니다. 내부 가변성을 갖춘 `RefCell<T>` 타입은 불변 타입을 원하지만 그 타입의 내부 값을 변경하기를 원할 때 사용할 수 있는 타입을 제공합니다; 이는 또한 컴파일 타임 대신 런타임에 빌림 규칙을 따르도록 강제합니다.

또한 `Deref` 및 `Drop` 트레이트를 다루었는데, 이는 스마트 포인터의 수많은 기능을 활성화해줍니다. 우리는 메모리 릭을 발생시킬 수 있는 순환 참조에 대한 것과 `Weak<T>` 을 이용하여 이들을 방지하는 방법도 탐구하였습니다.

만일 이번 장이 여러분의 흥미를 언짢게 하고 여러분이 직접 여러분만의 스마트 포인터를 구현하기를 원한다면, “[러스토노미콘](#)”에서 더 유용한 정보를 확인하세요.

다음으로 우리는 러스트의 동시성에 대해 이야기해볼 것입니다. 여러분은 심지어 몇 개의 새로운 스마트 포인터에 대해서도 배우게 될 것입니다.

겁없는 동시성

안전하고 효율적으로 동시성 프로그래밍을 다루는 것은 러스트의 또 다른 주요 목표들 중 하나입니다. 동시성 프로그래밍 (*concurrent programming*), 즉 프로그램의 서로 다른 부분이 독립적으로 실행되는 것과, 병렬 프로그래밍 (*parallel programming*), 즉 프로그램의 서로 다른 부분이 동시에 실행되는 것은 더 많은 컴퓨터들이 여러 개의 프로세서로 이점을 얻음에 따라 그 중요성이 증가하고 있습니다. 역사적으로, 이러한 맥락에서 프로그래밍하는 것은 어렵고 에러를 내기 쉬웠습니다: 러스트는 이를 바꾸기를 바라고 있습니다.

초기에 러스트 팀은 메모리 안전을 보장하는 것과 동시성 문제를 방지하는 것은 다른 방법으로 해결되야 하는 별개의 도전 과제라고 생각했습니다. 시간이 흘러 러스트 팀은 소유권과 타입 시스템이 메모리 안전성 및/동시성 문제를 관리하는 것을 돋기 위한 강력한 도구들의 집합이라는 사실을 발견했습니다! 소유권과 타입 검사를 지렛대삼아서 많은 동시성 에러들이 러스트 내에서 런타임 에러가 아닌 컴파일 타임 에러가 되었습니다. 그러므로, 런타임 동시성 버그가 발생하는 정확한 환경을 재현하는 시도를 하는데 여러분이 수많은 시간을 소비하도록 만들지 않고, 부정확한 코드는 컴파일 되기를 거부하고 문제점을 설명하는 에러를 보여줄 것입니다. 결과적으로 여러분은 잠재적으로 프로덕션에 배포된 이후가 아니라 작업을 하는 동안에 여러분의 코드를 고칠 수 있습니다. 우리는 러스트의 이러한 측면을 겁없는 동시성 (*fearless concurrency*)이라고 별명지어 주었습니다. 겁없는 동시성은 여러분이 감지하기 힘든 버그 없고 새로운 버그 생성 없이 리팩토링하기 쉬운 코드를 작성하도록 해줍니다.

노트: 단순함을 목적으로 우리는 많은 수의 문제들을 더 정교하게 동시성 및/또는 병렬성이라고 말하기보다는 그냥 동시성에 대한 문제로서 참고할 것입니다. 만일 이 책이 동시성 및/또는 병렬성에 대한 것이었다면, 우리는 더 정확하게 말했을 것입니다. 이번 장에서는 우리가 동시성이라고 말할 때마다 마음속으로 동시성 및/또는 병렬성을 대입해 주세요..

많은 언어들은 동시성 문제를 다루기 위해 그들이 제공하는 해결책에 대해 독단적입니다. 예를 들어, Erlang은 메세지-패싱 (message-passing) 동시성을 위한 우아한 기능을 가지고 있지만 스레드 간에 상태를 공유하기 위한 이해하기 힘든 방법만을 가지고 있습니다. 가능한 해결책 중 일부만을 제공하는 것은 고수준의 언어를 위한 타당한 전략인데, 이는 고수준의 언어가 추상화를 얻기 위해 몇몇의 제어권을 포기함으로서 얻는 이득을 약속하기 때문입니다. 하지만 저수준의 언어는 어떠한 주어진 상황 내에서 최고의 성능을 갖는 해결책을 제공하도록 기대받고 있고 하드웨어에 대하여 더 적은 추상화를 갖습니다. 그러므로, 러스트는 여러분의 상황과 요구사항에 적합한 방법이 무엇이든간에 문제를 모델링하기 위한 다양한 도구들을 제시합니다.

이번 장에서 다루게 될 주제들입니다:

- 여러 조각의 코드를 동시에 실행시키기 위해 스레드를 생성하는 방법
- 체널들이 스레드 간에 메세지를 보내는 메세지-패싱 동시성
- 여러 스레드가 어떤 동일한 데이터를 접근할 수 있는 상태-공유 (*shared-state*) 동시성
- 표준 라이브러리가 제공하는 타입 뿐만 아니라 러스트의 동시성 보장을 사용자 정의 타입으로 확장하

는 **Sync** 와 **Send** 트레이트

스레드를 이용하여 코드를 동시에 실행하기

대부분의 요즘 운영 체제에서, 실행되는 프로그램의 코드는 프로세스 내에서 실행되고, 운영 체제는 한번에 여러 개의 프로세스들을 관리합니다. 여러분의 프로그램 내에서도 동시에 실행되는 독립적인 부분들을 가질 수 있습니다. 이러한 독립적인 부분들을 실행하는 기능을 스레드라고 부릅니다.

여러분의 프로그램 내에 계산 부분을 여러 개의 스레드로 쪼개는 것은 프로그램이 동시에 여러 개의 일을 할 수 있기 때문에 성능을 향상시킬 수 있지만, 프로그램을 복잡하게 만들기도 합니다. 스레드가 동시에 실행될 수 있기 때문에, 다른 스레드 상에서 실행될 여러분의 코드 조각들의 실행 순서에 대한 내재적인 보장이 없습니다. 이는 다음과 같은 문제들을 야기할 수 있습니다:

- 여러 스레드들이 일관성 없는 순서로 데이터 혹은 리소스에 접근하게 되는, 경쟁 조건 (race condition)
- 두 스레드가 서로 상대방 스레드가 가지고 있는 리소스의 사용을 끝내길 기다려서 양쪽 스레드 모두 계속 실행되는 것을 막아버리는, 데드록 (deadlock)
- 특정한 상황에서만 발생되어 재현하기와 안정적으로 수정하기가 힘든 버그들

러스트는 스레드 사용의 부정적인 효과를 줄이려는 시도를 하지만, 다중 스레드 컨텍스트 내에서의 프로그래밍은 여전히 신중하게 생각해야 하고 단일 스레드 내에서 실행되는 프로그램의 것과는 다른 코드 구조가 필요합니다.

프로그래밍 언어들은 몇 가지 다른 방식으로 스레드를 구현합니다. 많은 운영 체제들이 새로운 스레드를 만들기 위한 API를 제공합니다. 언어가 운영 체제의 API를 호출하여 스레드를 만드는 이러한 구조는 때때로 1:1이라 불리는데, 이는 하나의 운영 체제 스레드가 하나의 언어 스레드에 대응된다는 의미입니다.

많은 프로그래밍 언어들은 그들만의 특별한 스레드 구현을 제공합니다. 프로그래밍 언어가 제공하는 스레드는 그린 (green) 스레드라고 알려져 있으며, 이러한 그린 스레드를 사용하는 언어들은 다른 숫자의 운영 체제 스레드로 구성된 컨텍스트 내에서 그린 스레드들을 실행할 것입니다. 이러한 이유로 인하여 그린 스레드 구조는 $M:N$ 이라고 불립니다: M 개의 그린 스레드가 N 개의 시스템 스레드에 대응되는데, 여기서 M 과 N 은 굳이 동일한 숫자가 아니어도 됩니다.

각각의 구조는 고유한 장점과 트레이드 오프를 가지고 있으며, 러스트에게 있어 가장 중요한 트레이드 오프는 런타임 지원입니다. 런타임은 혼동하기 쉬운 용어이고 맥락에 따라 다른 의미를 가질 수 있습니다.

이 글의 맥락에서 런타임이라 하는 것은 언어에 의해 모든 바이너리 내에 포함되는 코드를 의미합니다. 이 코드는 언어에 따라 크거나 작을 수 있지만, 모든 어셈블리 아닌 언어들은 어느 정도 크기의 런타임 코드를 가지게 될 것입니다. 이러한 이유로 인하여, 흔히 사람들이 “런타임이 없다”라고 말할 때는, 종종 “런타임이 작다”는 것을 의미하는 것입니다. 런타임이 작을 수록 더 적은 기능을 갖지만 더 작아진 바이너리로 인해 얻어지는 장점을 갖는데, 이는 더 큰 컨텍스트 내에서 다른 언어들과 조합하기 쉬워진다는 점입니다. 비록 많은 언어들이 더 많은 기능을 위하여 런타임 크기를 늘리는 거래를 수락하더라도, 러스트는 거의 런타임이 없을 필요가 있고 성능을 관리하기 위해 C를 호출하는 것에 대해 타협할 수 없습니다.

그린 스레드 M:N 구조는 스레드들을 관리하기 위해 더 큰 언어 런타임이 필요합니다. 그런 이유로 러스트 표준 라이브러리는 오직 1:1 스레드 구현만 제공합니다. 러스트가 이러한 저수준 언어이기 때문에, 여러분이 예를 들어 어떤 스레드를 언제 실행시킬지에 대한 더 많은 제어권과 컨텍스트 교환(context switching)의 더 저렴한 비용 같은 관점을 위해 오버헤드와 맞바꾸겠다면 M:N 스레드를 구현한 크레이트도 있습니다.

이제 러스트에서의 스레드를 정의했으니, 표준 라이브러리가 제공하는 스레드 관련 API를 어떻게 사용하는지를 탐구해봅시다.

spawn으로 새로운 스레드 생성하기

새로운 스레드를 생성하기 위해서는 `thread::spawn` 함수를 호출하고 여기에 우리가 새로운 스레드 내에서 실행하기를 원하는 코드가 담겨 있는 클로저를 넘깁니다 (클로저에 대해서는 13장에서 다뤘습니다). Listing 16-1의 예제는 메인 스레드에서 어떤 텍스트를 출력하고 새로운 스레드에서는 다른 텍스트를 출력합니다:

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Listing 16-1: 메인 스레드에서 무언가를 출력하는 동안 다른 것을 출력하는 새로운 스레드 생성하기

이 함수를 가지고, 새로운 스레드는 실행이 종료되었든 혹은 그렇지 않은 메인 스레드가 종료될 때 멈추게 될 것이라는 점을 주의하세요. 이 프로그램의 출력은 매번 약간씩 다를지도 모르겠으나, 아래와 비슷하게 보일 것입니다:

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

`thread::sleep`의 호출은 강제로 스레드가 잠깐 동안 실행을 멈추게 하는데, 다른 스레드가 실행되는 것을 허용합니다. 스레드들은 아마도 교대로 실행될 것이지만, 보장되지는 않습니다: 여러분의 운영 체제가 어떻게 스레드를 스케줄링 하는지에 따라 달린 문제입니다. 위의 실행 예에서는 생성된 스레드로부터의 출력 구문이 코드의 첫번째에 나타나 있음에도 불구하고 메인 스레드가 먼저 출력하였습니다. 그리고 생성된 스레드에게 `i`가 9일때까지 출력하라고 했음에도 불구하고, 메인 스레드가 멈추기 전까지 고작 5에 도달했습니다.

만일 여러분이 이 코드를 실행하고 메인 스레드로부터의 출력만 보았다면, 혹은 어떠한 오버랩도 보지 못했다면, 숫자 범위를 늘려서 운영 체제로 하여금 스레드간의 전환에 더 많은 기회를 주는 시도를 해보세요.

join 핸들을 사용하여 모든 스레드들이 끝날때까지 기다리기

Listing 16-1의 코드는 대개의 경우 메인 스레드가 종료되는 이유로 생성된 스레드가 조기에 멈출 뿐만 아니라, 생성된 스레드가 모든 코드를 실행할 것임을 보장해 줄수도 없습니다. 그 이유는 스레드들이 실행되는 순서에 대한 보장이 없기 때문입니다!

생성된 스레드가 실행되지 않거나, 전부 실행되지 않는 문제는 `thread::spawn`의 반환값을 변수에 저장함으로서 해결할 수 있습니다. `thread::spawn`의 반환 타입은 `JoinHandle`입니다. `JoinHandle`은 이것에 가지고 있는 `join` 메소드를 호출했을 때 그 스레드가 끝날때까지 기다리는 소유된 값입니다. Listing 16-2는 어떤식으로 우리가 Listing 16-1에서 만들었던 스레드의 `JoinHandle`을 사용하고 `join`을 호출하여 `main`이 끝나기 전에 생성된 스레드가 종료되는 것을 확실하게 하는지를 보여줍니다:

Filename: src/main.rs

```

use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}

```

Listing 16-2: 스레드가 완전히 실행되는 것을 보장하기 위해 `thread::spawn`으로부터 `JoinHandle`을 저장하기

핸들에 대해 `join`을 호출하는 것은 핸들에 대한 스레드가 종료될 때까지 현재 실행중인 스레드를 블록합니다. 스레드를 블록(*Block*)한다는 것은 그 스레드의 작업을 수행하거나 종료되는 것이 방지된다는 의미입니다. 우리가 메인 스레드의 `for` 루프 이후에 `join`의 호출을 넣었으므로, Listing 16-2의 실행은 아래와 비슷한 출력을 만들어야 합니다:

```

hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!

```

두 스레드가 교차를 계속하지만, `handle.join()`의 호출로 인하여 메인 스레드는 기다리고 생성된 스레드가 종료되기 전까지 끝나지 않습니다.

그런데 만일 아래와 같이 `main`의 `for` 루프 이전으로 `handle.join()`을 이동시키면 어떤 일이 생기는지 봅시다:

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

메인 스레드는 생성된 스레드가 종료될 때까지 기다릴 것이고 그 다음 자신의 `for` 루프를 실행시키게 되어, 아래처럼 출력값이 더 이상 교차되지 않을 것입니다:

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

`join`이 호출되는 위치와 같은 작은 디테일들이 여러분의 스레드가 동시에 실행되는지 혹은 아닌지에 대해 영향을 미칠 수 있습니다.

스레드에 `move` 클로저 사용하기

`move` 클로저는 `thread::spawn`과 함께 자주 사용되는데 그 이유는 이것이 여러분으로 하여금 어떤 스레

드의 데이터를 다른 스레드 내에서 사용하도록 해주기 때문입니다.

13장에서는 클로저의 파라미터 목록 앞에 `move` 키워드를 이용하여 클로저가 그 환경에서 사용하는 값의 소유권을 강제로 갖게한다고 언급했습니다. 이 기술은 값의 소유권을 한 스레드에서 다른 스레드로 이전하기 위해 새로운 스레드를 생성할 때 특히 유용합니다.

Listing 16-1에서 우리가 `thread::spawn`에 넘기는 클로저는 아무런 인자도 갖지 갖지 않는다는 점을 주목하세요: 생성된 스레드의 코드 내에서는 메인 스레드로부터 온 어떤 데이터도 이용하고 있지 않습니다. 메인 스레드로부터의 데이터를 생성된 스레드 내에서 사용하기 위해서는 생성된 스레드의 클로저가 필요로 하는 값을 캡처해야 합니다. Listing 16-3은 메인 스레드에서 백터 생성하여 이를 생성된 스레드 내에서 사용하는 시도를 보여주고 있습니다. 그러나 잠시 후에 보시게 될 것처럼 아직은 동작하지 않습니다.

Filename: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Listing 16-3: 메인 스레드에서 생성된 벡터를 다른 스레드 내에서 사용하는 시도

클로저는 `v`를 사용하므로, `v`는 캡처되어 클로저의 환경의 일부가 됩니다. `thread::spawn`이 이 클로저를 새로운 스레드 내에서 실행하므로, `v`는 새로운 스레드 내에서 접근 가능해야 합니다. 하지만 이 예제를 컴파일하면 아래와 같은 에러를 얻게 됩니다:

```

error[E0373]: closure may outlive the current function, but it borrows `v`,
which is owned by the current function
--> src/main.rs:6:32
   |
6 |     let handle = thread::spawn(|| {
   |                         ^^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
   |                     - `v` is borrowed here
   |
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
   |
6 |     let handle = thread::spawn(move || {
   |                         ^^^^^^

```

러스트는 `v`를 어떻게 캡처하는지 추론하고, `println!`이 `v`의 참조자만 필요로 하기 때문에, 클로저는 `v`를 빌리는 시도를 합니다. 하지만 문제가 있습니다: 러스트는 생성된 스레드가 얼마나 오랫동안 실행될지 말해줄 수 없으므로, `v`에 대한 참조자가 항상 유효할 것인지를 알지 못합니다.

Listing 16-4는 유효하지 않게 된 `v`의 참조자를 갖게 될 가능성이 더 높은 시나리오를 제공합니다:

Filename: src/main.rs

```

use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}

```

Listing 16-4: `v`를 드롭하는 메인 스레드로부터 `v`에 대한 참조자를 캡처하는 시도를 하는 클로저를 갖는 스레드

만약 우리가 이 코드를 실행할 수 있다면, 생성된 스레드가 전혀 실행되지 않고 즉시 백그라운드에 들어갈 가능성이 있습니다. 생성된 스레드는 내부에 `v`의 참조자를 가지고 있지만, 메인 스레드는 우리가 15장에서 다루었던 `drop` 함수를 사용하여 `v`를 즉시 드롭시킵니다. 그러면 생성된 스레드가 실행되기 시작할 때 `v`가 더 이상 유효하지 않게 되어, 참조자 또한 유효하지 않게 됩니다. 이런!

Listing 16-3의 컴파일 에러를 고치기 위해서는 에러 메세지의 조언을 이용할 수 있습니다:

```
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
|
6 |     let handle = thread::spawn(move || {
|           ^^^^^^
```

move 키워드를 클로저 앞에 추가함으로서 우리는 러스트가 값을 빌려와야 된다고 추론하도록 하는 것이 아니라 사용하는 값의 소유권을 강제로 가지도록 합니다. Listing 16-3을 Listing 16-5에서 보이는 것처럼 수정하면 컴파일되어 우리가 원하는 대로 실행됩니다:

Filename: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Listing 16-5: **move** 키워드를 사용하여 사용하는 값의 소유권을 클로저가 갖도록 강제하기

메인 스레드에서 **drop**을 호출하는 Listing 16-4의 코드에서 **move** 클로저를 이용한다면 어떤 일이 벌어질까요? **move**가 이 경우도 고칠 수 있을까요? 불행하게도, 아닙니다; Listing 16-4이 시도하고자 하는 것이 다른 이유로 허용되지 않기 때문에 우리는 다음 에러를 얻게 됩니다. 만일 클로저에 **move**를 추가하면, **v**를 클로저의 환경으로 이동시킬 것이고, 더이상 메인 스레드에서 이것에 대한 **drop** 호출을 할 수 없게 됩니다. 대신 우리는 아래와 같은 컴파일 에러를 얻게 됩니다:

```
error[E0382]: use of moved value: `v`
--> src/main.rs:10:10
|
6 |     let handle = thread::spawn(move || {
|           ^----- value moved (into closure) here
...
10 |     drop(v); // oh no!
|           ^ value used here after move
|
= note: move occurs because `v` has type `std::vec::Vec<i32>`, which does
not implement the `Copy` trait
```

러스트의 소유권 규칙이 다시 한번 우리를 구해주었습니다! Listing 16-3의 코드로부터 에러를 받은 이유는 러스트가 보수적이고 스레드를 위해 `v`를 단지 빌리려고만 했기 때문이었는데, 이는 메인스레드가 이론적으로 생성된 스레드의 참조자를 무효화할 수 있음을 의미합니다. 러스트에게 `v`의 소유권을 생성된 스레드로 이동시키라 말해줌으로서, 우리는 러스트에게 메인 스레드가 `v`를 더 이상 이용하지 않음을 보장하고 있습니다. 만일 우리가 Listing 16-4를 같은 방식으로 바꾸면, 우리가 `v`를 메인스레드 상에서 사용하고자 할 때 소유권 규칙을 위반하게 됩니다. `move` 키워드는 러스트의 빌림에 대한 보수적인 기본 기준을 무효화합니다; 즉 우리가 소유권 규칙을 위반하지 않도록 해줍니다.

스레드와 스레드 API에 대한 기본적인 이해를 하고서, 우리가 스레드를 가지고 어떤 것을 할 수 있는지 살펴봅시다.

메세지 패싱을 사용하여 스레드 간에 데이터 전송하기

안전한 동시성을 보장하는 인기 상승중인 접근법 하나는 **메세지 패싱** (*message passing*) 인데, 이는 스레드들 혹은 액터들이 데이터를 담고 있는 메세지를 서로 주고받는 것입니다. [Go 언어 문서](#)의 슬로건에 있는 아이디어는 다음과 같습니다: "메모리를 공유하는 것으로 통신하지 마세요; 대신, 통신해서 메모리를 공유하세요"

러스트가 메세지 보내기 방식의 동시성을 달성하기 위해 갖춘 한가지 주요 도구는 **채널** (*channel*) 인데, 이는 러스트의 표준 라이브러리가 구현체를 제공하는 프로그래밍 개념입니다. 프로그래밍에서의 채널은 개울이나 강 같은 물의 통로와 비슷하다고 상상할 수 있습니다. 만일 여러분이 고무 오리나 배 같은 것을 개울에 띄우면, 물길의 끝까지 하류로 여행하게 될 것입니다.

프로그래밍에서의 채널은 둘로 나뉘어져 있습니다: 바로 송신자(transmitter)와 수신자(receiver)입니다. 송신자 측은 여러분이 강에 고무 오리를 띄우는 상류 위치이고, 수신자 측은 하류에 고무 오리가 도달하는 곳입니다. 여러분 코드 중 한 곳에서 여러분이 보내고자 하는 데이터와 함께 송신자의 메소드를 호출하면, 다른 곳에서는 도달한 메세지에 대한 수신 종료를 검사합니다. 송신자 혹은 송신자가 드롭되면 채널이 **닫혔다** (*closed*) 라고 말합니다.

여기서 우리는 값을 생성하여 채널로 내려보내는 한 스레드와, 값을 받아서 이를 출력하는 또 다른 스레드를 가지고 있는 프로그램을 만들어볼 것입니다. 우리는 기능을 설명하기 위해서 채널을 사용해 스레드 간에 단순한 값을 보내게 될 것입니다. 여러분이 이 기술에 익숙해지고 나면, 여러분은 채팅 시스템이나 다수의 스레드가 계산의 일부분을 수행하여 결과를 종합하는 하나의 스레드에 이를 보내는 시스템을 구현하기 위해 채널을 이용할 수 있습니다.

먼저 Listing 16-6에서는 채널을 만들지만 이걸 가지고 아무것도 하지 않을 것입니다. 우리가 채널을 통해 어떤 타입의 값을 보내는지에 대해 러스트에게 말하지 않았기 때문에 아직 컴파일되지 않는다는 점을 주의하세요.

Filename: src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

Listing 16-6: 채널을 생성하여 두 결과값을 `tx` 와 `rx`에 할당하기

우리는 `mpsc::channel` 함수를 사용하여 새로운 채널을 생성합니다; `mpsc` 는 복수 생성자, 단수 소비자 (*multiple producer, single consumer*) 를 나타냅니다. 짧게 줄이면, 러스트의 표준 라이브러리가 채널을 구현한 방법은 한 채널이 값을 생성하는 복수개의 송신/단말을 가질 수 있지만 값을 소비하는 단 하나의 수신 단말을 가질 수 있음을 의미합니다. 하나의 큰 강으로 함께 흐르는 여러 개울들을 상상해 보세요: 개울 중 어

떤 쪽에라도 흘려보낸 모든 것은 끝에 하나의 강에서 끝날 것입니다. 지금은 단일 생성자를 가지고 시작하겠지만, 이 예제가 동작하기 시작하면 여러 생성자를 추가할 것입니다.

`mpsc::channel` 함수는 튜플을 반환하는데, 첫번째 요소는 송신 단말이고 두번째 요소는 수신 단말입니다. `tx` 와 `rx`라는 약어는 많은 분야에서 각각 송신자 (*transmitter*) 와 수신자 (*receiver*) 를 위해 사용하므로, 각각의 단말을 가리키기 위해 그렇게 변수명을 지었습니다. 우리는 튜플을 해체하는 패턴과 함께 `let` 구문을 사용하는 중입니다; `let` 구문 내에서의 패턴의 사용과 해체에 대해서는 18장에서 다룰 것입니다. 이런 방식으로 `let` 구문을 사용하는 것은 `mpsc::channel`이 반환하는 튜플의 조각들을 추출하는데 편리한 접근법입니다.

Listing 16-7에서 보는 바와 같이 송신 단말을 생성된 스레드로 이동시키고 하나의 스트링을 전송하게 하여 생성된 스레드가 메인 스레드와 통신하도록 해봅시다. 이는 강 상류에 고무 오리를 띄우는 것 혹은 한 스레드에서 다른 스레드로 채팅 메세지를 보내는 것과 비슷합니다.

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

Listing 16-7: `tx`를 생성된 스레드로 이동시키고 “hi”를 보내기

다시 한번 `thread::spawn`을 이용하여 새로운 스레드를 생성한 뒤 `move`를 사용하여 `tx`를 클로저로 이동시켜 생성된 스레드가 `tx`를 소유하도록 합니다. 생성된 스레드는 채널을 통해 메세지를 보낼 수 있도록 하기 위해 채널의 송신 단말을 소유할 필요가 있습니다.

송신 단말은 우리가 보내고 싶어하는 값을 취하는 `send` 메소드를 가집니다. `send` 메소드는 `Result<T, E>` 타입을 반환하므로, 만일 수신 단말이 이미 드롭되어 있고 값을 보내는 곳이 없다면, 송신 연산은 에러를 반환할 것입니다. 이 예제에서는 에러가 나는 경우 패닉을 일으키기 위해 `unwrap`을 호출하는 중입니다. 그러나 실제 애플리케이션에서는 이를 적절히 다뤄야 할 것입니다: 적절한 에러 처리를 위한 전략을 다시 보려면 9장으로 돌아가세요.

Listing 16-8에서 우리는 메인 스레드에 있는 채널의 수신 단말로부터 값을 받을 것입니다. 이는 강의 끝물에서 고무 오리를 건져올리는 것 혹은 채팅 메세지를 받는 것과 비슷합니다.

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Listing 16-8: 메인 스레드에서 "hi" 값을 받아 출력하기

채널의 수신 단말은 두 개의 유용한 메소드를 가지고 있습니다: `recv`와 `try_recv`입니다. 우리는 수신 (`receive`) 의 줄임말인 `recv`를 사용하는 중인데, 이는 메인 스레드의 실행을 블록시키고 채널로부터 값이 보내질 때까지 기다릴 것입니다. 값이 일단 전달되면, `recv`는 `Result<T, E>` 형태로 이를 반환할 것입니다. 채널의 송신 단말이 닫히면, `recv`는 더 이상 어떤 값도 오지 않을 것이란 신호를 하는 에러를 반환할 것입니다.

`try_recv` 메소드는 블록하지 않는 대신 즉시 `Result<T, E>`를 반환합니다: 전달 받은 메세지가 있다면 이를 담고 있는 `Ok` 값을, 이 시점에서 메세지가 없다면 `Err` 값을 반환합니다. `try_recv`를 사용하는 것은 메세지를 기다리는 동안 해야 하는 다른 작업이 있을 때 유용합니다: `try_recv`을 매번마다 호출하여, 가능한 메세지가 있으면 이를 처리하고, 그렇지 않으면 다음번 검사때까지 잠시동안 다른 일을 하는 루프를 만들 수 있습니다.

이 예제에서는 단순함을 위해 `recv`를 이용했습니다; 이 메인 스레드에서는 메세지를 기다리는 동안 해야 할 다른 일이 없으므로, 메인 스레드를 블록시키는 것이 적절합니다.

Listing 16-8의 코드를 실행하면, 메인 스레드로부터 출력된 값을 보게 될 것입니다:

Got: hi

완벽하군요!

채널과 소유권 전달

소유권 규칙은 여러분들이 안전하고 동시적인 코드를 작성하는 것을 돋기 때문에 메세지 보내기 방식 내에서 강건한 역할을 합니다. 동시성 프로그래밍 내에서 에러를 방지하는 것은 여러분의 러스트 프로그램 전체에 걸친 소유권에 대한 생각해볼 수 있는 장점이 있습니다. 어떤 식으로 채널과 소유권이 문제를 방지하기 위해 함께 동작하는지를 보기 위한 실험을 해봅시다: 우리가 채널로 `val` 값을 내려보낸 이후에 생성된 스레드에서 이 값을 사용하는 시도를 해볼 것입니다. Listing 16-9의 코드를 컴파일하여 이 코드가 왜 허용되지 않는지를 보세요:

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Listing 16-9: `val`을 채널로 내려보낸 뒤 이에 대한 사용 시도

여기서는 `tx.send`를 통하여 채널에 `val`을 내려보낸 뒤 이를 출력하는 시도를 하였습니다. 이 코드를 허용하는 것은 나쁜 생각입니다: 일단 값이 다른 스레드로 보내지고 나면, 우리가 값을 다시 사용해보기 전에 그 스레드에서 수정되거나 버려질 수 있습니다. 잠재적으로, 다른 스레드에서의 수정은 불일치하거나 존재하지 않는 데이터로 인한 에러를 일으킬 수 있습니다. 그러나, 우리가 Listing 16-9의 코드를 컴파일 시도하면 러스트는 에러를 내놓습니다:

```
error[E0382]: use of moved value: `val`
--> src/main.rs:10:31
   |
9  |         tx.send(val).unwrap();
   |                 ^-- value moved here
10 |         println!("val is {}", val);
   |                     ^^^ value used here after move
   |
= note: move occurs because `val` has type `std::string::String`, which
does
not implement the `Copy` trait
```

우리의 동시성에 관한 실수가 컴파일 타임 에러를 야기했습니다. `send` 함수가 그 파라미터의 소유권을 가져가고, 이 값이 이동될 때, 수신자가 이에 대한 소유권을 얻습니다. 이는 우리가 값을 보낸 이후에 우발적으로 이 값을 다시 사용하는 것을 방지합니다; 소유권 시스템은 모든게 정상인지 확인합니다.

복수의 값을 보내고 수신자가 기다리는지 보기

Listing 16-8의 코드는 컴파일되고 실행도 되지만, 두개의 분리된 스레드가 채널을 통해 서로 대화를 했는지를 우리에게 명확히 보여주진 못했습니다. Listing 16-10에서는 Listing 16-8의 코드가 동시에 실행된다는 것을 입증해 중 수정본을 만들었습니다: 이제 생성된 스레드가 여러 메세지를 보내면서 각 메세지 사이에 1초씩 잠깐 멈출 것입니다.

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Listing 16-10: 여러 메세지를 보내고 각 사이마다 멈추기

이번에 생성된 스레드는 우리가 메인 스레드로 보내고 싶어하는 스트링의 벡터를 가지고 있습니다. 스트링마다 반복하여 각각의 값을 개별적으로 보내고, `Duration` 값에 1을 넣어서 `thread::sleep` 함수를 호출하는 것으로 각각의 사이에 멈춥니다.

메인 스레드에서는 더 이상 `recv` 함수를 명시적으로 호출하지 않고 있습니다: 대신 `rx`를 반복자처럼 다루고 있습니다. 각각의 수신된 값에 대해서 이를 출력합니다. 채널이 닫힐 때는 반복이 종료될 것입니다.

Listing 16-10의 코드를 실행시키면 다음과 같은 출력이 각 줄마다 1초씩 멈추면서 보일 것입니다:

```
Got: hi
Got: from
Got: the
Got: thread
```

메인 스레드의 `for` 루프 내에는 어떠한 멈춤 혹은 지연 코드를 넣지 않았으므로, 우리는 메인 스레드가 생성된 스레드로부터 값을 전달받는 것을 기다리는 중이라고 말할 수 있습니다.

송신자를 복제하여 여러 생성자 만들기

이전에 `mpsc`가 복수 생성자 단일 소비자 (*multiple producer, single consumer*)의 약어라는 것을 언급했었지요. `mpsc`를 Listing 16-10의 코드에 넣어 모두 동일한 수신자로 값을 보내는 여러 스레드들을 만들도록 코드를 확장해봅시다. Listing 16-11에서 보시는 것처럼 채널의 송신자를 복제하는 것으로 그렇게 할 수 있습니다:

Filename: src/main.rs

```
// --snip--  
  
let (tx, rx) = mpsc::channel();  
  
let tx1 = mpsc::Sender::clone(&tx);  
thread::spawn(move || {  
    let vals = vec![  
        String::from("hi"),  
        String::from("from"),  
        String::from("the"),  
        String::from("thread"),  
    ];  
  
    for val in vals {  
        tx1.send(val).unwrap();  
        thread::sleep(Duration::from_secs(1));  
    }  
});  
  
thread::spawn(move || {  
    let vals = vec![  
        String::from("more"),  
        String::from("messages"),  
        String::from("for"),  
        String::from("you"),  
    ];  
  
    for val in vals {  
        tx.send(val).unwrap();  
        thread::sleep(Duration::from_secs(1));  
    }  
});  
  
for received in rx {  
    println!("Got: {}", received);  
}  
  
// --snip--
```

Listing 16-11: 여러 개의 생성자로부터 여러 메세지 보내기

이번에는 우리가 첫번째 스레드를 생성하기 전에, 채널의 송신 단말에 대해 `clone`을 호출했습니다. 이는 우리에게 첫번째 생성된 스레드로 값을 보낼 수 있는 새로운 송신 핸들을 제공해줄 것입니다. 두번째 생성된 스레드에게는 원래의 채널 송신 단말을 넘깁니다. 이렇게 함으로써 각각이 다른 메세지를 채널의 수신 단말로 보내주는 두 스레드를 만듭니다.

여러분이 이 코드를 실행시키면, 다음과 같은 출력과 비슷하게 보여야 합니다:

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

값들의 순서가 다르게 보일 수도 있습니다; 이는 여러분의 시스템에 따라 다릅니다. 이것이 바로 동시성을 흥미로울 뿐만 아니라 어렵게 만드는 것입니다. 만일 여러분이 `thread::sleep`을 가지고 실험하면서 서로 다른 스레드마다 다양한 값을 썼다면, 각각의 실행이 더욱 비결정적이고 매번 다른 출력을 생성할 것입니다.

이제 채널이 어떤 식으로 동작하는지 보았으니, 동시성을 위한 다른 방법을 알아봅시다.

공유 상태 동시성

메세지 패싱은 동시성을 다루는 좋은 방법이지만, 유일한 수단은 아닙니다. Go 언어 문서로부터 나온 슬로건의 일부를 다시한번 고려해보죠: “메모리를 공유함으로써 소통하세요.”

메모리를 공유하는 통신은 어떤 형태로 보일까요? 더불어서 메세지 패싱의 열광적인 지지자들은 왜 이걸 안 쓰고 대신 반대편의 것을 쓸까요?

어떤 면에서, 프로그래밍 언어의 채널들은 단일 소유권과 유사한데, 이는 여러분이 채널로 값을 송신하면, 그 값을 더이상 쓸 수 없게되기 때문입니다. 공유 메모리 동시성은 복수 소유권과 유사합니다: 복수개의 스레드들이 동시에 동일한 메모리 위치를 접근할 수 있지요. 스마트 포인터들이 복수 소유권을 가능하게 만드는 내용을 담은 15장에서 보셨듯이, 복수 소유권은 이 서로 다른 소유자들의 관리가 필요하기 때문에 복잡성을 더 할 수 있습니다. 러스트의 타입 시스템과 소유권 규칙은 이러한 관리를 올바르도록 훌륭히 유도합니다. 예를 들면, 공유 메모리를 위한 더 일반적인 동시성의 기초 재료 중 하나인 뮤텍스(mutex)를 살펴 봅시다.

뮤텍스를 사용하여 한번에 한 스레드에서의 데이터 접근을 허용하기

뮤텍스는 상호 배제(mutual exclusion)의 줄임말로서, 내부에서 뮤텍스는 주어진 시간에 오직 하나의 스레드만 데이터 접근을 허용합니다. 뮤텍스 내부의 데이터에 접근하기 위해서 스레드는 먼저 뮤텍스의 락(lock)을 얻기를 요청함으로써 접근을 원한다는 신호를 보내야 합니다. 락은 누가 배타적으로 데이터에 접근하는지를 추적하는 뮤텍스의 부분인 데이터 구조입니다. 그러므로, 뮤텍스는 잠금 시스템을 통해 가지고 있는 데이터를 보호하는 것으로 묘사됩니다.

뮤텍스는 사용하기 어렵다는 평판을 가지고 있는데 이는 여러분이 다음 두 가지 규칙을 기억해야 하기 때문입니다:

- 여러분은 데이터를 사용하기 전에 반드시 락을 얻는 시도를 해야 합니다.
- 만일 뮤텍스가 보호하는 데이터의 사용이 끝났다면, 다른 스레드들이 락을 얻을 수 있도록 반드시 언락해야 합니다.

뮤텍스에 대한 실세계 은유를 위해서, 마이크가 딱 하나만 있는 컨퍼런스 패널 토의를 상상해보세요. 패널 참가자들이 말하기 전, 그들은 마이크 사용을 원한다고 요청하거나 신호를 줘야 합니다. 마이크를 얻었을 때는 원하는 만큼 길게 말을 한 다음 말하기를 원하는 다음 패널 참가자에게 마이크를 건네줍니다. 만일 패널 참여자가 마이크 사용을 끝냈을 때 이를 건네주는 것을 잊어먹는다면, 그 외 아무도 말할 수 없게 됩니다. 공유된 마이크의 관리가 잘못되면, 패널은 계획된데로 되지 않을겁니다!

뮤텍스의 관리는 바로잡기 위해 믿을 수 없으리만치 교묘해질 수 있는데, 이것이 바로 많은 사람들이 채널의 열성 지지자가 되는 이유입니다. 하지만, 러스트의 타입 시스템과 소유권 규칙에 감사하게도, 여러분은 잘못 락을 얻거나 언락 할 수가 없습니다.

Mutex<T>의 API

어떻게 뮤텍스를 이용하는지에 대한 예제로서, Listing 16-12와 같이 단일 스레드 맥락 내에서 뮤텍스를 사용하는 것으로 시작해봅시다:

Filename: src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Listing 16-12: 단순함을 위해 단일 스레드 맥락 내에서 `Mutex<T>`의 API 탐색하기

많은 타입들처럼 `Mutex<T>`는 연관함수 `new`를 사용하여 만들어집니다. 뮤텍스 내의 데이터에 접근하기 위해서는 `lock` 메소드를 사용하여 락을 얻습니다. 이 호출은 현재의 스레드를 막아설 것이므로, 락을 얻는 차례가 될 때까지 아무런 작업도 할 수 없습니다.

`lock`의 호출은 다른 스레드가 패닉 상태의 락을 가지고 있을 경우 실패할 수 있습니다. 그런 경우 아무도 락을 얻을 수 없게 되므로, `unwrap`을 택하여 그런 상황일 경우 이 스레드에 패닉을 일으킵니다.

락을 얻고난 다음에는 그 반환값 (위의 경우에는 `num`이라는 이름의 값) 을 내부의 데이터에 대한 가변 참조자처럼 다를 수 있습니다. 타입 시스템은 `m` 내부의 값을 사용하기 전에 우리가 락을 얻는 것을 확실히 해줍니다: `Mutex<i32>`는 `i32`가 아니므로 우리는 반드시 `i32` 값을 사용하기 위해 락을 얻어야 합니다. 우리는 이를 잊어버릴 수 없습니다; 잊어버린다면 타입 시스템이 내부의 `i32`에 접근할 수 없게 할 것입니다.

여러분이 의심한 것처럼, `Mutex<T>`는 스마트 포인터입니다. 더 정확하게는, `lock`의 호출은 `MutexGuard`라고 불리우는 스마트 포인터를 반환합니다. 이 스마트 포인터는 우리의 내부 데이터를 가리키도록 `Deref`가 구현되어 있습니다; 이 스마트 포인터는 또한 `MutexGuard`가 스코프 밖으로 벗어났을 때 자동으로 락을 해제하는 `Drop` 구현체를 가지고 있는데, 이는 Listing 16-12의 내부 스코프의 끝에서 일어나는 일입니다. 결과적으로 락이 자동으로 해제되기 때문에, 우리는 락을 해제하는 것을 잊어버리고 다른 스레드에 의해 뮤텍스가 사용되는 것을 막는 위험을 짊어지지 않아도 됩니다.

락이 버려진 후, 뮤텍스 값을 출력하여 내부의 `i32`를 6으로 바꿀 수 있음을 확인할 수 있습니다.

여러 스레드들 사이에서 **Mutex<T>** 공유하기

이제 **Mutex<T>**를 사용하여 여러 스레드들 사이에서 값을 공유하는 시도를 해봅시다. 우리는 10개의 스레드를 돌리고 이들이 카운터 값을 1만큼씩 증가 시켜서, 카운터가 0에서 10으로 가도록 할 것입니다. 다음 몇 개의 예제가 컴파일 에러가 날 것이고, 우리가 이 에러를 사용하여 **Mutex<T>**를 사용하는 방법과 러스트가 이를 고치는 것을 어떻게 돋는지에 대해 학습할 것임을 주의하세요. Listing 16-13이 시작 예제입니다:

Filename: src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Listing 16-13: **Mutex<T>**에 의해 보호되는 카운터를 각자 증가시키는 10개의 스레드

Listing 16-12에서 했던 것처럼 **Mutex<T>** 내부에 **i32**를 담는 **counter** 변수를 만듭니다. 그 다음, 숫자 범위로 반복하여 10개의 스레드를 만듭니다. 우리는 **thread::spawn**을 사용하여 동일한 클로저를 모든 스레드에게 주었는데, 이 클로저는 스레드로 카운터를 이동시키고, **lock** 메소드를 호출함으로써 **Mutex<T>**의 락을 얻은 다음, 뮤텍스 내의 값을 1만큼 증가시킵니다. 스레드가 자신의 클로저 실행을 끝냈을 때, **num**은 스코프 밖으로 벗어내고 락이 해제되어 다른 스레드가 이를 얻을 수 있습니다.

메인 스레드에서 우리는 조인 핸들을 전부 모읍니다. 그리고나서 Listing 16-2에서 했던 것처럼, 각 핸들에 **join**을 호출하여 모든 스레드가 종료되는 것을 확실히 합니다. 이 시점에서 메인 스레드는 락을 얻고 이 프로그램의 결과를 출력합니다.

이 예제가 컴파일되지 않는다는 힌트를 줬었죠. 이제 왜 그런지 알아봅시다!

```

error[E0382]: capture of moved value: `counter`
--> src/main.rs:10:27
|
9 |         let handle = thread::spawn(move || {
|                           ----- value moved (into closure)
here
10|             let mut num = counter.lock().unwrap();
|                           ^^^^^^^^ value captured here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
--> src/main.rs:21:29
|
9 |         let handle = thread::spawn(move || {
|                           ----- value moved (into closure)
here
...
21|             println!("Result: {}", *counter.lock().unwrap());
|                           ^^^^^^^^ value used here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error: aborting due to 2 previous errors

```

이 에러 메세지는 `counter` 값이 클로저 내부로 이동되어서 우리가 `lock`을 호출할 때 캡처되었다고 설명합니다. 이 설명은 우리가 원하는 것처럼 들리지만, 허용되지 않습니다!

프로그램을 단순화하여 이를 알아내봅시다. 10개의 스레드를 `for` 루프 내에서 만드는 대신, 루프 없이 두 개의 스레드만 만들어서 어떤 일이 일어나는지 봅시다. Listing 16-13의 첫번째 `for` 루프를 아래 코드로 바꿔 넣으세요:

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();

        *num += 1;
    });
    handles.push(handle);

    let handle2 = thread::spawn(move || {
        let mut num2 = counter.lock().unwrap();

        *num2 += 1;
    });
    handles.push(handle2);

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

우리는 두 개의 스레드를 만들고 두 번째 스레드에서 사용되는 변수 이름을 `handle2` 와 `num2` 로 바꿨습니다. 이제 이 코드를 실행하면, 컴파일러가 우리에게 다음 에러 메세지를 줍니다:

```

error[E0382]: capture of moved value: `counter`
--> src/main.rs:16:24
8 |     let handle = thread::spawn(move || {
|                         ----- value moved (into closure) here
...
16 |         let mut num2 = counter.lock().unwrap();
|                         ^^^^^^^^ value captured here after move
|
|= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
--> src/main.rs:26:29
8 |     let handle = thread::spawn(move || {
|                         ----- value moved (into closure) here
...
26 |         println!("Result: {}", *counter.lock().unwrap());
|                         ^^^^^^^^ value used here after move
|
|= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error: aborting due to 2 previous errors

```

아하! 첫번째 에러 메세지는 `counter` 가 `handle`과 연관된 스레드에 대한 클로저 내부로 이동되었음을 나타냅니다. 이 이동이 우리가 두번째 스레드에서 `lock`의 호출을 시도하고 `num2`에 결과를 저장할 때 `counter`를 캡처하는 것을 방지합니다! 따라서 러스트는 우리가 `counter`의 소유권을 여러 스레드로 이동시킬 수 없음을 말하는 중입니다. 이는 더 일찍 발견하기 어려운데 그 이유는 우리의 스레드들이 루프 내에 있었고, 러스트는 루프의 다른 반복 회차 내의 다른 스레드를 지적할 수 없기 때문입니다. 우리가 15장에서 다루었던 복수 소유자 메소드를 이용하여 이 컴파일에러를 고쳐봅시다.

여러 스레드들과 함께하는 복수 소유권

15장에서 우리는 참조 카운팅 값을 만들기 위해 스마트 포인터 `Rc<T>` 을 사용함으로써 값에게 복수의 소유권자를 주었습니다. 동일한 일을 여기서도 해서 어떻게 되는지 봅시다. Listing 16-14에서 `Mutex<T>` 를 `Rc<T>` 로 감싸서 스레드로 소유권을 이동시키기 전에 이 `Rc<T>` 를 복제하겠습니다. 이제는 우리가 에러를 봤으므로, `for` 루프를 이용하도록 다시 전환하고 클로저와 함께 쓴 `move` 키워드를 유지하겠습니다.

Filename: src/main.rs

```

use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}

```

Listing 16-14: 여러 스레드가 `Mutex<T>`를 소유할 수 있도록 `Rc<T>`를 사용하는 시도

다시 한번 컴파일을 하고 그 결과가... 다른 에러들이네요! 컴파일러는 우리에게 많은 것을 가르치고 있습니다.

```

error[E0277]: the trait bound `std::rc::Rc<std::sync::Mutex<i32>>: std::marker::Send` is not satisfied in `[closure@src/main.rs:11:36: 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`  

--> src/main.rs:11:22  

|  

|         let handle = thread::spawn(move || {  

|             ^^^^^^^^^^^^^ `std::rc::Rc<std::sync::Mutex<i32>>`  

cannot be sent between threads safely  

|  

= help: within `[closure@src/main.rs:11:36: 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`, the trait `std::marker::Send` is not implemented for `std::rc::Rc<std::sync::Mutex<i32>>`  

= note: required because it appears within the type `[closure@src/main.rs:11:36: 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`  

= note: required by `std::thread::spawn`
```

와우, 이 에러는 정말 장황하네요! 여기 초점을 맞출 몇몇 중요한 부분이 있습니다: 첫번째 인라인 에러는 `std::rc::Rc<std::sync::Mutex<i32>>`는 스레드 사이에 안전하게 보내질 수 없다 라고 말합니다. 이에 대한 이유는 초점을 맞출 그 다음 중요한 부분인 에러 메세지 내에 있습니다. 정제된 에러 메세지는 트레이잇 바운드 `Send`가 만족되지 않았다 라고 말합니다. `Send`는 다음 절에서 얘기할 것입니다: 이것은 우리가 스레드와 함께 사용하는 타입들이 동시적 상황들 내에서 쓰이기 위한 것임을 확실히 하는 트레이잇 중 하나입니다.

안타깝게도, `Rc<T>`는 스레드를 교차하면서 공유하기에는 안전하지 않습니다. `Rc<T>`가 참조 카운트를 관리할 때, 각각의 `clone` 호출마다 카운트에 더하고 각 클론이 버려질 때마다 카운트에서 제합니다. 하지만 그것은 다른 스레드에 의해 카운트를 변경하는 것을 방해할 수 없도록 확실히 하는 어떠한 동시성 기초 재료도 이용하지 않습니다. 이는 잘못된 카운트를 야기할 수 있습니다-결과적으로 메모리 누수를 발생시키거나 아직 다 쓰기 전에 값이 버려질 수 있는 교묘한 버그를 낳겠죠. 우리가 원하는 것은 정확히 `Rc<T>`와 비슷하지만 스레드-안전한 방식으로 참조 카운트를 바꾸는 녀석입니다.

Atomic Reference Counting with `Arc<T>`

`Arc<T>`을 이용하는 아토믹 (atomic) 참조 카운팅

다행히도, `Arc<T>`가 바로 동시적 상황에서 안전하게 사용할 수 있는 `Rc<T>` 타입입니다. `a`는 아토믹 (*atomic*)을 의미하는데, 즉 이것이 원자적으로 참조자를 세는 (*atomically reference counted*) 타입임을 의미합니다. 아토믹은 우리가 여기서 자세히 다루지 않을 추가적인 동시성 기초 재료 종류입니다: 더 자세히 알고 싶으면 `std::sync::atomic`에 대한 표준 라이브러리 문서를 보세요. 이 시점에서 여러분은 아토믹이 기초 타입처럼 동작하지만 스레드를 교차하며 공유해도 안전하다는 것만 알면 됩니다.

그렇다면 여러분은 왜 모든 기초 타입이 아토믹하지 않은지, 그리고 표준 라이브러리 타입은 왜 기본적으로 `Arc<T>`을 구현에 이용하지 않는지를 궁금해 할런지도 모르겠습니다. 그 이유는 스레드 안전성이란 것이 여러분이 정말로 원할 때만 지불하고 싶을 성능 저하를 일으키기 때문입니다. 만일 여러분이 단일 스레드 내의 값에 대한 연산만 수행하는 중이라면, 아토믹이 제공하는 보장을 강제하지 않아도 된다면 여러분의 코드는 더 빠르게 실행될 수 있습니다.

우리의 예제로 다시 돌아갑시다: `Arc<T>`와 `Rc<T>`는 같은 API를 가지고 있으므로, 우리는 `use`을 사용하는 라인과 `new` 호출, 그리고 `clone` 호출 부분을 바꾸는 것으로 프로그램을 수정합니다. Listing 16-15의 코드는 마침내 컴파일 및 실행이 될 것입니다:

Filename: src/main.rs

```

use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}

```

Listing 16-15: `Arc<T>`를 사용하여 `Mutex<T>`를 감싸서 여러 스레드 사이에서 소유권을 공유할 수 있도록 하기

이 코드는 다음을 출력할 것입니다:

Result: 10

해냈군요! 우리는 0부터 10까지 세었고, 이는 그렇게 크게 인상적인 것 같지 않을지도 모르겠지만, 우리에게 `Mutex<T>` 와 스레드 안전성에 대하여 많은 것을 가르쳐 주었습니다. 여러분은 또한 이 프로그램의 구조를 사용하여 단지 카운터를 증가시키는 것 보다 더 복잡한 연산을 할 수도 있습니다. 이 전략을 사용하여, 여러분은 계산할 것을 독립적인 부분들로 나누고, 해당 부분들을 스레드로 쪼갠 다음, `Mutex<T>`를 사용하여 각 스레드가 해당 부분의 최종 결과를 갱신하도록 할 수 있습니다.

RefCell<T>/Rc<T>와 Mutex<T>/Arc<T> 간의 유사성

여러분은 `counter` 이 불변적이지만 이것 내부의 값에 대한 가변 참조자를 가지고 올 수 있었음을 알아챘을 런지 모르겠습니다; 이는 `Mutex<T>` 가 `Cell` 가족이 그러하듯 내부 가변성을 제공한다는 의미입니다. 우리가 15장에서 `Rc<T>` 의 내용물을 변경할 수 있도록 하기 위해 `RefCell<T>` 을 사용한 것과 같은 방식으로, `Arc<T>` 내부의 값을 변경하기 위해 `Mutex<T>` 를 이용합니다.

주목할만한 또 다른 세부 사항은 여러분이 `Mutex<T>`를 사용할 때 러스트가 여러분을 모든 종류의 논리적 에러로부터 보호해줄 수 없다는 것입니다. 15장에서 `Rc<T>`를 사용하는 것은 두 `Rc<T>` 값들이 서로를 참조하여 메모리 누수를 야기하는 순환 참조자를 만들 위험성이 따라오는 것이었음을 상기하세요. 이와 유사하게, `Mutex<T>`는 데드락 (deadlock) 을 생성할 위험성이 따라옵니다. 이것은 어떤 연산이 두 개의 리소스에 대한 락을 얻을 필요가 있고 두 개의 스레드가 하나씩의 락을 얻는다면, 서로가 서로를 영원히 기다리는 식으로 발생됩니다. 여러분이 데드락에 흥미가 있다면, 데드락이 있는 러스트 프로그램 만들기를 시도해보세요; 그리고나서 어떤 언어에 있는 뮤텍스를 위한 데드락 완화 전략을 연구해보고 이를 러스트에서 구현해보세요. `Mutex<T>` 와 `MutexGuard`에 대한 표준 라이브러리 API 문서가 유용한 정보를 제공합니다.

이제 `Send` 와 `Sync` 트레이잇에 대해 얘기하고 커스텀 타입과 함께 어떻게 이용할 수 있는지에 대해 얘기하는 것으로 이 장을 마무리 하겠습니다.

Sync와 Send 트레잇을 이용한 확장 가능한 동시성

흥미롭게도, 러스트 언어는 매우 적은 숫자의 동시성 기능을 갖고 있습니다. 우리가 이 장에서 여지껏 얘기해온 거의 모든 동시성 기능들이 언어의 부분이 아니라 표준 라이브러리의 영역이었습니다. 동시성 제어를 위한 여러분의 옵션들은 언어 혹은 표준 라이브러리에 제한되지 않습니다; 여러분은 여러분만의 동시성 기능을 작성하거나 다른 이들이 작성한 것을 이용할 수 있습니다.

그러나, 두 개의 동시성 개념이 이 언어에 내재되어 있습니다: 바로 `std::marker` 트레잇인 `Sync` 와 `Send`입니다.

Send를 사용하여 스레드 사이에 소유권 이전을 허용하기

`Send` 마커 트레잇은 `Send` 가 구현된 타입의 소유권이 스레드 사이에서 이전될 수 있음을 나타냅니다. 거의 대부분의 러스트 타입이 `Send` 이지만, 몇 개의 예외가 있는데, 그 중 `Rc<T>` 도 있습니다: 이것은 `Send` 가 될 수 없는데 그 이유는 여러분이 `Rc<T>` 값을 클론하여 다른 스레드로 복제본의 소유권 전송을 시도한다면, 두 스레드 모두 동시에 참조 카운트 값을 갱신할지도 모르기 때문입니다. 이러한 이유로, `Rc<T>` 는 여러분이 스레드-안전성 성능 저하를 지불하지 않아도 되는 단일 스레드의 경우에 사용되도록 구현되었습니다.

따라서, 러스트의 타입 시스템과 트레잇 바운드는 여러분들이 결코 우연히라도 스레드 사이로 `Rc<T>` 값을 불안전하게 보낼 수 없도록 확실히 해줍니다. Listing 16-14의 것을 시도할 때, 우리는 `the trait Send is not implemented for Rc<Mutex<i32>>` 라는 에러를 얻었습니다. `Send` 가 구현된 `Arc<T>` 로 바꿨을 때는 코드가 컴파일 되었습니다.

전체적으로 `Send` 타입으로 구성된 어떤 타입은 또한 자동적으로 `Send` 로 마킹됩니다. 로우 포인터 (raw pointer)를 빼고 거의 모든 기초 타입이 `Send` 인데, 이는 19장에서 다루겠습니다.

Sync를 사용하여 여러 스레드로부터의 접근을 허용하기

`Sync` 마커 트레잇은 `Sync` 가 구현된 타입이 여러 스레드로부터 안전하게 참조 가능함을 나타냅니다. 바꿔 말하면, 만일 `&T` (`T`의 참조자) 가 `Send` 이면, 즉 참조자가 다른 스레드로 안전하게 보내질 수 있다면, `T` 는 `Sync` 합니다. `Send` 와 유사하게, 기초 타입들은 `Sync` 하고, 또한 `Sync` 한 타입들로 전체가 구성된 타입 또한 `Sync` 합니다.

스마트 포인터 `Rc<T>` 는 `Send` 가 아닌 이유와 동일한 이유로 또한 `Sync` 하지도 않습니다. (15장에서 애기했었던) `RefCell<T>` 타입과 연관된 `Cell<T>` 타입의 가족들도 `Sync` 하지 않습니다. `RefCell<T>` 가 런타임에 수행하는 빌림 검사 구현은 스레드-안전하지 않습니다. 스마트 포인터 `Mutex<T>` 는 `Sync` 하고 여러분이 “여러 스레드 사이로 `Mutex<T>` 공유하기” 절에서 본 것처럼 여러 스레드에서 접근을 공유하는데 사용될 수 있습니다.

Send와 Sync를 손수 구현하는 것은 안전하지 않습니다

Send와 Sync 트레이트들로 구성된 타입들이 자동적으로 Send 될 수 있고 Sync하기 때문에, 우리가 이 트레이트들을 손수 구현치 않아도 됩니다. 마커 트레이트으로서, 이들은 심지어 구현할 어떠한 메소드도 없습니다. 이들은 그저 동시성과 관련된 불변성을 강제하는데 유용할 따름입니다.

이 트레이트들을 손수 구현하는 것은 안전하지 않은 러스트 코드 구현을 수반합니다. 19장에서 안전하지 않은 러스트 코드에 대하여 이야기 하겠습니다; 지금으로서 중요한 정보는 Send되고 Sync하지 않은 요소들로 구성된 새로운 동시적 타입을 만드는 것이 안전성 보장을 유지하기 위해 조심스러운 생각을 요구한다는 점입니다. [러스토노미콘](#)이 이러한 보장과 이를 어떻게 유지하는지에 대한 더 많은 정보를 갖고 있습니다.

정리

여기가 이 책에서 동시성에 대해 보게될 마지막은 아닙니다: 20장의 프로젝트에서는 여기서 다룬 작은 예제보다 더 실제와 같은 상황에서 이번 장에서 다룬 개념들을 이용하게 될 것입니다.

일찍이 언급한 것처럼, 러스트가 동시성을 제어하는 방법이 언어의 매우 작은 부분이기 때문에, 많은 동시성 솔루션이 크레이트로 구현됩니다. 이들은 표준 라이브러리보다 더 빠르게 진화하므로, 멀티스레드 상황에서 사용하기 위하여 현재 가장 최신 기술의 크레이트를 온라인으로 검색해보세요.

러스트 표준 라이브러리는 메세지 패싱을 위해 채널을 제공하고, 동시적 맥락에서 사용하기에 안전한 Mutex<T>와 Arc<T> 같은 스마트 포인터 타입들을 제공합니다. 타입 시스템과 빌림 검사기는 이러한 솔루션을 이용하는 코드가 데이터 레이스 혹은 유효하지 않은 참조자로 끝나지 않을 것을 보장합니다. 여러분의 코드가 컴파일된다면, 여러분은 다른 언어에서는 흔한 추적하기 어려운 버그 종류들 없이 여러 스레드 상에서 행복하게 동작할 것이라고 자신감 있게 쉴 수 있습니다. 동시성 프로그래밍은 더 이상 두려워할 개념이 아닙니다: 앞으로 나아가 두려움없이 여러분의 프로그램을 동시적으로 만드세요!

다음으로, 우리는 여러분의 러스트 프로그램이 점차 커짐에 따라서 문제를 모델링하고 솔루션을 구조화하는 자연스러운 방법에 대해 이야기할 것입니다. 더불어서 여러분이 객체 지향 프로그램으로부터 친숙할지도 모를 개념들과 러스트의 관용구가 어떻게 연관되어 있는지 다루겠습니다.

러스트의 객체 지향 프로그래밍 기능들

객체 지향 프로그래밍(OOP)은 프로그램을 모델링하는 방식입니다. 객체는 1960년대 Simula에서 유래됐습니다. 이 객체들은 임의의 객체들이 서로에게 메세지를 전달하는 Alan Kay의 프로그래밍 아키텍처에 영향을 끼쳤습니다. 1967년 그는 *객체 지향 프로그래밍*이라는 용어를 이 아키텍처를 설명하기 위해 사용했습니다. 다수의 정의가 경쟁적으로 OOP이 무엇인지 설명합니다; 그 중 일부는 Rust를 객체 지향이라고 분류하지만 다른 정의는 그렇지 않습니다. 이번 장에서 우리는, 일반적인 객체 지향이 가진 특성들과 어떻게 이런 특성들이 러스트다운 표현들로 번역되었는지 알아볼 것입니다. 그런 후에 객체 지향적 디자인 패턴을 Rust에서 어떻게 구현하는지 보여주고 이를 Rust가 가진 강점을 사용하여 구현했을 경우의 기회비용에 대해 토의합니다.

객체 지향 언어의 특성

객체 지향적인 언어가 반드시 갖춰야 할 기능에 대해 프로그래밍 커뮤니티들은 의견 일치를 보지 못하고 있습니다. 러스트는 OOP도 포함하여 많은 프로그래밍 패러다임에 영향을 받았습니다; 예를 들면, 우리가 13장에서 살펴본 기능인 함수형 프로그래밍에서 온 기능들 말이지요. OOP 언어라면 거의 틀림없이 몇 가지 공통적인 특성을 공유하는데, 객체, 캡슐화 및 상속이 있습니다. 이 특성들이 각각 뜻하는 것과 러스트가 이를 지원하는지에 대해 살펴봅시다.

객체는 데이터와 동작을 담습니다

흔히 The Gang of Four라고도 불리우는 Erich Gamma, Richard Helm, Ralph Johnson, 그리고 John Vlissides (Addison-Wesley Professional, 1994)의 책 *Design Patterns: Elements of Reusable Object-Oriented Software*은 객체 지향 디자인 패턴의 편람입니다. 이 책에서는 OOP를 다음과 같이 정의합니다.

객체-지향 프로그램은 객체로 구성된다. 객체는 데이터 및 이 데이터를 활용하는 프로시저를 묶는다. 이 프로시저들은 보통 메소드 혹은 연산 (*operation*)으로 불린다.

이 정의에 따르면, 러스트는 객체 지향적입니다: 구조체와 열거형은 데이터를 갖고, `impl` 블럭은 그 구조체와 열거형에 대한 메소드를 제공하죠. 설령 메소드를 갖는 구조체와 열거형을 객체라고 호칭하지 않더라도, 그들은 동일한 기능을 수행하며, 이는 Gang of Four의 객체에 대한 정의를 따릅니다.

상세 구현을 은닉하는 캡슐화

일반적으로 OOP와 관련된 또 다른 면은 캡슐화로, 그 의미는 객체를 이용하는 코드에서 그 객체의 상세 구현에 접근할 수 없게 한다는 것입니다. 따라서, 유일하게 객체와 상호작용하는 방법은 이것의 공개 API를 통하는 것입니다; 객체를 사용하는 코드는 직접 객체의 내부에 접근하여 데이터나 동작을 변경해서는 안됩니다. 이는 프로그래머가 객체를 사용하는 코드의 변경없이 이 객체 내부를 변경하거나 리팩토링할 수 있도록 해줍니다.

우리는 7장에서 어떻게 캡슐화를 제어하는지에 대해 논의했습니다: 우리는 `pub` 키워드를 사용하여 어떤 모듈들, 타입들, 함수들, 그리고 메소드들이 공개될 것인가를 결정할 수 있으며, 기본적으로는 모든 것들이 비공개입니다. 예를 들면, 우리는 `i32` 값의 벡터 항목을 가지고 있는 `AveragedCollection` 구조체를 정의할 수 있습니다. 또한 이 구조체는 벡터의 값에 대한 평균값을 담는 항목도 갖는데, 이는 누구든 평균값이 필요한 순간마다 매번 이를 계산할 필요는 없음을 의미합니다. 바꿔 말하면, `AveragedCollection`은 우리를 위해 계산된 평균값을 캐싱할 것입니다. Listing 17-1가 이 `AveragedCollection` 구조체에 대한 정의입니다.

다.

Filename: src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

Listing 17-1: 콜렉션 내의 정수 항목들과 그의 평균을 관리하는 `AveragedCollection` 구조체

구조체가 `pub`으로 표기되면 다른 코드가 이를 사용할 수 있게 되지만, 구조체 안에 존재하는 항목들은 여전히 비공개입니다. 이는 이번 사례에 매우 중요한데, 그 이유는 하나의 값이 리스트에서 더해지거나 제거될 때마다 평균 또한 갱신되는 것을 확신하기 원하기 때문입니다. 우리는 `add`, `remove`, 그리고 `average` 메소드를 구조체에 구현하여 이를 달성하고자 하며, 이는 Listing 17-2과 같습니다:

Filename: src/lib.rs

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            },
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

Listing 17-2: `AveragedCollection`의 공개 메소드 `add`, `remove`, 그리고 `average`

공개 메소드들 `add`, `remove`, 그리고 `average`는 `AveragedCollection`의 인스턴스를 수정하는 유일한 방법입니다. 아이템이 `list`에 `add` 메소드를 통해 추가되거나 `remove` 메소드를 통해 제거될 때, 각각의 호출은 비공개 `update_average` 메소드를 호출하여 `average` 필드를 변경하도록 하는 역할 또한 수행합니다.

우리가 `list`와 `average` 필드를 비공개로 두었으므로 외부 코드가 `list` 필드에 직접 아이템들을 추가하거나 제거할 방법은 없습니다; 그렇지 않으면, `average` 필드는 `list`가 변경될 때 동기화되지 않을지도 모릅니다. `average` 메소드는 `average` 필드의 값을 반환하여, 외부 코드가 `average`를 읽을 수 있도록 하지만, 변경은 안됩니다.

우리가 `AveragedCollection`의 내부 구현을 캡슐화했기 때문에, 차후에 데이터 구조 등을 쉽게 변경할 수 있습니다. 예를 들면, 우리는 `list` 필드에 대해서 `Vec<i32>`가 아닌 `HashSet<i32>`를 사용할 수 있습니다. `add`, `remove` 그리고 `average` 공개 메소드들의 선언이 그대로 유지되는 한, `AveragedCollection`를 사용하는 코드들은 변경될 필요가 없습니다. 대신 우리가 `list`를 공개했다면 꼭 그런 상황이 될 수는 없을 것입니다: `HashSet<i32>`와 `Vec<i32>`는 아이템들을 추가하거나 제거하기 위한 메소드들이 다르므로, 만약 `list`에 직접 접근하여 변경하는 방식의 외부 코드들이 있다면 모두 변경되어야겠죠.

만약 캡슐화가 객체 지향을 염두하는 언어를 위한 필요 요소라면, 러스트는 이를 만족합니다. 코드의 서로 다른 부분들에 대해 `pub`을 사용하거나 사용하지 않는 옵션이 구현 세부 사항의 캡슐화를 가능케 합니다.

타입 시스템과 코드 공유로서의 상속

상속은 어떤 객체가 다른 객체의 정의를 상속받아서, 이를 통해 부모 객체의 데이터와 동작들을 다시 정의하지 않고도 얻을 수 있게 해주는 메커니즘입니다.

만약 객체 지향 언어가 반드시 상속을 제공해야 한다면, 러스트는 그렇지 않은 쪽입니다. 부모 구조체의 필드와 메소드 구현을 상속받는 구조체를 정의할 방법은 없습니다. 하지만 여러분이 상속에 익숙하다면, 우선 이를 사용하고자 하는 이유에 따라 러스트의 다른 솔루션들을 이용할 수 있습니다.

여러분은 두 가지 주요한 이유에 의해 상속을 택합니다. 하나는 코드를 재사용하는 것입니다: 여러분은 어떤 타입의 특정한 행위를 구현할 수 있고, 상속은 당신이 다른 타입을 위해 그 구현을 재사용할 수 있도록 만들어 줍니다. 여러분은 대신 기본 트레이트 메소드의 구현을 이용하여 러스트 코드를 공유할 수 있는데, 이는 Listing 10-14에서 우리가 `Summary` 트레이트에 `summarize` 메소드의 기본 구현을 추가할 때 봤던 것입니다. `Summary` 트레이트를 구현하는 어떤 타입이든, `summarize` 메소드를 별도로 작성하지 않더라도 사용 가능합니다. 이는 어떤 메소드의 구현체를 갖는 부모 클래스와 그를 상속받는 자식 클래스 또한 그 메소드의 해당 구현체를 갖는 것과 유사합니다. 우리는 또한 `Summary` 트레이트를 구현할 때 `summarize`의 기본 구현을 오버라이딩할 수 있고, 이는 자식 클래스가 부모 클래스에서 상속받는 메소드를 오버라이딩하는 것과 유사합니다.

상속을 사용하는 다른 이유는 타입 시스템과 관련있습니다: 자식 타입을 같은 위치에서 부모 타입처럼 사용할 수 있게 하기 위함입니다. 이를 또한 **다형성 (polymorphism)** 이라고도 부르는데, 이는 여러 객체들이 일정한 특성을 공유한다면 이들을 런타임에 서로 바꿔 대입하여 사용할 수 있음을 의미합니다.

다형성

많은 사람들이 다형성을 상속과 동일시 합니다. 하지만 다형성은 다수의 타입들의 데이터에 대해 동작 가능한 코드를 나타내는 더 범용적인 개념입니다. 상속에서는 이런 타입들이 일반적으로 하위클래스에 해당합니다.

러스트는 대신 제네릭을 사용하여 호환 가능한 타입을 추상화하고 트레이트 바운드를 이용하여 해당 타입들이 반드시 제공해야 하는 제약사항을 부과합니다. 이것을 종종 **범주내 매개변수형 다형성 (bounded parametric polymorphism)** 이라고 부릅니다.

최근에는 상속이 많은 프로그래밍 언어에서 프로그래밍 디자인 솔루션으로서의 인기가 떨어지고 있는데 그 이유는 필요한 것보다 더 많은 코드를 공유할 수 있는 위험이 있기 때문입니다. 하위 클래스가 늘 그들의 부모 클래스의 모든 특성을 공유해서는 안되지만 상속한다면 그렇게 됩니다. 이는 프로그램의 유연성을 저하시킬 수 있습니다. 또한, 하위 클래스에서는 타당하지 않거나 적용될 수 없어서 에러를 유발하는 메소드들이 호출될 수 있는 가능성을 만듭니다. 게다가, 어떤 언어들은 하나의 클래스에 대한 상속만을 허용하기 때문에 프로그램 디자인의 유연성을 더욱 제한하게 됩니다.

이런 이유로, 러스트는 다른 방식을 취하여, 상속 대신에 트레이트 객체를 사용합니다. 러스트에서 어떤 식으로 트레이트 객체가 다형성을 가능케 하는지 살펴봅시다.

트레이트 객체를 사용하여 다른 타입 간의 값 허용하기

8장에서는 벡터가 한 번에 하나의 타입만 보관할 수 있다는 제약사항이 있다고 언급했습니다. 우리가 만들었던 Listing 8-10의 작업내역에서는 정수, 부동소수점, 그리고 문자를 보관하기 위한 variant들을 가지고 있는 `SpreadsheetCell` 열거형을 정의했습니다. 이것은 우리가 각 칸마다 다른 타입의 데이터를 저장할 수 있으면서도 여전히 그 칸들의 한 묶음을 대표하는 벡터를 가질 수 있다는 것을 의미했습니다. 이는 우리의 교환가능한 아이템들이 코드를 컴파일할 때 알 수 있는 정해진 몇 개의 타입인 경우 완벽한 해결책입니다.

하지만, 가끔 우리는 우리의 라이브러리 사용자가 특정 상황에서 유효한 타입 묶음을 확장할 수 있도록 하길 원합니다. 우리가 원하는 바를 이룰 수 있는지를 보이기 위해, 우리는 아이템들의 리스트에 걸쳐 각각에 대해 `draw` 메소드를 호출하여 이를 화면에 그리는 그래픽 유저 인터페이스(GUI) 도구는 만들 것입니다 - GUI 도구들에게 있어서는 흔한 방식이죠. 우리가 만들 라이브러리 크레이트는 `gui`라고 호명되고 GUI 라이브러리 구조를 포괄합니다. 이 크레이트는 사용자들이 사용할 수 있는 몇 가지 타입들, `Button`이나 `TextField`들을 포함하게 될 것이구요. 추가로, `gui` 사용자들은 그들 고유의 타입을 만들어 그리고자 할 것입니다: 일례로, 어떤 프로그래머는 `Image`를 추가할지도 모르고 또 다른 누군가는 `SelectBox`를 추가 할지도 모르겠습니다.

우리는 이번 예제에서 총체적인 GUI 라이브러리를 구현하지 않겠지만 어떻게 이 조각들이 맞물려 함께 동작 할 수 있는지 보여주고자 합니다. 라이브러리를 작성하는 시점에서는 다른 프로그래머들이 만들고자 하는 모든 타입들을 알 수 없죠. 하지만 우리가 알 수 있는 것은 `gui` 가 다른 타입들의 다양한 값에 대해 계속해서 추적해야 하고, `draw` 메소드가 이 다양한 값을 각각에 호출되어야 한다는 겁니다. 우리가 `draw` 메소드를 호출했을 때 벌어지는 일에 대해서 정확히 알 필요는 없고, 그저 우리가 호출할 수 있는 해당 메소드를 그 값이 가지고 있음을 알면 됩니다.

상속이 있는 언어를 가지고 이 작업을 하기 위해서는 `draw`라는 이름의 메소드를 갖고 있는 `Component`라는 클래스를 정의할 수도 있습니다. 다른 클래스들, 이를테면 `Button`, `Image`, 그리고 `SelectBox` 같은 것들은 `Component`를 상속받고 따라서 `draw` 메소드를 물려받게 됩니다. 이들은 각각 `draw` 메소드를 오버라이딩하여 그들의 고유 동작을 정의할 수 있으나, 프레임워크는 모든 유형을 마치 `Component`인 것처럼 다룰 수 있고 `draw`를 호출할 수 있습니다. 하지만 러스트가 상속이 없는 관계로, `gui` 라이브러리를 구축하는 다른 방법을 찾아 사용자들이 새로운 타입을 정의하고 확장할 수 있도록 할 필요가 있습니다.

공통된 동작을 위한 트레이트 정의하기

`gui` 가 갖길 원하는 동작을 구현하기 위해, 우리는 `draw`라는 이름의 메소드 하나를 갖는 `Draw`라는 이름의 트레이트를 정의할 것입니다. 그러면 트레이트 객체 (*trait object*) 를 취하는 벡터를 정의할 수 있습니다. 트레이트 객체는 특정 트레이트를 구현한 타입의 인스턴스를 가리킵니다. 우리는 `&` 참조자나 `Box<T>` 스마트 포인터 같은 포인터 종류로 명시함으로서 트레이트 객체를 만들고, 그런 다음 관련된 트레이트를 특정합니다. (우리가 트레이트 객체에 포인터를 사용해야 하는 이유는 19장의 “동적인 크기의 타입과 Sized” 절에서 다룰 겁니다.)

우리는 제네릭 타입이나 구체 타입 대신 트레이트 객체를 사용할 수 있습니다. 트레이트 객체를 사용하는 곳이 어디든, 러스트의 타입 시스템은 컴파일 타임에 해당 문맥 안에 사용된 값이 트레이트 객체의 트레이트를 구현할 것을 보장합니다. 결론적으로, 우리는 컴파일 타임에 모든 가능한 타입을 알 필요가 없습니다.

러스트에서는 구조체와 열거형을 다른 언어의 객체와 구분하기 위해 “객체”라고 부르는 것을 자제한다고 언급했었습니다. 구조체 또는 열거형에서는 구조체 필드의 데이터와 `impl` 블록의 동작이 분리되는 반면, 다른 언어에서는 데이터와 동작이 결합되어 객체로 명명됩니다. 그러나 트레이트 객체들은 데이터와 동작을 결합한다는 의미에서 다른 언어의 객체와 *비슷합니다*. 하지만 트레이트 객체는 트레이트 객체에 데이터를 추가 할 수 없다는 점에서 전통적인 객체들과 다릅니다. 트레이트 객체는 다른 언어들의 객체만큼 범용적으로 유용하지는 않습니다: 그들의 명확한 목적은 공통된 동작들에 걸친 추상화를 가능하도록 하는 것이죠.

Listing 17-3은 `draw`라는 이름의 메소드를 갖는 `Draw`라는 트레이트를 정의하는 방법을 보여줍니다:

Filename: src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

Listing 17-3: `Draw` 트레이트의 정의

이 문법은 10장에 있는 트레이트를 정의하는 방법에서 다뤘으니 익숙하실 겁니다. 다음에 새로운 문법이 등장합니다: Listing 17-4는 `components`라는 벡터를 보유하고 있는 `Screen`이라는 구조체를 정의합니다. `Box<Draw>` 타입의 벡터인데, 이것이 트레이트 객체입니다; 이것은 `Draw` 트레이트를 구현한 `Box`에 담긴 임의의 타입에 대한 대역입니다.

Filename: src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<Draw>>,
}
```

Listing 17-4: `Draw` 트레이트를 구현하는 트레이트 객체들의 벡터 항목 `components`를 소유한 구조체 `Screen`

`Screen` 구조체에서는 Listing 17-5와 같이 각 `components`마다 `draw` 메소드를 호출하는 `run` 메소드를 정의합니다:

Filename: src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Listing 17-5: 각 컴포넌트에 대해 `draw` 메소드를 호출하는 `Screen`의 `run` 메소드

이것은 트레이트 바운드와 함께 제네릭 타입 파라미터를 사용하는 구조체를 정의하는 것과는 다르게 작동합니다. 제네릭 타입 파라미터는 한 번에 하나의 구체 타입으로만 대입될 수 있는 반면, 트레이트 객체를 사용하면 런타임에 여러 구체 타입을 트레이트 객체에 대해 채워넣을 수 있습니다. 예를 들면, Listing 17-6처럼 제네릭 타입과 트레이트 바운드를 사용하여 `Screen` 구조체를 정의할 수도 있을 겁니다.

Filename: src/lib.rs

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where T: Draw {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Listing 17-6: 제네릭과 트레이트 바운드를 사용한 `Screen` 구조체와 `run` 메소드의 대체 구현

이렇게하면 전부 `Button` 타입 혹은 전부 `TextField` 타입인 컴포넌트 리스트를 가지는 `Screen` 인스턴스로 제한됩니다. 동일 유형의 콜렉션만 사용한다면 제네릭과 특성 범위를 사용하는 것이 바람직한데, 왜냐하면 그 정의들은 구체 타입을 사용하기 위해 컴파일 타임에 단형성화 (monomorphize) 되기 때문입니다.

반면에 트레이트 객체를 사용하는 메소드를 이용할때는 하나의 `Screen` 인스턴스가 `Box<Button>` 혹은 `Box<TextField>`도 담을 수 있는 `Vec<T>`를 보유할 수 있습니다. 이것이 어떻게 작동하는지 살펴보고 런타임 성능에 미치는 영향에 대해 설명하겠습니다.

트레이트 구현하기

이제 우리는 `Draw` 트레이트를 구현하는 몇가지 타입을 추가하려고 합니다. 우리는 `Button` 타입을 제공할

것입니다. 다시금 말하지만, 실제 GUI 라이브러리를 구현하는 것은 이 책의 범위를 벗어나므로, 우리는 `draw`에는 별다른 구현을 하지 않을 겁니다. 구현하려는 것을 상상해보자면, `Button` 구조체는 Listing 17-7에서 보시는 바와 같이 `width`, `height` 그리고 `label` 항목들을 가지게 될 것입니다:

Filename: src/lib.rs

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

Listing 17-7: `Draw` 트레이트를 구현하는 `Button` 구조체

`Button`의 `width`, `height` 및 `label` 필드는 다른 컴포넌트와는 차이가 있는데, `TextField` 타입을 예로 들면, 이 필드들에 추가로 `placeholder` 필드를 소유할 겁니다. 우리가 화면에 그리고자 하는 각각의 타입은 `Draw` 트레이트를 구현할테지만 해당 타입을 그리는 방법을 정의하기 위하여 `draw` 메소드 내에 서로 다른 코드를 사용하게 될 것이며, `Button`이 그러한 경우죠 (이 챕터의 범주를 벗어나기 때문에 실질적인 GUI 코드는 없지만요). 예를 들어, `Button` 타입은 추가적인 `impl` 블록에 사용자가 버튼을 클릭했을 때 어떤 일이 벌어질지와 관련된 메소드들을 포함할 수 있습니다. 이런 종류의 메소드는 `TextField`와 같은 타입에는 적용할 수 없죠.

우리의 라이브러리를 사용하는 누군가가 `width`, `height` 및 `options` 필드가 있는 `SelectBox` 구조체를 구현하기로 했다면, Listing 17-8과 같이 `SelectBox` 타입에도 `Draw` 트레이트를 구현합니다:

Filename: src/main.rs

```
extern crate gui;
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
```

Listing 17-8: `gui`를 사용하고 `Draw` 트레이트를 `SelectBox` 구조체에 구현한 또 다른 크레이트

우리 라이브러리의 사용자는 이제 `Screen` 인스턴스를 만들기 위해 `main` 함수를 구현할 수 있습니다. `Screen` 인스턴스에는 `SelectBox`와 `Button`가 트레이트 객체가 되도록 하기 위해 `Box<T>` 안에 넣음으로서 이들을 추가할 수 있습니다. 그러면 `Screen` 인스턴스 상의 `run` 메소드를 호출할 수 있는데, 이는 각 컴포넌트들에 대해 `draw`를 호출할 것입니다. Listing 17-9는 이러한 구현을 보여줍니다:

Filename: src/main.rs

```

use gui::{Screen, Button};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No")
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };
    screen.run();
}

```

Listing 17-9: 트레이트 객체를 사용하여 동일한 트레이트를 구현하는 서로 다른 타입들의 값 저장하기

우리가 라이브러리를 작성할 때는, 누군가 `SelectBox` 타입을 추가할 수도 있다는 것을 알 수 없었지만, 우리의 `Screen` 구현체는 새로운 타입에 대해서도 동작하고 이를 그려낼수 있는데, 그 이유는 `SelectBox`가 `Draw` 타입을 구현했기 때문이고, 이는 `draw` 메소드가 구현되어 있음을 의미합니다.

이러한 개념 —값의 구체적인 타입이 아닌 값이 응답하는 메시지 만을 고려하는 개념—은 동적 타입 언어들의 *오리 타이핑* (*duck typing*) 이런 개념과 유사합니다: 만약 오리처럼 뒤뚱거리고 오리처럼 꽥꽥거리면, 그것은 오리임에 틀림없습니다! Listing 17-5에 나오는 `Screen`에 구현된 `run`을 보면, `run`은 각 컴포넌트가 어떤 구체적 타입인지 알 필요가 없습니다. 이 함수는 컴포넌트가 `Button`의 인스턴스인지 혹은 `SelectBox`의 인스턴스인지 검사하지 않고 그저 각 컴포넌트의 `draw` 메소드를 호출할 뿐입니다. `components` 벡터에 담기는 값의 타입을 `Box<Draw>`로 특정함으로서 우리는 `draw` 메소드를 호출할 수 있는 값을 요구하는 `Screen`을 정의했습니다.

오리 타이핑을 사용하는 코드와 유사한 코드를 작성하기 위해서 트레이트 객체와 러스트의 타입 시스템을 사용하는 것의 장점은 어떤 값이 특정한 메소드를 구현했는지를 검사해야 하거나 혹은 값이 메소드를 구현하지 않았는데 우리가 그걸 어쨌든 호출한다면 생길 수 있는 에러에 대한 걱정을 전혀 할 필요가 없다는 겁니다. 러스트는 트레이트 객체가 요구하는 트레이트를 해당 값이 구현하지 않았다면 컴파일하지 않을 겁니다.

예를 들어, Listing 17-10은 `String`을 컴포넌트로 사용하여 `Screen`을 생성하는 시도를 하면 어떤 일이

벌어지는지 보여줍니다:

Filename: src/main.rs

```
extern crate gui;
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(String::from("Hi")),
        ],
    };

    screen.run();
}
```

Listing 17-10: 트레이트 객체의 트레이트를 구현하지 않은 타입의 사용 시도하기

우리는 아래와 같은 에러를 보게 될 것이며 이유는 `String`이 `Draw` 트레이트를 구현하지 않기 때문입니다:

```
error[E0277]: the trait bound `std::string::String: gui::Draw` is not
satisfied
--> src/main.rs:7:13
 |
7 |         Box::new(String::from("Hi")),
|         ^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait gui::Draw is not
implemented for `std::string::String`
|
= note: required for the cast to the object type `gui::Draw`
```

이 에러는 우리가 넘길 뜻이 없었던 무언가를 `Screen`에게 넘기는 중이고 이를 다른 타입으로 교체해야 하거나, 혹은 우리가 `String`에 대해 `Draw`를 구현하여 `Screen`이 이것에 대해 `draw`를 호출할 수 있도록 해야한다는 것을 알려줍니다.

트레이트 객체는 동적 디스패치를 수행합니다

10장의 “제네릭을 사용한 코드의 성능” 절에서 우리가 제네릭에 트레이트 바운드를 사용했을 때 컴파일러에 의해 이뤄지는 단형성화 프로세스의 실행에 대한 논의를 상기해보세요: 컴파일러는 우리가 제네릭 타입 파라미터를 사용한 각각의 구체 타입을 위한 함수와 메소드의 제네릭 없는 구현체를 생성합니다. 단형성화로부터 야기된 코드는 정적 디스패치 (*static dispatch*)를 수행하는데, 이는 여러분이 호출하고자 하는 메소드가 어떤 것인지 컴파일러가 컴파일 시점에 알고 있는 것입니다. 이는 동적 디스패치 (*dynamic dispatch*)와 반대되는 개념으로, 동적 디스패치는 컴파일러가 여러분이 호출하는 메소드를 컴파일 시에 알 수 없을 경우 수행됩니다.

니다. 동적 디스패치의 경우, 컴파일러는 런타임에 어떤 메소드가 호출되는지 알아내는 코드를 생성합니다.

우리가 트레이트 객체를 사용할 때, 러스트는 동적 디스패치를 이용해야 합니다. 컴파일러는 트레이트 객체를 사용중인 코드와 함께 사용될 수도 있는 모든 타입을 알지 못하기 때문에, 어떤 타입에 구현된 어떤 메소드를 호출할지 알지 못합니다. 대신 런타임에서, 러스트는 트레이트 객체 내에 존재하는 포인터를 사용하여 어떤 메소드가 호출될지 알아냅니다. 정적 디스패치 시에는 일어나지 않는 이러한 탐색이 발생할 때 런타임 비용이 있습니다. 동적 디스패치는 또한 컴파일러가 메소드의 코드를 인라인 (inline)화하는 선택을 막아버리는데, 이것이 결과적으로 몇가지 최적화를 수행하지 못하게 합니다. 하지만, 우리는 추가적인 유연성을 얻어 Listing 17-5와 같은 코드를 작성할 수 있었고, Listing 17-9과 같은 지원이 가능해졌으니, 여기에는 고려할 기회비용이 있다고 하겠습니다.

트레이트 객체에 대하여 객체 안전성이 요구됩니다

여러분은 객체-안전 (*object-safe*) 한 트레이트만 트레이트 객체로 만들 수 있습니다. 트레이트 객체를 안전하게 만드는 모든 속성들을 관찰하는 몇가지 복잡한 규칙이 있지만, 실전에서는 두 가지 규칙만 관련되어 있습니다. 어떤 트레이트 내의 모든 메소드들이 다음과 같은 속성들을 가지고 있다면 해당 트레이트는 객체 안전합니다:

- 반환값의 타입이 `Self`가 아닙니다.
- 제네릭 타입 매개변수가 없습니다.

`Self` 키워드는 우리가 트레이트 혹은 메소드를 구현하고 있는 타입의 별칭입니다. 트레이트 객체가 반드시 객체 안전해야 하는 이유는 일단 여러분이 트레이트 객체를 사용하면, 러스트가 트레이트에 구현된 구체(concrete) 타입을 알 수 없기 때문입니다. 만약 트레이트 메소드가 고정된 `Self` 타입을 반환하는데 트레이트 객체는 `Self`의 정확한 타입을 잊었다면, 메소드가 원래 구체 타입을 사용할 수 있는 방법이 없습니다. 트레이트를 사용할 때 구체 타입 파라미터로 채워지는 제네릭 타입 파라미터도 마찬가지입니다: 그 구체 타입들은 해당 트레이트를 구현하는 타입의 일부가 됩니다. 트레이트 객체를 사용을 통해 해당 타입을 잊게되면, 제네릭 타입 파라미터를 채울 타입을 알 수 없습니다.

메소드가 객체 안전하지 않은 트레이트의 예는 표준 라이브러리의 `Clone` 트레이트입니다. `Clone` 트레이트의 `clone` 메소드에 대한 시그니처는 다음과 같습니다:

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

`String` 타입은 `Clone` 트레이트를 구현하고, `String` 인스턴스에 대하여 `clone` 메소드를 호출하면 우리는 `String`의 인스턴스를 반환받을 수 있습니다. 비슷하게, 우리가 `Vec<T>`의 인스턴스 상의 `clone`을 호출하면, 우리는 `Vec<T>` 인스턴스를 얻을 수 있습니다. `clone` 선언은 `Self`에 어떤 타입이 사용되는지 알 필요가 있는데, 왜냐면 그게 반환 타입이기 때문이죠.

컴파일러는 여러분이 트레이트 객체와 관련하여 객체 안전성 규칙을 위반하는 무언가를 시도하려고 하면 알려 줍니다. 예를 들어, Listing 17-4에서 `Screen` 구조체가 `Draw` 트레이트 대신 `Clone` 트레이트를 구현하는 타입을 보관하도록 아래처럼 구현 시도를 해봅시다:

```
pub struct Screen {
    pub components: Vec<Box<Clone>>,
}
```

우리는 이런 에러를 얻게 될 겁니다:

```
error[E0038]: the trait `std::clone::Clone` cannot be made into an object
--> src/lib.rs:2:5
  |
2 |     pub components: Vec<Box<Clone>>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone` cannot
be
made into an object
  |
= note: the trait cannot require that `Self : Sized`
```

이 에러가 의미하는 바는 이러한 방식으로 이 트레이트를 트레이트 객체로 사용할 수 없다는 겁니다. 혹시 객체 안전에 대해 보다 자세하게 알고 싶으시면 [Rust RFC 255](#)를 참고하세요.

객체 지향 디자인 패턴 구현하기

*상태 패턴 (state pattern)*은 객체 지향 디자인 패턴입니다. 이 패턴의 핵심은 어떤 값이 상태 객체들의 집합으로 표현되는 일종의 내부 상태를 가지며, 이 값의 동작은 내부 상태에 기반하여 바뀐다는 것입니다. 상태 객체들은 기능을 공유합니다: 당연히 러스트에서는 객체와 상속보다는 구조체와 트레이트를 사용합니다. 각 상태 객체는 자신의 동작 및 다른 상태로 변경되어야 할 때의 제어에 대한 책임이 있습니다. 상태 객체를 보유한 값은 상태들의 서로 다른 행동 혹은 상태 간의 전이가 이뤄지는 때에 대해 아무것도 모릅니다.

상태 패턴을 사용한다는 것은 프로그램의 사업적 요구사항들이 변경될 때, 상태를 보유한 값의 코드 혹은 그 값을 사용하는 코드는 변경될 필요가 없음을 의미합니다. 단지 우리는 상태 객체 중에 하나의 내부 코드를 갱신하여 그 규칙을 바꾸거나 혹은 상태 객체를 더 추가할 필요가 있을 따름입니다. 상태 디자인 패턴 예제를 살펴보고 이를 러스트에서 사용하는 방법에 대해 알아봅시다.

우리는 점진적인 방식으로 블로그에 게시물을 올리는 작업 흐름을 구현하려고 합니다. 블로그의 최종적인 기능은 다음과 같을 것입니다:

1. 블로그 게시물은 빈 초안으로 시작합니다.
2. 초안이 완료되면 게시물의 검토가 요청됩니다.
3. 게시물이 승인되면 게시됩니다.
4. 오직 게시된 블로그 게시물만이 출력될 내용물을 반환하므로, 승인되지 않은 게시물은 실수로라도 게시될 수 없습니다.

이 외에 게시물에 시도되는 어떠한 변경사항도 영향을 미쳐서는 안됩니다. 예를 들어, 만약 리뷰를 요청하기도 전에 블로그 게시물 초안을 승인하려는 시도를 했다면, 그 게시물은 비공개 초안인 상태로 유지되야 합니다.

Listing 17-11은 코드의 형태로 이 작업 흐름을 보여줍니다: 이는 우리가 구현할 라이브러리 크레이트 **blog**의 API를 사용하는 예제입니다. 아직 컴파일되지 않는 이유는 **blog** 크레이트를 아직 구현하지 않았기 때문이죠.

Filename: src/main.rs

```

extern crate blog;
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}

```

Listing 17-11: `blog` 크레이트가 갖길 원하는 요구 동작들을 보여주는 코드

우리는 사용자가 새로운 블로그 게시물의 초안을 `Post::new`를 통해 만들 수 있도록 허용하고 싶습니다. 이후에는 블로그 게시물에 초안인 상태로 글을 추가할 수 있도록 하고자 합니다. 만약 우리가 게시물의 내용을 승인 전에 즉시 얻어오는 시도를 하면, 해당 게시물이 아직 초안이기 때문에 아무 일도 일어나지 않아야 합니다. 이를 보여주는 용도로 코드 내에 `assert_eq!`를 추가했습니다. 이를 위한 훌륭한 단위 테스트는 블로그 게시물 초안이 `content` 메소드에 대해 빈 문자열을 반환하는 것이겠지만, 우리는 이 예제를 위한 테스트를 구현하지 않을 겁니다.

다음으로, 게시물의 리뷰를 요청하는 것을 허용하고자 하고, 리뷰를 기다리는 동안에는 `content` 가 빈 문자열을 반환하도록 하고 싶습니다. 게시물이 허가를 받은 시점에는 게시가 되어야 하는데, 이는 `content`의 호출되었을 때 게시물의 글이 반환될 것임을 뜻합니다.

이 크레이트로부터 우리가 상호작용 하고 있는 유일한 타입이 `Post` 타입임을 주목하세요. 이 타입은 상태 패턴을 사용하며 게시물이 가질 수 있는 초안, 리뷰 대기중, 게시됨을 나타내는 세가지 상태 중 하나가 될 값을 보유할 것입니다. 어떤 상태에서 다른 상태로 변경되는 것은 `Post` 타입 내에서 내부적으로 관리됩니다. 이 상태들은 우리 라이브러리의 사용자가 `Post` 인스턴스 상에서 호출하는 메소드에 응답하여 변경되나, 상태 변화를 직접 관리할 필요는 없습니다. 또한, 사용자는 리뷰 전에 게시물이 게시되는 것 같은 상태와 관련된 실수를 할 수 없습니다.

Post를 정의하고 초안 상태의 새 인스턴스 생성하기

라이브러리의 구현을 시작해보죠! 우리는 어떤 내용물을 담고 있는 공개된 `Post` 구조체가 필요하다는 것을 아니까, Listing 17-12에서 보시는 바와 같이 `Post`의 인스턴스를 만들기 위해서 구조체 및 관련된 공개 `new` 함수 대한 정의로 시작할 것입니다. 비공개 `State` 트레이트 또한 만들겁니다. 그 다음 `Post`는 비공개

필드 `state`에 `Option`으로 감싸진 `Box<State>` 형태의 트레이트 객체를 보유할 겁니다. 곧 `Option`이 왜 필요한지 보게 될 겁니다.

Filename: src/lib.rs

```
pub struct Post {
    state: Option<Box<State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

Listing 17-12: `Post` 구조체, `Post` 인스턴스를 만드는 `new` 함수, `State` 트레이트와 `Draft` 구조체의 정의

`State` 트레이트는 게시물의 상태 변화에 따라 달라지는 동작을 정의하고, `Draft`, `PendingReview`, 그리고 `Published` 상태는 모두 `State` 트레이트를 구현하게 됩니다. 지금은 트레이트가 아무런 메소드도 갖지 않고, 우리는 그저 `Draft` 상태의 구현부터 시작하려고 하는데, 왜냐면 그게 게시물이 최초로 갖는 상태이거든요.

우리가 새로운 `Post`를 생성할 때, 이것의 `state` 필드에 `Box`를 보유한 `Some` 값을 설정합니다. 이 `Box`는 `Draft` 구조체의 새 인스턴스를 가리킵니다. 이는 우리가 언제 `Post`의 새 인스턴스를 생성하는지, 초안으로 시작하는 것을 보장합니다. `Post`의 `state` 필드가 비공개이기 때문에, 다른 상태로 `Post`를 생성할 방법은 없습니다! `Post::new` 함수에서는 `content` 필드를 새로운, 빈 `String`로 설정합니다.

게시물 콘텐츠의 글을 저장하기

Listing 17-11은 우리가 `add_text`로 명명된 메소드를 호출하고 여기에 `&str`을 넘겨 블로그 게시물의 콘텐츠 글에 추가할 수 있도록 하길 원한다는 것을 보여줍니다. 우리는 `content` 필드를 `pub`으로 노출시키는 것보다는 메소드로서 이를 구현할 겁니다. 이는 우리가 `content` 필드의 데이터를 읽는 방식을 제어할

수 있는 메소드를 나중에 구현할 수 있음을 뜻합니다. `add_text` 메소드는 매우 직관적이니까, Listing 17-13에서 `impl Post` 블록에 구현을 추가해봅시다:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Listing 17-13: `content`에 글을 추가하기 위한 `add_text` 메소드 구현하기

`add_text` 메소드는 가변 참조자 `self`를 취하는데, 그 이유는 우리가 `add_text`를 호출하고 있는 해당 `Post` 인스턴스를 변경하게 되기 때문입니다. 그런 다음 우리는 `content`의 `String` 상에서 `push_str`을 호출하고 `text`를 인자로 전달해 저장된 `content`에 추가합니다. 이 동작은 게시물의 상태와 무관하게 이뤄지므로, 상태 패턴의 일부가 아닙니다. `add_text` 메소드는 `state` 필드와 전혀 상호작용을 하지 않지만, 우리가 지원하고자 하는 동작 요소입니다.

초안 게시물의 내용이 비어있음을 보장하기

우리가 `add_text`를 호출하고 우리의 게시물에 어떤 콘텐츠를 추가한 이후일지라도, 여전히 `content` 메소드가 빈 스트링 슬라이스를 반환하길 원하는데, 그 이유는 Listing 17-11의 8번째 줄처럼 게시물이 여전히 초안 상태이기 때문입니다. 당장은 이 요건을 만족할 가장 단순한 것으로 `content` 메소드를 구현해놓으려고 합니다: 언제나 빈 스트링 슬라이스를 반환하는 것으로요. 우리가 게시물의 상태를 변경하여 이것이 게시될 수 있도록 하는 기능을 구현하게 되면 그 후에 이 메소드를 변경하겠습니다. 그 때까지 게시물은 오직 초안 상태로만 존재하기에 게시물 컨텐츠는 항상 비어 있어야 합니다. Listing 17-14는 이 껍데기 구현을 보여줍니다:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        ""
    }
}
```

Listing 17-14: 항상 비어있는 스트링 슬라이스를 반환하는 `Post`의 `content` 메소드에 대한 껍데기 구현

content 메소드를 추가함으로서, Listing 17-11의 8번째 줄까지는 의도한대로 작동됩니다.

게시물에 대한 리뷰 요청이 그의 상태를 변경합니다

다음으로, 우리는 게시물의 리뷰를 요청하는 기능을 만들어야 하는데, 이는 게시물의 상태를 **Draft**에서 **PendingReview**로 변경해야 합니다. Listing 17-15는 이에 관련된 코드입니다:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<State> {
        self
    }
}
```

Listing 17-15: **Post**와 **State** 트레이잇에 **request_review** 메소드를 구현하기

우리는 **Post**에게 **self**에 대한 가변 참조자를 취하려는 **request_review**란 이름의 공개 메소드를 주어줬습니다. 그 다음 우리가 **Post**의 현재 상태 상에서 내부 메소드 **request_review**를 호출하고, 이 두 번째 **request_review** 메소드는 현재의 상태를 소비하고 새로운 상태를 반환합니다.

우리는 **State** 트레이잇에 **request_review** 메소드를 추가했습니다; 트레이잇을 구현하는 모든 타입은 이제

`request_review` 메소드를 구현할 필요가 있을 것입니다. 주목할 점은 메소드의 첫 인자를 `self`, `&self`, 나 `&mut self`를 취하기보다 `self: Box<Self>`를 취한다는 겁니다. 이 문법은 메소드가 오직 그 타입을 보유한 `Box` 상에서 호출될 경우에만 유효함을 뜻합니다. 해당 문법은 `Box<Self>`의 소유권을 가져가는데, `Post`의 예전 상태를 무효화하여 새 상태로 변화하게 해줍니다.

이전 상태를 소비하기 위해서 `request_review` 메소드는 상태 값의 소유권을 취할 필요가 있습니다. 이것 이 `Post`의 `state` 필드 내 `Option`이 들어온 까닭입니다: 우리는 `take` 메소드를 호출하여 `state` 필드 밖으로 `Some` 값을 빼내고 그 자리에 `None`을 남기는데, 왜냐면 러스트는 구조체 내에 값이 없는 필드를 허용하지 않기 때문입니다. 이는 우리가 `state` 값을 빌리기 보다는 게시물 밖으로 이동시키도록 만듭니다. 이후 우리는 게시물의 `state` 값을 이런 연산의 결과물로 설정하려고 합니다.

우리는 `state` 값의 소유권을 얻기 위해서 `self.state = self.state.request_review();`처럼 직접 설정하는 것 보다는 `state`를 임시로 `None`으로 설정할 필요가 있습니다. 이는 `Post`가 예전 `state` 값을 새 상태로 변경시킨 뒤에는 사용할 수 없음을 보장합니다.

`Draft`의 `request_review` 메소드는 새 박스로 포장된 `PendingReview` 구조체의 새 인스턴스를 반환해야 하며, 이는 게시물이 리뷰를 기다리고 있다는 상태를 표현합니다. `PendingReview` 구조체 또한 `request_review` 메소드를 구현하지만 어떤 변경도 하지 않습니다. 그보다 이 구조체는 자기 자신을 반환하는데, 그 이유는 이미 `PendingReview` 상태인 게시물에 대한 리뷰를 요청할 때는 `PendingReview` 상태를 그대로 유지해야 하기 때문입니다.

이제 우리는 상태 패턴의 장점을 알아보기 시작할 수 있습니다: `Post`의 `request_review` 메소드는 그것의 `state`가 무엇이든 상관없이 동일합니다. 각 상태는 그 자신의 규칙에 따라 맡은 책임을 다할 것입니다.

우리는 `Post`의 `content` 메소드가 여전히 빈 스트링 슬라이스를 반환하도록 그대로 놔두려고 합니다. 현재 우리는 `Draft` 상태에 있는 `Post` 뿐만 아니라 `PendingReview` 상태에 있는 `Post`를 소유할 수 있습니다만, `PendingReview` 상태에서도 동일한 동작을 원합니다. Listing 17-11은 이제 11번째 줄까지 동작합니다!

content의 동작을 변경하는 approve 메소드 추가하기

`approve` 메소드는 `request_review` 메소드와 유사할겁니다: 이것은 Listing 17-16과 같이 해당 상태가 허가되었을때 가져야 하는 값으로 `state`를 설정할 것입니다:

Filename: src/lib.rs

```

impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<State>;
    fn approve(self: Box<Self>) -> Box<State>;
}

struct Draft {}

impl State for Draft {
    // --snip--
    fn approve(self: Box<Self>) -> Box<State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<State> {
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<State> {
        self
    }
}

```

Listing 17-16: `Post` 및 `State` 트레이트에 대한 `approve` 메소드 구현하기

우리는 `approve` 메소드를 `State` 트레이트에 추가했고 `State`를 구현하는 새 구조체 `Published` 상태도 추가했습니다.

`request_review`와 유사하게, 우리가 `Draft`의 `approve` 메소드를 호출하면, 이는 별 효과가 없는데 이유는 이 때 반환되는 것이 `self`이기 때문이죠. 우리가 `PendingReview` 상에서 `approve`를 호출하면, 박스로 포장된 `Published` 구조체의 새 인스턴스가 반환됩니다. `Published` 구조체는 `State` 트레이잇을 구현하고, `request_review`와 `approve` 메소드 양 쪽 모두에서 자기 자신을 반환하는데, 이러한 경우에는 그 게시물이 `Published` 상태를 유지해야 하기 때문입니다.

이제 우리가 해야 할 일은 `Post`의 `content` 메소드를 갱신하는 겁니다: Listing 17-17처럼 만일 상태가 `Published`이면, 우리는 게시물의 `content` 필드의 값을 반환하길 원합니다; 그렇지 않다면, 우리는 빈 스트링 슬라이스를 반환하고자 합니다.

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(&self)
    }
    // --snip--
}
```

Listing 17-17: `State`의 `content` 메소드를 위임하기 위한 `Post`의 메소드 갱신하기

목표하는 바가 `State`를 구현하는 구조체들 내에서 이 모든 규칙을 유지하는 것이기 때문에, 우리는 `state`의 값에 `content` 메소드를 호출하면서 게시물 인스턴스 (여기서는 `self`)를 인자로 넘깁니다. 그러면 우리는 `state`의 `content` 메소드를 사용하여 반환되는 값을 받게 됩니다.

우리는 `Option`의 `as_ref` 메소드를 호출하는데 `Option` 내의 값에 대한 소유권을 가져오기 보다는 그에 대한 참조자를 원하기 때문입니다. `state`는 `Option<Box<State>>`이므로, 우리가 `as_ref`를 호출하면 `Option<&Box<State>>`가 반환됩니다. `as_ref`를 호출하지 않는다면, 우리는 해당 함수 파라미터의 빌려온 `&self`로부터 `state`를 이동시킬 수 없기 때문에 에러를 얻게 될 겁니다.

그런 다음 우리는 `unwrap` 메소드를 호출하고 이것이 패닉을 결코 발생시키지 않을 것을 알고 있는데, 그 이유는 `Post`의 메소드들은 이들이 실행 완료됐을 때 `state`가 항상 `Some` 값을 담고 있을 것을 보장하기 때문입니다. 이는 우리가 9장의 “여러분이 컴파일러보다 더 많은 정보를 가진 경우” 절에서 말했던 경우 중 하나이며, 심지어 컴파일러가 그런 경우를 이해할 수 없더라도 `None` 값이 결코 가능하지 못한 경우입니다.

이 지점에서 우리가 `&Box<State>`의 `content`를 호출할 때, 역참조 강제는 `&`와 `Box`에 영향을 줘서 `content` 메소드가 궁극적으로 `State` 트레이잇을 구현하는 타입 상에서 호출되도록 합니다. 이는 우리가 `State` 트레이잇 정의에 `content`를 추가할 필요가 있음을 의미하고, 이곳이 Listing 17-18처럼 우리가 가진 상태에 따라 어떤 컨텐츠를 반환할지에 대한 로직을 삽입할 위치입니다:

Filename: src/lib.rs

```

trait State {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ""
    }
}

// --snip--
struct Published {}

impl State for Published {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        &post.content
    }
}

```

Listing 17-18: `State` 트레이잇에 `content` 메소드 추가하기

우리는 빈 스트링 슬라이스를 반환하는 `content` 메소드의 기본 구현을 추가합니다. 이는 우리가 `Draft` 와 `PendingReview` 구조체에 대한 `content`를 구현할 필요가 없다는 뜻입니다. `Published` 구조체는 `content` 메소드를 오버라이딩하고 `post.content`의 값을 반환할 겁니다.

10장에서 우리가 토의했던 대로 이 메소드에 대한 라이프타임 명시가 필요함을 주의하세요. 우리는 `post`에 대한 참조자를 인자로 취하고 있고 해당 `post`의 일부에 대한 참조를 반환하는 중이므로, 반환되는 참조자의 라이프타임은 `post` 인자의 라이프타임과 관련이 있습니다.

그리고 이제 끝났습니다—Listing 17-11의 모든 코드가 이제 작동합니다! 우리는 블로그 게시물의 작업 흐름을 상태 패턴을 통해 구현해냈습니다. 규칙과 관련있는 로직들은 `Post`에 흩어져있지 않고 상태 객체에 존재합니다.

상태 패턴의 기회비용

우리는 게시물이 각 상태에 대해 가져야 하는 서로 다른 종류의 동작을 캡슐화하기 위해서 러스트로 객체 지향 상태 패턴을 충분히 구현할 수 있음을 보여줬습니다. `Post`의 메소드는 이런 다양한 동작에 대해서 알지 못합니다. 우리가 코드를 구조화한 방식에 따라, 게시된 게시물이 취할 수 있는 서로 다른 방법을 알기 위해서는 단 한 곳만 보면 됩니다: `Published` 구조체의 `State` 트레이잇에 구현된 내용말이죠.

만약 우리가 상태 패턴을 사용하지 않고 다른 방식으로 구현한다면, `Post` 혹은 `main` 코드에서 `match` 표현식을 대신 사용하여 게시물의 상태를 검사하고 이에 따라 해야 할 행동을 변경해야 할지도 모르겠습니다. 이는 우리가 게시된 상태의 게시물의 모든 결과들에 대해 이해하기 위해서 여러 곳을 봐야 한다는 것을 뜻합니다! 이는 우리가 상태를 추가하는 것을 더 늘게 할 뿐입니다: 각각의 `match` 표현식은 또 다른 갈래를 필요

로 할테지요.

상태 패턴을 이용하면 `Post` 메소드들과 `Post`를 사용하는 곳에서는 `match` 표현식을 사용할 필요가 없고, 새로운 상태를 추가하려면, 그저 새로운 구조체와 구조체에 대한 트레이트 메소드들을 구현하면 됩니다.

상태 패턴을 사용하면 추가 기능을 구현하기 쉽습니다. 상태 패턴을 사용하는 코드를 유지하는 것의 단순성을 체험해보려면, 다음 제안 중 몇가지를 시도해보세요:

- `reject` 메소드를 추가하여 게시물의 상태를 `PendingReview`에서 `Draft`로 변경하기
- `Published`로 상태 변경이 가능해지기 전에 `approve`가 두 번 호출되도록 요구하기
- 게시물이 `Draft` 상태일 때는 사용자들에게 글 내용의 추가만 허용하기. 힌트: 상태 객체가 내용에 관한 변경에는 역할을 하지만 `Post`를 수정하기 위한 역할은 하지 않게 하기

상태 패턴의 단점 중에 하나는, 상태가 상태 간의 전환을 구현하기 때문에, 몇몇 상태들이 서로 둘이게 된다는 점입니다. 만약 우리가 `PendingReview`와 `Published` 사이에 `Scheduled`과 같은 상태를 추가하면, `PendingReview`에서 `Scheduled`로 전환되도록 코드를 변경하여야 합니다. 새로운 상태의 추가와 함께 `PendingReview`가 변경될 필요가 없었다면 좀 더 작은 작업이 되겠지만, 이는 다른 디자인 패턴으로의 전환을 의미할 겁니다.

또다른 단점은 우리가 몇몇 로직을 중복시킨다는 겁니다. 중복의 일부를 제거하려면, 우리는 `State` 트레이트의 `request_review`와 `approve` 메소드가 `self`를 반환하도록 기본 구현을 만드는 시도를 할 수도 있습니다; 하지만, 이는 객체 안전성을 위배할 수 있는데, 그 이유는 해당 트레이트가 어떤 구체적인 `self`가 될 것인지 알지 못하기 때문입니다. 우리는 `State`를 트레이트 객체로서 사용하길 원하기에, 이것의 메소드들은 객체 안전성을 지킬 필요가 있습니다.

`Post`에 메소드 `request_review`와 `approve`처럼 유사한 구현들도 그 밖의 중복에 포함됩니다. 두 메소드 모두 `Option`의 `state` 필드 값에 대해 동일한 메소드의 구현을 대행하며, 그 결과값을 `state` 필드의 새 값으로 설정합니다. 이 패턴을 따르는 `Post`의 메소드를 많이 갖게 되면, 이러한 반복을 제거하기 위해 매크로의 정의를 고려할 수도 있습니다 (매크로에 대한 자세한 내용은 부록 D를 참조하세요).

객체 지향 언어에서 정의하는 것과 동일하게 상태 패턴을 구현함으로써, 우리가 사용할 수 있는 러스트의 강점을 모두 이용하지 못하고 있습니다. 유효하지 않은 상태 및 전환이 컴파일 타임 에러가 될 수 있도록 하기 위해 우리가 할 수 있는 `blog` crate에 대한 변경 몇가지를 살펴봅시다.

상태와 동작을 타입처럼 인코딩하기

우리는 다른 기회 비용을 얻기 위해 상태 패턴을 재고하는 방법을 보여줄 것입니다. 오히려 상태와 전환을 완전히 캡슐화하여 외부의 코드들이 이를 알지 못하는 것보다는, 상태를 다른 타입들로 인코딩하려고 합니다. 결론적으로, 러스트의 타입 검사 시스템은 게시된 게시물만 허용되는 곳에서 게시물 초안을 사용하려는 시도에 대해 컴파일 에러를 발생시킴으로서 방지할 것입니다.

Listing 17-11의 `main` 첫 부분을 주의깊게 살펴봅시다:

Filename: src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

우리는 여전히 `Post::new`를 사용하여 초안 상태의 새 게시물을 생성할 수 있도록 하며 게시물의 내용에 새 글을 추가할 수 있는 기능을 허용합니다. 하지만 빈 문자열을 반환하는 초안 게시물의 `content` 메소드 대신, 초안 게시물이 `content` 메소드를 갖지 않도록 만들려고 합니다. 이렇게 하면, 우리가 초안 게시물의 내용을 얻는 시도를 할 경우, 해당 메소드가 존재하지 않는다는 컴파일 에러를 얻게 될 겁니다. 결과적으로, 우리가 의도치않게 제작 중인 초안 게시물의 내용을 얻게 되는 일이 불가능하게 되는데, 왜냐면 그런 코드는 아예 컴파일이 되지 않으니까요. Listing 17-19에서는 `Post` 구조체와 `DraftPost` 구조체의 정의와 각각의 메소드를 보여줍니다:

Filename: src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Listing 17-19: `content` 메소드가 있는 `Post` 와 `content` 메소드가 없는 `DraftPost`

`Post` 와 `DraftPost` 구조체 모두 비공개인 `content` 필드를 가지고 블로그 게시물의 글을 보관합니다. 이 구조체들이 더 이상 `state` 필드를 갖지 않는 이유는 상태의 인코딩을 구조체의 타입으로 이동시켰기 때문입니다. `Post` 구조체는 공개된 게시물을 나타낼 것이고, 그의 `content` 메소드는 `content`를 반환할 겁니다.

우리는 여전히 `Post::new` 함수를 유지하지만, `Post`의 인스턴스를 반환하는 대신 `DraftPost`를 반환합니다. `content`는 비공개이고 `Post`를 반환할 어떤 함수도 존재하지 않기 때문에, 당장 `Post`의 인스턴스를 생성하는 것은 불가능합니다.

`DraftPost` 구조체는 `add_text` 메소드를 가지고 있으므로, 우리는 전처럼 `content`에 글을 추가할 수 있지만, `DraftPost`는 정의된 `content` 메소드가 없음을 주의하세요! 따라서 이제 프로그램은 모든 게시물이 초안 게시물로 시작되고, 초안 게시물들은 그들의 내용을 출력할 능력이 없음을 보장합니다. 이 제약사항을 벗어나는 어떤 시도라도 컴파일러 에러로 끝나게 될 것입니다.

다른 타입으로 변환하는 것처럼 전환 구현하기

그러면 어떻게 게시된 게시물을 얻는 걸까요? 우리는 초안 게시물이 공개되기 전에 리뷰와 승인을 받도록 강제하고 싶습니다. 리뷰를 기다리는 상태인 게시물은 여전히 어떤 내용도 보여줘서는 안되구요. Listing 17-20처럼 새 구조체 `PendingReviewPost`를 추가하고, `DraftPost`에 `PendingReviewPost`를 반환하는 `request_review` 메소드를 정의하고, `PendingReviewPost`에 `Post`를 반환하는 `approve` 메소드를 정의하여 위의 제약사항들을 구현해봅시다:

Filename: src/lib.rs

```

impl DraftPost {
    // --snip--

    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}

```

Listing 17-20: `DraftPost`의 `request_review`를 호출하여 생성되는 `PendingReviewPost` 및 `PendingReviewPost`를 게시된 `Post`로 전환하는 `approve` 메소드

`request_review`와 `approve` 메소드는 `self`의 소유권을 취하므로, `DraftPost`와 `PendingReviewPost`의 인스턴스를 소비하여 이들을 각각 `PendingReviewPost`와 게시된 `Post`로 변환시킵니다. 이 방식으로 우리가 `request_review`를 호출한 후 등등에는 `DraftPost` 인스턴스를 질질 끌지 않게 될 겁니다. `PendingReviewPost` 구조체는 `content` 메소드의 정의를 갖지 않기 때문에, 그의 내용물을 읽으려는 시도는 `DraftPost`와 마찬가지로 컴파일 에러를 발생시킵니다. `content` 메소드를 정의하고 있는 게시된 `Post` 인스턴스를 얻을 수 있는 유일한 방법은 `PendingReviewPost`의 `approve` 메소드를 호출하는 것이고, `PendingReviewPost`를 얻을 수 있는 유일한 방법은 `DraftPost`의 `request_review`를 호출하는 것이기에, 우리는 이제 블로그 게시물의 작업 흐름을 타입 시스템으로 인코딩했습니다.

그뿐 아니라 우리는 `main`에 약간의 변화를 줘야 합니다. `request_review`와 `approve` 메소드는 호출되고 있는 구조체를 변경하기보다는 새 인스턴스를 반환하기 때문에, 우리는 반환되는 인스턴스를 보관하기 위해 더 많은 `let post =`를 추가할 필요가 있습니다. 또한 초안과 리뷰 중인 게시물의 내용이 빈 문자열이라고 단언할 수도 없고, 단언할 필요도 없습니다: 이 상태들에서 게시물이 내용을 사용 시도하는 코드는 더이상 컴파일되지 않습니다. Listing 17-12에 간단히 간단한 `main` 코드가 있습니다:

Filename: src/main.rs

```

extern crate blog;
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}

```

Listing 17-21: 블로그 게시물 작업 흐름의 새 구현을 사용하기 위한 `main` 수정

`main`에서 `post`의 다시 대입하기 위해 필요한 이 변경사항은 즉 이 구현이 더 이상 객체 지향 상태 패턴을 잘 따르지 않는다는 것을 의미합니다: 상태간의 변환은 더 이상 `Post`의 구현체 내에 모두 캡슐화되지 않습니다. 하지만, 우리가 얻은 것은 타입 시스템과 컴파일 타임에 일어나는 타입 검사 때문에 유효하지 않은 상태가 이제 불가능해진다는 것입니다! 이는 게시되지 않은 게시물의 내용이 보여진다거나 하는 특정 버그들이 제품화가 되기 전에 발견될 것임을 보장합니다.

여러분이 이 버전의 코드 디자인에 대해 어떻게 생각하는지 알아보려면 이번 절의 시작점에서 우리가 언급했던 추가적인 요구사항으로서 제안된 작업을 Listing 17-20 이후의 `blog` 크레이트 상에서 시도해보세요. 몇가지 작업은 이번 디자인에서 이미 완료됐음을 알려드립니다.

우리는 러스트가 객체 지향 디자인 패턴을 사용할 수 있을지라도, 상태를 타입 시스템으로 인코딩하는 다른 패턴 또한 러스트 내에서 가능함을 봤습니다. 이 패턴들은 서로 다른 기회비용을 갖고 있습니다. 여러분이 객체 지향 패턴에 매우 익숙할런지 몰라도, 몇몇 버그를 컴파일 타임에 방지하는 등 러스트의 기능들이 제공할 수 있는 이점들을 이용하기 위해서는 문제를 다시 생각해보세요. 객체 지향 패턴은 러스트 내에서 제공하는 소유권 같이 객체 지향 언어에서는 갖지 못한 기능들 때문에 늘 최고의 해결책이 되지는 못합니다.

정리

이 장을 읽은 후 러스트가 객체 지향 언어라고 생각하든 아니든, 이제 여러분은 트레잇 객체를 사용하여 몇 가지 객체 지향 기능을 러스트 내에서 사용할 수 있다는 것을 알게 되었습니다. 동적 디스패치는 약간의 실행 성능과 맞바꿔 여러분의 코드에 유연성을 줄 수 있습니다. 여러분은 이 유연성을 사용하여 여러분의 코드 관리를 도와줄 수 있는 객체 지향 패턴을 구현할 수 있습니다. 러스트는 또한 소유권과 같은 객체 지향 언어들에는 없는 기능들도 갖고 있습니다. 객체 지향 패턴이 항상 러스트의 강점을 이용하는 최고의 방법은 아니겠지만, 선택 가능한 옵션입니다.

다음으로, 우리는 패턴을 살펴볼 것인데, 이는 높은 유연성을 가능케하는 러스트의 또 다른 기능 중 하나입니다. 이 책 전체에 걸쳐 간단히 살펴보긴 했지만 아직 패턴들의 모든 능력을 살펴본건 아닙니다. 어서 가즈아!

패턴과 매칭

패턴은 단순하거나 복잡한 타입의 구조에 값들을 비교하기 위한 러스트의 특별한 문법입니다. 패턴을 **match** 표현 및 다른 구문들과 함께 사용하면 프로그램 흐름을 더 많이 제어할 수 있습니다. 패턴은 다음의 조합으로 이루어집니다:

- 리터럴 값(Literals)
- 분해한 배열(Array), 열거형(Enum), 구조체(Struct), 튜플(Tuple)
- 변수(Variiable)
- 와일드카드(Wildcard)
- 임시 값(Placeholders)

이들은 프로그램이 처리 할 자료들의 형태를 나타냅니다. 자료들을 이 구조들에 대응시키고, 대응 시킨 자료를 값들과 비교하여 특정 구간의 코드가 실행 될 수 있는지 판단할 수 있게 됩니다.

패턴을 이용하기 위해선 그 패턴을 어떠한 값에 비교해야 합니다. 만일 패턴이 값에 대응된다면 그 값에 해당되는 부분을 코드상에서 이용하게 됩니다. 6장의 **match** 표현 예제, 동전 계수기 예제를 떠올려 봅시다. 값이 패턴의 형태에 들어맞는다면 패턴이 정한 이름들로 값을 이용할 수 있습니다. 형태가 다르다면 해당 패턴과 관련된 코드는 실행 되지 않았습니다.

이번 장은 패턴과 관련된 모든 것의 레퍼런스입니다. 어느 곳에서 패턴을 사용할 수 있는지와, 반증 가능 패턴(*refutable patterns*)과 반증 불가 패턴(*irrefutable patterns*)의 차이, 여러분이 접해볼 수 있는 다양한 종류의 패턴 문법에 대해서 다룹니다. 이번장을 마치고 나면 패턴을 이용해 다양한 개념을 명확하게 표현하는 방법에 대해 알게 될 것입니다.

패턴이 사용될 수 있는 모든 곳

패턴은 러스트 코드 곳곳에 튀어나옵니다. 아마 모르는 사이 이미 아주 많이 쓰고 있었을 겁니다! 이번 절에선 패턴을 사용할 수 있는 모든 코드상의 장소에 대한 레퍼런스를 제공합니다.

match 갈래

6장에서도 다루었듯, 패턴은 `match` 표현의 각 갈래에 사용됩니다. `match` 표현은 `match` 키워드, 대응 시킬 값, 갈래들로 이루어지고, 각 갈래는 갈래를 나타내는 패턴, 해당 패턴에 값이 대응 될 때 실행할 표현으로 구성됩니다. 결국 다음과 같은 구조를 가집니다:

```
match 값 {
    패턴 => 표현,
    패턴 => 표현,
    패턴 => 표현,
}
```

`match` 표현을 사용하기 위해 지켜야 할 사항이 있습니다. `match` 표현에 대응 시킬 값이 가질 수 있는 모든 경우의 수를 **빼짐 없이** 표현해야 합니다. 이 조건을 보장하는 하나의 방법은 `match`의 마지막 갈래에 모든 경우에 대응되는 패턴을 이용하는 것입니다. 예를 들자면 아무 값에나 대응될 변수명 하나를 놓는 패턴이죠. 이 경우 주어진 그 어떠한 값도 해당 변수에 묶이게 되어 실패할 수 없게 되고, 즉 가능한 모든 경우를 표현하게 됩니다.

`_`라는 특별한 패턴은 아무 값에나 대응되지만 그 어떠한 변수에도 묶이지 않습니다. 그래서 `match`의 마지막 갈래에 종종 쓰입니다. `_` 패턴은 명시 하지 않은 값들을 무시할 때 유용합니다. 이 특별한 패턴은 뒤의 "패턴에서 값 무시하기" 절에서 더 자세하게 다룰 것입니다.

if let 조건 표현

6장에서 `if let` 표현을 사용하는 법을 다뤘습니다. 주로 갈래가 하나 밖에 없는 `match` 표현을 더 짧게 표현하는 방법으로 사용됐습니다. 추가적으로 `if let`은 `else` 절을 가질 수 있습니다. 예상 하셨듯 `if let`의 패턴에 값이 대응되지 않을 때 실행됩니다.

예제 18-1은 `if let`, `else if`, `else if let` 표현을 섞어서 사용할 수 있음을 알려주는 코드입니다. 이들을 이용하면 여러 패턴에 하나의 값밖에 대응시키지 못 하는 `match` 표현보다 더 유연하게 표현 가능합니다. 또 연속된 `if let`, `else if`, `else if let`의 각 조건이 꼭 연관될 필요도 없습니다.

예제 18-1의 코드는 배경 색이 어떤 색이 될지 정하기 위해 여러개의 조건을 연이어 체크하는 것을 보여줍니다.

다. 이번 예제에서는 실제 프로그램이 받을지도 모르는 유저의 입력 값들이 하드코딩된 변수들을 이용합니다.

유저가 좋아하는 색을 지정할 경우 그 색이 배경 색이 됩니다. 만일 오늘이 화요일이라면 배경은 녹색이 됩니다. 만일 유저가 나이를 스트링으로 입력했고, 이를 성공적으로 숫자로 파싱해낼 수 있다면 숫자 값에 따라 보라나 주황으로 배경 색이 정해집니다. 위의 그 어떤 조건에도 맞지 않을 경우 배경은 파란색이 됩니다:

Filename: src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {}, as the background", color);
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

예제 18-1: `if let`, `else if`, `else if let`, `else`의 조합

이런 조건절 구조는 복잡한 요구사항도 대응할 수 있게 해줍니다. 주어진 하드코딩된 값들로 예제코드를 실행할 경우 `Using purple as the background color`를 출력 할 것입니다.

보시다시피 `if let` 표현은 `match`의 각 갈래가 그랬듯 변수를 가리는 변수를 새로이 등장 시킬 수 있습니다. `if let Ok(age) = age`는 `Ok`에서 값을 추출한 새로운 `age` 변수가 전의 `age`를 가리게 됩니다. 이는 `if age > 30`을 조건절 밑의 새로운 스코프에서 사용 해야 함을 의미합니다. 우리의 새로운 `age`는 새로운 중괄호가 나타나 새로운 스코프가 시작하기 전까지 유효하지 않으므로 위의 조건절 밑의 스코프 안에서 사용해야 합니다. `if let Ok(age) = age && age > 30`처럼 두 조건을 합쳐서 하나의 절에서 사용할 수 없습니다.

`if let` 표현의 단점은 `match` 표현과 다르게 컴파일러가 해당 구문이 모든 경우를 다뤘는지 판단하지 않는다는 점입니다. 예제의 마지막 `else` 절을 빼먹어서 처리 되지 않는 경우가 생기더라도 컴파일러가 이를 찾아내지 않고 이에 따라 발생 할 수 있는 논리 버그를 사전에 경고해주지 않습니다.

while let 조건 루프

`if let`과 구조적으로 비슷하게 생긴 `while let` 조건 루프는 주어진 값이 패턴에 계속 대응 되는한 실행되는 `while` 루프입니다. 예제 18-2는 vector를 스택처럼 이용해 push된 역순으로 값을 출력하는 `while let` 조건 루프입니다.

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

예제 18-2: `while let`을 이용해 `stack.pop()`이 `Some`을 반환하는 한 값을 출력

이 예제는 3, 2, 1을 출력할 것입니다. `pop` 메소드는 vector의 마지막 값을 꺼내서 `Some(value)`를 반환합니다. 만일 vector가 비었을 경우 `None`을 반환합니다. 코드의 `while` 루프는 `pop`이 `Some`을 반환 할 동안 실행됩니다. `None`이 반환되는 경우에 더 이상 패턴에 맞지 않으므로 멈춥니다. `while let` 표현으로 스택의 모든 값을 간편하게 꺼낼 수 있습니다.

for 루프

3장에서 `for` 루프가 러스트에서 가장 흔하게 쓰이는 반복문 구문인 것을 언급 했습니다. 다만 그 `for` 루프들에서 사용하는 패턴에 대해선 아직 다루지 않았습니다. `for` 루프의 패턴은 `for` 키워드 바로 다음에 오는 값입니다. 즉 `for x in y`에서 `x`가 패턴입니다.

예제 18-3은 `for`에서 패턴을 이용해 튜플을 분해하여 튜플을 `for` 루프의 일부로써 사용하는 것을 보여줍니다.

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{} is at index {}", value, index);
}
```

예제 18-3: 튜플을 분해하기 위해 `for` 루프에서 패턴 사용

예제의 출력결과는 다음과 같습니다.

```
a is at index 0
b is at index 1
c is at index 2
```

`enumerate` 메소드는 해당 반복자를 자신의 값과 값의 `index`를 포함한 튜플들을 순회하도록 바꿔 줍니다. 그래서 `enumerate`의 첫 호출은 `(0, 'a')`를 반환합니다. 이 반환된 튜플은 `(index, value)` 패턴에 대응 되면 `index`는 `0`, `value`는 `'a'` 값을 갖게 합니다. 이들의 값은 출력의 첫 줄에 등장하게 됩니다.

let 구문

이번 장 전까지 `match`나 `if let`에서만 직접적으로 패턴의 사용을 언급했습니다. 하지만 이미 여러 곳에서 패턴을 사용하고 있었습니다. 대표적으로 `let` 구문이 있습니다. `let`을 이용한 변수 대입을 예로 들어 봅시다:

```
let x = 5;
```

이미 수백번은 봤을 `let`입니다. 하지만 아셨을지 모르겠지만 사실 이 구문에는 패턴이 쓰이고 있습니다! `let` 구문을 더 형식에 맞춰 나타내면 다음과 같습니다:

```
let PATTERN = EXPRESSION;
```

`let x = 5;`처럼 `PATTERN` 자리에 변수명이 들어가는 경우, 그 변수명 자체가 하나의 무지하게 단순한 패턴의 한 형태입니다. 러스트는 `EXPRESSION`을 `PATTERN`에 비교하고 거기서 발견한 이름들에 그 `EXPRESSION`을 대입합니다. `let x = 5;` 예제에서 `x`라는 패턴은 "이 패턴에 대응 되는 값을 변수 `x`에 대입하는" 패턴입니다. 이 경우 `x`가 패턴의 전부이므로 "아무 값을 통으로 `x`에 대입한다"는 의미를 갖게 됩니다.

`let`의 좀 더 "패턴 매칭"스러운 면모를 보기 위해 예제 18-4를 봅시다. 이 예제는 튜플을 분해하는 패턴을 사용하고 있습니다.

```
let (x, y, z) = (1, 2, 3);
```

예제 18-4: 패턴을 이용해 튜플을 분해하며 세개의 변수를 생성

이 코드는 튜플을 하나의 패턴에 대응시키고 있습니다. 러스트는 튜플 `(1, 2, 3)`을 `(x, y, z)`에 비교하고 튜플이 해당 패턴에 대응된다는 것을 알아냅니다. 그래서 러스트는 `1`을 `x`에, `2`를 `y`에, `3`을 `z`에 각각 끕습니다. 이 튜플예제를 3개의 변수 대입 패턴을 하나의 패턴으로 끕은 것으로 생각하셔도 좋습니다.

만약 패턴의 원소의 수가 주어진 튜플의 원소의 수와 다를 경우, 둘의 타입이 서로 달라 대응에 실패하고 컴파

일 에러가 발생하게 됩니다. 예제18-5는 원소가 3개인 튜플을 원소가 2개인 패턴에 대응시키려고 하고 있습니다. 물론 실패합니다:

```
let (x, y) = (1, 2, 3);
```

예제 18-5: 대응시키려는 튜플과 원소의 개수가 맞지 않는 패턴의 사용

컴파일을 시도할 경우 다음과 같은 타입에러가 발생합니다:

```
error[E0308]: mismatched types
--> src/main.rs:2:9
 |
2 |     let (x, y) = (1, 2, 3);
 |           ^^^^^^ expected a tuple with 3 elements, found one with 2
elements
 |
= note: expected type `({integer}, {integer}, {integer})`
        found type `(_, _)`
```

튜플의 원소중 하나, 혹은 여럿을 무시하고 싶을 경우 "패턴에서 값 무시하기"에서 자세하게 다룬 `_`나 `..`를 사용할 수 있습니다. 반대로 문제가 패턴 쪽의 변수가 너무 많아서 생겼다면 패턴의 변수 일부를 없애는 것으로 튜플의 원소의 수와 패턴의 변수 원소의 수를 맞추어 둘의 타입을 맞춰주면 됩니다.

함수의 매개변수

함수의 매개변수들도 물론 패턴입니다. 예제 18-6은 `i32` 타입의 매개변수 `x`를 가지는 함수 `foo`를 정의하는 코드입니다. 이제는 익숙한 코드라 생각합니다.

```
fn foo(x: i32) {
    // code goes here
}
```

예제 18-6: 함수 선언의 매개변수는 패턴을 사용

예상대로 `x` 부분은 패턴입니다! `let`에서도 그랬듯, 함수에 인자를 넘기는 과정에서 튜플을 패턴에 대응시킬 수 있습니다. 예제 18-7은 함수에 튜플을 넘기면서 그 값들을 분해해서 사용하는 코드입니다.

Filename: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

예제 18-7: 튜플을 분해하는 매개변수를 가진 함수

이 코드는 `Current location: (3, 5)`를 출력합니다. 값 `&(3, 5)`는 패턴 `&(x, y)`에 대응되므로 `x`는 `3`, `y`는 `5`의 값을 갖게 됩니다.

패턴은 함수의 매개변수에서 사용한 방법 그대로 클로저의 매개변수에서도 사용할 수 있습니다. 13장에서도 다뤘듯 함수와 클로저는 매우 닮았기 때문입니다.

지금까지 다양한 패턴의 사용법을 보았습니다. 하지만 패턴은 사용하는 장소마다 다른 방식으로 작동합니다. 몇 장소에서는 패턴은 반증 불가(irrefutable) 해야 합니다. 이는 패턴이 어떠한 값이든 대응해야 한다는 뜻입니다. 그외의 경우엔 반증 가능(refutable) 해도 됩니다. 이 둘의 차이를 다음 절에서 자세히 다루도록 합시다.

반증 가능성(Refutability): 패턴이 매칭에 실패할지의 여부

패턴은 2가지 형태가 존재합니다. 반증 가능 패턴과 반증 불가 패턴. 주어진 어떠한 값에도 대응되는 패턴을 반증 불가(irrefutable) 패턴이라 합니다. 예를 들면 `let x = 5;` 의 `x`가 있습니다. `x`에 어떠한 값이 오건 대응하기 때문에 실패할 수 없고, 곧 반증 불가합니다. 주어진 값에 대응이 실패할 수 있는 패턴을 반증 가능(refutable) 패턴이라 합니다. `if let Some(x) = a_value;` 의 `Some(x)` 가 그 예시입니다. `a_value`의 값이 `None`인 경우가 있다면 `Some(x)`에 대응하지 못하고 실패하게 됩니다. 즉 반증 가능합니다.

함수의 매개변수, `let` 구문, `for` 루프들은 반증 불가한 패턴만 허용합니다. 이 표현들의 경우 값을 패턴에 대응하는데 실패할 경우 프로그램이 할 수 있는 행동이 없기 때문입니다. 반대로 `if let`과 `while let` 표현은 반증 가능 패턴만 허용합니다. 이 표현들은 성공 여부에 따라 다른 행동을하도록 설계 됐기 때문에 실패의 여지가 있는 패턴이 올 것을 가정합니다.

일반적으로 반증 가능 패턴과 반증 불가 패턴의 차이에 대해 걱정 할 필요는 없습니다. 다만 관련된 에러메세지를 보고 코드를 고치기 위해선 이 반증 가능성이라는 개념을 숙지 해야할 필요가 있습니다. 만일 관련된 에러가 생길 경우 원래 의도한 기능에 맞춰 패턴을 고치거나, 패턴을 이용하는 구문을 고치셔야 합니다.

반증 불가한 패턴이 필요한 곳에서 반증 가능 패턴을 쓰는 경우와 그 반대의 경우를 살펴 봅시다. 예제 18-8은 `let` 구문에서 반증 가능 패턴 `Some(x)`를 쓰고 있습니다. 예상 하셨듯 해당 코드는 에러가 발생합니다.

```
let Some(x) = some_option_value;
```

예제 18-8: `let`에서 반증 가능 패턴의 사용

`some_option_value` 가 `None` 일 경우 `Some(x)`에 대응하는데 실패합니다. 즉 반증 가능 패턴입니다. 하지만 `let` 구문은 반증 불가한 패턴만을 허용합니다. `None`이 왔을 경우 할 수 있는 일이 없기 때문입니다. 러스트는 컴파일 시에 반증 불가 패턴이 필요한 곳에 반증 가능 패턴이 왔다고 불평할겁니다.

```
error[E0005]: refutable pattern in local binding: `None` not covered
-->
|
3 | let Some(x) = some_option_value;
|     ^^^^^^ pattern `None` not covered
```

`Some(x)`의 가능한 모든 경우를 다루지 않았기에 (정확히는 다를 수 없었기에) 러스트는 컴파일 에러를 납니다.

이런 문제를 해결하기 위해 패턴을 이용하는 구문을 바꾸는 방법이 있습니다. `let`을 쓰는 대신 `if let`을 쓰는 것입니다. 이 경우 패턴에 값을 대응하는데 실패하면 중괄호 안의 코드를 넘어가면 됩니다. 전처럼 할 수 있는 일이 없지 않습니다. 예제 18-9는 18-8을 고친 코드입니다.

```
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

예제 18-9: `let` 대신 `if let`과 반증 가능 패턴의 사용

코드에게 탈출구를 만들어줬습니다! 이 코드에 문제는 없습니다. 다만 반증 불가 패턴을 사용할 경우 에러를 받게 되겠죠. 예제 18-10처럼 `if let`에 `x`처럼 모든 값에 대응되는 패턴을 쓸 경우 에러가 발생할 겁니다:

```
if let x = 5 {
    println!("{}", x);
};
```

예제 18-10: `if let`과 반증 불가 패턴의 사용

러스트는 틀려도 되야할 `if let` 구문에 틀릴리 없는 패턴을 쓰는 것은 말이 되지 않는다고 불평할겁니다:

```
error[E0162]: irrefutable if-let pattern
--> <anon>:2:8
  |
2 | if let x = 5 {
  |         ^ irrefutable pattern
```

이 때문에 `match`의 갈래는 마지막 갈래를 제외하고는 반증 가능한 패턴을 써야합니다. 마지막 갈래는 빠짐 없이 대응해야 하는 `match` 표현의 특성상 반증 불가한 패턴을 써야하고요. 러스트에서 `match`를 반증 불가 패턴을 가진 하나의 갈래만으로 구성하는 것은 가능합니다만 `let` 구문 하나로 대체될 수 있기에 딱히 득 볼 것이 없습니다.

패턴이 사용 될 수 있는 코드 상의 모든 곳을 알고, 반증 가능 패턴과 반증 불가 패턴의 차이를 이해 했으니 다음은 패턴들을 만드는데 사용할 수 있는 모든 문법에 대해 알아봅시다.

패턴 문법의 모든 것

여러분은 이 책을 읽는 내내 수많은 종류의 패턴 예시를 보셨을 겁니다. 이번 절에선 유효한 패턴 구문을 모두 살펴보고, 그것들을 각각 왜 사용해야 하는지 알아보도록 하겠습니다.

리터럴 매칭

6장에서 보신 것처럼 여러분은 패턴과 리터럴을 직접 매칭할 수 있습니다. 다음 코드가 예시입니다.

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

x의 값이 1이기 때문에 이 코드는 one을 출력합니다. 이 구문은 특정한 구체적인 값을 가질 때 행동하도록 여러분의 코드를 작성하는데 도움이 됩니다.

명명 변수 매칭

명명 변수는 어떠한 값에도 매칭되는 반증 불가능한 패턴이며, 우린 이걸 이 책에서 여러번 써왔습니다. 어찌 됐건, 여러분이 match 표현에서 명명 변수를 사용할 때 문제가 있습니다. 바로 match는 새로운 스코프를 만들기 때문에 match 표현 내에서 패턴의 일부로서 선언된 변수는 match 구조 외부의 동일한 이름을 가진 변수를 가려버린다는 겁니다. Listing 18-11에서 Some(5) 값으로 x 변수를 선언하고, 10 값으로 y를 선언했습니다. 한번 코드를 실행하거나 뒷부분의 설명을 읽지 않고 매칭 갈래 내의 패턴과 마지막의 println!을 보고 이 코드가 뭘 출력할지 맞춰보세요:

Filename: src/main.rs

```

fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {:?}", y),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
}

```

Listing 18-11: 갈래에서 `y` 변수를 새로 만들어 기존의 것이 가려지도록 한 `match` 표현

`match` 표현이 실행되었을 때 어떤 일이 일어나는지 살펴봅시다. 일단 첫 번째 갈래는 정의된 변수 `x` 와 매칭되지 않으니, 해당 코드는 실행되지 않고 넘어갑니다.

두 번째 갈래 패턴에서는 `Some` 값 안에 있는 값에 대응될 새 변수 `y` 가 등장합니다. 현재 우린 `match` 표현 내의 새로운 스코프에 위치해 있기 때문에, 이 `y` 는 처음에 10 의 값을 갖도록 선언한 것이 아니라 새로운 변수입니다. 이 새로운 `y` 는 우리가 `x` 로 가지고 있는 `Some` 내부의 어떠한 값으로 바인딩될 것입니다. 따라서, 이 새 `y` 는 `x` 의 `Some` 내부 값인 `5` 로 바인딩 되고, 해당 갈래의 표현이 실행되어 `Matched, y = 5` 가 출력됩니다.

만약 `x` 가 `Some(5)` 이 아니라 `None` 값을 갖고 있었다면 첫 번째와 두 번째 갈래는 매치되지 않고 언더스코어와 매칭되었을 겁니다. 언더스코어 갈래에선 `x` 변수를 새로 만들지 않았기에 `x` 는 가려지지 않은 상태로 여전히 바깥의 `x` 변수를 나타내고, 만약 코드를 실행한다면 `match` 는 `Default case, x = None` 을 출력 할 겁니다.

`match` 표현이 끝나면 안쪽의 `y` 를 갖던 스코프도 끝납니다. 그리고 마지막 `println!` 은 `at the end: x = Some(5), y = 10` 를 출력합니다.

기존 변수를 가리는 변수를 만들지 않고 외부의 `x` 와 `y` 의 값을 비교하는 `match` 표현을 만들기 위해선 조건부(conditional) 매치 가드(match guard)를 사용해야 하는데, 매치 가드에 대해선 이후 “매치 가드를 이용한 추가 조건” 절에서 다루도록 하겠습니다.

다중 패턴

여러분은 `match` 표현 내에서 `or` 을 뜻하는 `|` 구문을 이용해 여러 개의 패턴과 매치시킬 수 있습니다. 예를 들어, 다음 코드는 `x` 값을 매치 갈래와 매치시키는데, 첫 번째 갈래에서 `or` 옵션을 사용하고 있습니다. 이럴 경우 해당 갈래 내의 값 중 일치하는 값이 있으면, 즉 `x` 가 `1` 이나 `2` 일 경우 해당 갈래의 코드가 실행됩니다.

다:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

따라서 코드는 `one or two` 를 출력합니다.

... 를 이용한 값의 범위 매칭

우린 `...` 구문을 이용해 값의 범위 내에 매치시킬 수 있습니다. 다음 코드에선 패턴이 범위 내 값에 매칭될 경우 해당 갈래가 실행됩니다.

```
let x = 5;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("something else"),
}
```

만약 `x` 가 1, 2, 3, 4, 5 중 하나라면 첫번째 갈래와 매치됩니다. 우리가 같은 코드를 앞서 설명한 `|` 이용해 작성했다면 `1 | 2 | 3 | 4 | 5` 라고 써야 했겠지만, 이 구문을 이용하면 `1 ... 5` 로 더 간편하게 작성할 수 있습니다. 만약 우리가 1에서 1,000 까지의 숫자중 아무 숫자나 매치시켜야 하는 상황이라면 이는 훨씬 더 짧고, 유용할 겁니다.

이 값의 범위를 이용한 매칭 방식은 숫자 값이나 `char` 값에만 사용할 수 있습니다. 컴파일러가 컴파일 타임에 해당 범위가 비어있지 않은지 검사하는데, 러스트가 해당 범위가 비어있는지 알아낼 수 있는 타입의 종류가 정수 값과 `char` 뿐이기 때문입니다.

`char` 값의 범위를 이용하는 예시는 여기 있습니다:

```
let x = 'c';

match x {
    'a' ... 'j' => println!("early ASCII letter"),
    'k' ... 'z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

러스트는 `c` 가 첫번째 패턴(앞쪽 순서의 알파벳들)의 범위 안에 속한다는 것을 알아낼 수 있고, `early ASCII letter` 을 출력합니다.

값을 해체하여 분리하기

우린 패턴을 이용해 구조체, 열거형, 튜플, 참조자 등의 값을 해체(destructuring)할 수도 있습니다. 각각에 대해 알아봅시다.

구조체 해체하기

Listing 18-12 는 `x` 와 `y` 두개의 필드를 가진 `Point` 구조체를 나타냅니다. 이는 `let` 구문과 패턴을 이용해 이를 해체해봅시다:

Filename: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

Listing 18-12: 구조체의 필드를 여러 분리된 변수로 해체하기

이 코드는 `p` 변수의 필드인 `x` 와 `y` 에 각각 대응되는 `a` 와 `b` 변수를 생성합니다. 이 예시를 보면 패턴 내 변수의 이름이 꼭 구조체의 필드명과 일치할 필요는 없다는 것을 알 수 있습니다. 하지만 해당 변수가 어떤 필드를 나타내는지 기억하기 쉽도록 필드명과 일치하도록 작성하는게 일반적입니다.

다만 일치하도록 작성할 때 `let Point { x: x, y: y } = p;` 는 `x` 와 `y` 이 중복됩니다. 여러분은 이때, 즉 패턴이 구조체 필드명과 일치할 때 약칭 구문을 사용할 수 있습니다: 여러분은 구조체 필드명을 나열하는 것만으로 해당 필드의 값을 가진 변수를 만들어낼 수 있습니다. Listing 18-13 은 Listing 18-12 의 코드를 `let` 패턴내의 `a` 와 `b` 대신 `x` 와 `y` 변수를 사용하도록 변경한 예시입니다.

Filename: src/main.rs

```

struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}

```

Listing 18-13: 약칭 구문을 이용해 구조체 필드 해체하기

이 코드는 `p` 변수의 `x` 와 `y` 필드에 대응되는 `x` 와 `y` 변수를 만들어냅니다. 결과는 `x` 와 `y` 변수가 `p` 구조체 내의 값을 갖는 것으로 나옵니다.

또한, 모든 필드에 대응하는 변수를 만들지 않고 구조체 패턴의 일부에 리터럴 값을 이용해 해체할 수도 있습니다. 이렇게 함으로써 어떤 필드가 특정 값에 해당하는지를 검사하면서 나머지 필드를 해체한 변수를 만들 수 있습니다.

Listing 18-14 는 `Point` 값을 3가지 경우로 나눈 `match` 표현을 나타냅니다: `x` 축 위의 점(`y = 0` 일때 참) 인 경우, `y` 축 위인 경우(`x = 0`), 혹은 그 외일 경우:

Filename: src/main.rs

```

fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        Point { x: 0, y } => println!("On the y axis at {}", y),
        Point { x, y } => println!("On neither axis: ({}, {})", x, y),
    }
}

```

Listing 18-14: 한 패턴 내에서 해체와 리터럴 값 매칭

첫 번째 갈래는 `x` 축 위의 점인 경우에 해당하기기에, `y` 필드가 `0` 값과 매치될 경우에 매치될 수 있도록 하였습니다. 또한 패턴은 여전히 `x` 변수를 생성하기 때문에 해당 변수를 갈래 내 코드에 사용할 수 있습니다.

비슷하게, 두 번째 갈래는 `x` 필드를 `0` 값과 매치시킬 경우에 매치되도록 하여 `y` 축 위의 점인지 판별합니다. 마찬가지로 `y` 필드에 해당하는 `y` 변수도 생성됩니다. 세 번째 갈래는 어떤 리터럴도 특정하지 않습니다. 따라서 모든 `Point` 에 매치되고 모든 `x` 와 `y` 필드에 대한 변수를 생성합니다.

이 예제에서 `p` 값은 `x` 가 `0` 이기 때문에 두 번째 갈래에 매치됩니다. 따라서 이 코드는 `On the y axis at 7` 를 출력합니다.

열거형 해체

우린 이미 이 책 6장의 Listing 6-5 에서 열거형을 해체해 봤습니다. 다만 한 가지 다루지 않은 내용이 있는데, 열거형을 해체하기 위한 패턴은 해당 열거형에 내장된 데이터의 정의 방식과 일치해야 합니다. 예시를 들기 위해, Listing 18-15 에 Listing 6-2 에서 사용했던 `Message` 열거형을 이용하고 각 내부 값을 해체하기 위한 `match` 패턴을 작성했습니다:

Filename: src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.")
        },
        Message::Move { x, y } => {
            println!(
                "Move in the x direction {} and in the y direction {}",
                x,
                y
            );
        }
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}
```

Listing 18-15: 다른 종류의 값들을 갖는 열거형 variant 해체

이 코드는 `Change the color to red 0, green 160, and blue 255` 를 출력합니다. 한번 `msg` 값을 이리저리 변경해보고 다른 갈래들을 실행시켜보세요.

아무 값도 갖지 않는 `Message::Quit` 같은 variant는 값을 해체할 수 없습니다. 따라서 리터럴 `Message::Quit` 값만 매치시킬 수 있으며, 패턴 내의 변수는 없습니다.

`Message::Move` 등의 유사 구조체 variant의 경우는 우리가 구조체를 매칭시킬 때의 패턴과 유사한 패턴을 사용합니다. variant 명 뒤에 중괄호를 작성하고, 해당 갈래에서 사용할 변수를 나타내는 필드들을 나열합니다. 이땐 Listing 18-13에서 사용했던 약칭 구문을 사용했습니다.

하나의 요소를 가진 `Message::Write` 튜플과 세 개의 요소를 가진 `Message::ChangeColor` 튜플 등의 유사 튜플 variant에 사용하는 패턴은 튜플을 매칭시킬 때 지정한 패턴과 유사합니다. 이때 패턴 내 변수의 개수는 우리가 매칭하려는 variant 내 요소의 개수와 일치해야 합니다.

참조자 해체

우리가 패턴과 매칭하려는 값이 참조자를 포함하고 있을 땐 패턴 내에서 `&` 를 사용해 값으로부터 참조자를 해체해야 합니다. 이렇게 하면 참조자를 갖는 변수를 가져오는 대신 참조자가 가리키는 값을 갖는 변수를 가져올 수 있습니다. 이는 참조자들을 반복하는 반복자에서 클로저를 사용할 때, 해당 클로저 내에서 참조자가 아닌 참조자가 가리키는 값을 사용하기 원할 경우에 유용합니다.

Listing 18-16의 예시는 `vector` 내의 `Point` 객체들을 가리키는 참조자들을 반복하며 구조체 참조자를 해체하는 것으로 간단하게 `x` 와 `y` 값을 계산할 수 있습니다:

```
let points = vec![
    Point { x: 0, y: 0 },
    Point { x: 1, y: 5 },
    Point { x: 10, y: -3 },
];
let sum_of_squares: i32 = points
    .iter()
    .map(|&Point { x, y }| x * x + y * y)
    .sum();
```

Listing 18-16: 구조체 참조자를 구조체 필드 값들로 해체하기

이 코드의 `sum_of_squares` 변수는 135의 값을 갖습니다. 이 값은 `points` 벡터 내 `Point` 각각의 `x` 와 `y` 값을 제곱하고, 더한 값을 모두 합친 값입니다.

만약 `&Point { x, y }`에서 `&` 를 뺏다면 타입 불일치(type mismatch) 오류가 발생합니다. `iter` 는 벡터 내 요소들의 실제 값이 아닌 참조자이기 때문입니다. 실제 오류는 다음과 같습니다:

```
error[E0308]: mismatched types
 -->
14 |     .map(|Point { x, y }| x * x + y * y)
   |          ^^^^^^^^^^ expected &Point, found struct `Point`
   |
   = note: expected type `&Point`
           found type `Point`
```

이 오류는 리스트에선 클로저에서 `&Point` 를 사용하길 원했지만, 우리가 `Point` 참조자가 가리키는 값이 아니라 `Point` 값을 직접 매치시키려고 시도했기 때문입니다.

구조체와 튜플 해체

우린 패턴 해체를 더 복잡한 방법으로 섞고, 비교하고, 중첩시킬 수 있습니다. 다음 예제는 튜플 내에 구조체와 튜플을 갖는, 즉 중첩된 구조에서 본래 값을 얻는 복잡한 해체를 보여줍니다:

```
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 });
```

이 코드는 우리가 복잡한 타입의 컴포넌트를 분리하고 각각의 값을 사용할 수 있게 합니다.

패턴을 이용한 해체는 구조체의 각 필드값과 같은 일부의 값을 서로 별도로 사용하는 편리한 방법입니다.

패턴 내에서 값 무시하기

여러분은 패턴 내에서 값을 무시하는 게 유용한 것을 종종 보셨을 겁니다. 예를 들면, `match` 의 마지막 갈래에 될 수 있지만 아무것도 하지 않는 모든 나머지 값을 매칭시킬 때요. 전체 혹은 일부 값을 무시하는 방법은 몇 가지 있습니다: 여러분이 여태 보신 것처럼 `_` 패턴을 이용하거나, 다른 패턴 내에서 `_` 패턴을 사용하거나, 언더스코어(`_`)로 시작하는 이름을 사용하거나, 값의 나머지 부분을 무시하기 위해 `..` 를 사용하는 것이죠. 한번 이들에 대해서 각각 어떻게 사용하고, 왜 사용하는지 알아봅시다.

`_` 를 이용해 전체 값 무시하기

`_` 언더스코어 와일드카드 패턴은 어떤 값과도 매치되지만 값으로 바인드(bind) 하지는 않는 것으로 사용되어 왔습니다. `_` 언더스코어 패턴은 `match` 의 마지막 갈래 표현에서 특히 유용하지만, Listing 18-17처럼 함수 매개변수를 포함한 모든 패턴에서 사용할 수 있습니다.

Filename: src/main.rs

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

Listing 18-17: 함수 시그니처에서 `_` 사용하기

이 코드는 첫번째 인자로 전달된 값인 `3`을 완벽히 무시하고, `This code only uses the y parameter: 4`를 출력합니다.

대부분의 경우 특정 함수의 매개변수가 더 이상 필요하지 않다면, 해당 함수의 시그니처를 더 이상 사용되지 않는 매개변수가 포함되지 않도록 변경해야 합니다. 다만 몇몇 경우에는 함수의 매개변수를 무시하는 것이 유용할 때도 있습니다: 예를 들어, 여러분이 트레이잇을 구현할 때 특정 타입의 시그니처가 필요한데 함수 본문에 선 매개변수 중 하나가 필요하지 않은 경우입니다. 컴파일러는 이때 사용되지 않은 매개 변수에 관해서 경고하지 않습니다. 단, 언더스코어가 아닌 이름을 사용할 경우엔 경고합니다.

중첩된 `_` 를 이용해 값의 일부 무시하기

`_` 를 다른 패턴 내에 사용해서 값의 일부를 무시할 수도 있습니다: 값의 일부를 테스트 하려는데 해당 코드에서 그 외의 나머지 부분은 필요하지 않을 때 이 기능을 사용할 수 있습니다. Listing 18-18은 설정 값을 관리하는 코드를 보여줍니다. 비즈니스 요구 사항은 사용자가 기존 커스텀을 덮어쓰진 않아야 하지만 기존 설정을 해제할 순 있으며 해제된 상태에선 값을 설정하는 것이 가능해야 한다는 것입니다.

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value);
```

Listing 18-18: `Some` 내부 값이 필요하지 않은 상황에서 패턴 내에 언더스코어를 사용해 `Some` variant와 매치시키기

이 코드는 `Can't overwrite an existing customized value` 와 `setting is Some(5)` 를 출력합니다. 우린 첫 번째 매치 갈래에서 두 `Some` variant 모두 내부 값을 매치시키거나 사용할 필요가 없고, 그저 `setting_value` 와 `new_setting_value` 가 모두 `Some` variant 인지만 확인하면 됩니다. 조건을 만족하면, 왜 `setting_value` 를 변경하지 않는지 이유를 출력하고 값을 변경하지 않습니다.

나머지 모든 경우는 (`setting_value` 나 `new_setting_value` 둘 중 하나가 `None` 인 경우) 두 번째 갈래에서 `_` 로 표현되고, `setting_value` 는 `new_setting_value` 로 변경됩니다.

우린 특정 값들을 무시하기 위해 한 패턴 내에서 언더스코어를 여러번 사용할 수도 있습니다. Listing 18-19 는 5개의 요소를 가진 튜플에서 두 번째와 네 번째 요소만 무시하는 예제입니다:

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    },
}
```

Listing 18-19: 튜플의 여러 부분 무시하기

이 코드는 `Some numbers: 2, 8, 32` 를 출력하고, 4 와 16 은 무시됩니다.

언더스코어로 시작하는 이름을 이용해 쓰이지 않는 변수 무시하기

만약 변수를 생성했는데 아무 곳에서도 사용하지 않는다면, 보통 러스트는 이를 버그가 될 수 있다고 경고합니다. 하지만 프로토타입을 만드는 중이거나 프로젝트를 막 시작했을 때와 같이 아직 사용하진 않아도 미리 변수를 만들어 두는 것이 유용할 때도 있습니다. 이럴 경우 해당 변수명을 언더스코어로 시작하도록 만들면 러스트는 해당 미사용 변수에 대해 경고를 생성하지 않습니다. Listing 18-20 에선 두 개의 미사용 변수를 생성하지만 이 코드를 실행할 땐 하나의 경고만 나타납니다.

Filename: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Listing 18-20: 미사용 변수 경고를 피하기 위해 변수명을 언더스코어로 시작하도록 하기

`y` 변수를 사용하지 않았다는 경고가 나타나지만, 언더스코어로 시작하는 `x` 변수는 경고가 나타나지 않습니다.

`_` 하나만 쓰는것과 언더스코어로 시작하는 변수명의 차이점을 알아두세요. `_` 는 어떠한 값도 바인드되지 않지만, `_x` 는 여전히 값이 바인드됩니다. Listing 18-21 은 이 미묘한 차이가 중요한 포인트가 되는 좋은 예시입니다. 이 코드는 에러가 발생합니다:

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    println!("found a string");
}

println!("{:?}", s);
```

Listing 18-21: 언더스코어로 시작하는 변수는 여전히 값이 바인드되기 때문에 해당 값의 소유권을 가져갑니다

`s` 값이 `_s`로 이동되었기 때문에, `s` 를 다시 사용할 수 없다는 오류가 발생합니다. 반면 언더스코어만 사용하는 경우는 값이 바인드되지 않습니다. 따라서 `s` 는 `_` 로 이동하지 않고, Listing 18-22 는 컴파일 시 어떤 에러도 발생하지 않습니다:

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

println!("{:?}", s);
```

Listing 18-22: 언더스코어를 사용하면 값이 바인드되지 않습니다

우린 `s` 를 어느 것에도 바인드하지 않았습니다; 따라서 `s` 는 이동하지 않고, 이 코드는 잘 작동합니다.

.. 를 이용해 값의 나머지 부분 무시하기

여러 요소를 갖는 값을 다룰 때, 값의 일부만 사용하고 나머지는 무시하기 위해 `..` 구문을 사용할 수 있습니다. 이 구문은 무시할 각 값에 언더스코어를 하나하나 작성하는 끔찍한 사고를 막아주기도 합니다. `..` 패턴은 우리가 패턴에서 명시하지 않은 값의 나머지 부분을 모두 무시합니다. Listing 18-23 은 3차원 공간의 좌표를 갖는 `Point` 구조체를 갖습니다. 이 `match` 표현에서는 `x` 좌표만 사용하고 `y` 와 `z` 필드의 값은 무시합니다:

```

struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}

```

Listing 18-23: `..` 를 사용해 `Point` 의 `x` 필드 외 모든 필드 무시하기

`x` 를 나열하고, `..` 패턴을 포함했습니다. 이는 `y: _` 와 `z: _` 를 나열하는 것 보다 간결하고, 이보다 더 많은 필드를 갖는 구조체에서 한 두개의 필드만 필요할 상황에선 훨씬 더 간결합니다.

`..` 구문은 필요한 만큼 많은 값으로 확대될 수 있습니다. Listing 18-24 는 `..` 를 튜플과 사용하는 법을 보여줍니다:

Filename: src/main.rs

```

fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        },
    }
}

```

Listing 18-24: 튜플의 첫 번째와 마지막 값만 매칭시키고 나머지 값은 모두 무시하기

이 코드에선 첫 번째와 마지막 값이 `first` 와 `last` 에 매치되고, `..` 는 중간의 모든 값과 매치됩니다.

다만 `..` 를 사용할 땐 모호하지 않아야(unambiguous) 합니다. 만약 어떤 값이 매치되고 어떤 값이 무시되어야 하는지 명확하지 않다면 러스트는 에러를 발생시킵니다. Listing 18-25 는 `..` 를 모호하게 사용해 컴파일 되지 않는 경우의 예시를 보여줍니다.

Filename: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}
```

Listing 18-25: `..` 를 모호하게 사용해보기

이 예시를 컴파일하면 다음과 같은 에러가 나타납니다:

```
error: `..` can only be used once per tuple or tuple struct pattern
--> src/main.rs:5:22
   |
5 |         (.., second, ..) => {
   |             ^
   |
```

러스트는 무시할 튜플 요소 중 몇개를 `second` 전에 두고, 후에 몇개를 둘 지 결정할 수 없습니다. 이 코드는 `2` 를 무시하고 `4` 를 바인드한 뒤 `8, 16, 32` 를 무시하거나; `2` 와 `4` 를 무시하고 `8` 을 바인드한 뒤 `16` 과 `32` 를 무시하는 등을 의미할 수 있습니다. 러스트에서 `second` 변수명은 그 어떤 특별한 의미도 없고, 이렇게 두 곳에 `..` 를 사용하는 것은 모호하므로 우리는 컴파일러 에러를 받습니다.

ref 와 ref mut 를 이용해 패턴 내에서 참조자 생성하기

`ref` 를 사용해 참조자를 만들어서 패턴 내 변수로 값의 소유권이 이동하지 않도록 하는 법을 알아봅시다. 보통 패턴과 매치시킬 경우 패턴에 나타난 변수에 값이 바인드됩니다. 러스트의 소유권 규칙에 따르면 값은 `match` 내부 혹은 여러분이 패턴을 사용하는 모든 곳으로 이동됩니다. Listing 18-26 은 `match` 의 패턴에서 변수로 받고, 값을 `match` 이후 `println!` 구문에서 사용하는 예시입니다. 이 코드에서는 `robot_name` 값의 소유권이 `match` 의 첫 번째 갈래에서 `name` 변수로 이전되었기 때문에 컴파일 시 오류가 발생합니다:

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    Some(name) => println!("Found a name: {}", name),
    None => (),
}

println!("robot_name is: {:?}", robot_name);
```

Listing 18-26: `match` 갈래의 패턴에서 값의 소유권을 갖는 변수 생성하기

`robot_name` 의 소유권이 `name` 으로 이동했기 때문에 `robot_name` 은 더 이상 소유권을 갖지 않고, 따라서 `match` 이후 `println!` 에서 `robot_name` 을 사용할 수 없습니다.

이 코드를 고치기 위해선 `Some(name)` 에서 `robot_name` 의 소유권을 가져가는 것이 아닌 빌려야 (*borrow*) 합니다. 패턴을 벗어나서, 값을 빌리는 방법은 `&` 를 이용해 참조자를 생성하는 것이라고 우린 배웠습니다. 따라서 여러분은 `Some(name)` 을 `Some(&name)` 으로 변경하는 것이 해결책이라 생각할 것 입니다.

하지만 여러분이 "값을 해체하여 분리하기" 절에서 보신 것처럼, 패턴 내에서의 `&` 구문은 참조자를 생성하는 것이 아닌, 이미 존재하는 참조자를 값으로 매치합니다. `&` 는 이미 패턴 내에서 다른 뜻을 갖기 때문에, 우린 패턴 내에서 참조자를 생성하기 위해 `&` 를 사용할 수 없습니다.

대신 우린 Listing 18-27 에 나오는 것처럼 새 변수 앞에 `ref` 키워드를 사용해 패턴 내에서 참조자를 생성할 수 있습니다:

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    Some(ref name) => println!("Found a name: {}", name),
    None => (),
}

println!("robot_name is: {:?}", robot_name);
```

Listing 18-27: 패턴 변수가 값의 소유권을 갖지 않도록 참조자 생성하기

`robot_name` 의 `Some` 내 variant 값이 `match` 로 이동하지 않기 때문에 이 예제는 정상적으로 컴파일 됩니다; `match` 는 `robot_name` 의 데이터를 이동시키는 대신 참조자만 갖습니다.

매치된 패턴 내에서 값을 변경하기 위해 가변 참조자를 생성하려면 `&mut` 대신 `ref mut` 을 사용해야 합니다. 이유는 똑같이 패턴 내에서의 `&` 는 새 참조자를 생성하는 것이 아닌 이미 존재하는 가변 참조자를 매치시키는데 사용되기 때문입니다. Listing 18-28 은 가변 참조자를 생성하는 패턴의 예시를 보여줍니다:

```
let mut robot_name = Some(String::from("Bors"));

match robot_name {
    Some(ref mut name) => *name = String::from("Another name"),
    None => (),
}

println!("robot_name is: {:?}", robot_name);
```

Listing 18-28: `ref mut` 를 이용해 패턴 내에서 가변 참조자 생성하기

이 예제는 문제 없이 컴파일되고 `robot_name is: Some("Another name")` 를 출력합니다. `name` 은 가변 참조자이기 때문에 매치 갈래 코드 내에서 값을 변경하기 위해선 `*` 연산자를 이용해 역참조해야 합니다.

매치 가드를 이용한 추가 조건

매치 가드(match guard) 는 `match` 갈래 뒤에 추가로 붙는 `if` 조건으로, 이것이 있을 경우 패턴 매칭과 해당 조건이 모두 만족되어야 해당 갈래가 선택됩니다. 매치 가드는 패턴만 사용하는 것 보다 복잡한 아이디어를 표현하는데 유용합니다.

조건은 패턴 내에서 생성된 변수를 사용할 수 있습니다. Listing 18-29에서 `match` 의 첫 번째 갈래가 `Some(x)` 패턴과 `if x < 5` 매치 가드를 갖는 것을 볼 수 있습니다:

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

Listing 18-29: 패턴에 매치 가드 추가

이 예제는 `less than five: 4` 를 출력합니다. `num` 이 첫 번째 갈래에서 비교될 때, `Some(4)` 는 `Some(x)` 에 매치되기 때문에 매치됩니다. 그리고 매치 가드는 `x` 가 `5` 보다 작은지 검사합니다. 이 경우 참이므로, 첫 번째 갈래가 선택됩니다.

`num` 이 `Some(10)` 이었다면, 10 은 5보다 크기 때문에 첫 번째 갈래의 매치 가드는 거짓이 됩니다. 러스트는 두 번째 갈래로 이동하고, 두 번째 갈래는 매치 가드를 갖지 않으니 모든 `Some` variant 에 매치됩니다. 따라서 두 번째 갈래와 매치됩니다.

`if x < 5` 조건문을 패턴 내부에서 표현할 방법은 없지만, 매치 가드는 우리에게 이 로직을 표현할 수 있는 능력을 부여해 줍니다.

Listing 18-11에서 매치 가드를 이용해 패턴 가림 문제를 해결할 수 있다고 말했었습니다. `match` 표현의 패턴 내에서 새로 생성한 변수가 기존에 있던 `match` 바깥의 변수를 가려버려서 기존의 변수를 테스트 할 수 없다는 문제였죠. Listing 18-30은 매치 가드를 사용해 이 문제를 해결하는 방법을 보여줍니다:

Filename: src/main.rs

```

fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {:?}", n),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}", x, y);
}

```

Listing 18-30: 외부 변수와 같은지 테스트하기 위해 매치 가드를 사용

현재 이 코드는 `Default case, x = Some(5)` 를 출력합니다. 두 번째 매치 갈래는 새 변수 `y` 를 생성하지 않으며, 바깥의 `y` 를 가리지 않습니다. 즉, 우린 매치 가드에서 바깥의 `y` 를 사용할 수 있습니다. 그리고 바깥의 `y` 를 가리게 될 `Some(y)` 패턴을 지정하는 대신 `Some(n)` 을 사용했습니다. 이는 새 변수 `n` 를 생성하지만, `match` 밖에 `n` 변수는 존재하지 않기 때문에 아무것도 가리지 않습니다.

`if n == y` 매치 가드는 패턴이 아니므로 새 변수를 생성하지 않습니다. 여기서의 `y` 는 바깥의 `y` 를 가린 새 변수가 아닌 바깥의 `y` 입니다. 그리고 `n` 을 바깥의 `y` 와 비교하여 같은 값인지 판별합니다.

여러분은 `or` 연산자인 `|` 를 사용해 다중 패턴을 지정한 것에도 매치 가드를 사용할 수 있습니다. 이때 매치 가드 조건은 모든 패턴에 적용됩니다. Listing 18-31 은 매치 가드 앞에 `|` 를 이용한 다중 패턴을 결합한 모습을 보여줍니다. 여기서 중요한 점은 `if y` 매치 가드가 `6` 에만 적용되는 것처럼 보여도 사실은 `4`, `5`, `6` 모두에 적용된다는 것입니다:

```

let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}

```

Listing 18-31: 다중 패턴과 매치 가드의 결합

갈래의 매치 조건 상태는 `x` 가 `4`, `5`, `6` 중 하나이고 `y` 가 `true` 여야 합니다. 이 코드를 실행하면, `x` 가 `4` 이기에 첫 번째 갈래의 패턴에 매치되지만, `if y` 매치 가드가 거짓이 되기 때문에 첫 번째 갈래는 선택되지 않습니다. 코드는 두 번째 갈래로 이동하고 매치되며, 프로그램은 `no` 를 출력합니다. 이렇게 되는 이유는 `if` 조건이 마지막 값인 `6` 에만 적용되는 것이 아닌 `4 | 5 | 6` 패턴 전체에 적용되기 때문입니다. 즉, 매치 가드와 앞의 패턴과의 관계는 다음과 같습니다:

```
(4 | 5 | 6) if y => ...
```

다음은 틀린 관계입니다:

```
4 | 5 | (6 if y) => ...
```

이 코드를 실행하고 나면 전자가 맞다는 것이 명확해집니다: 만약 매치 가드가 1로 연결한 값 리스트의 마지막 값에만 적용되었다면, 해당 갈래는 매치되고 프로그램은 yes를 출력했을 것입니다.

❸ 바인딩

at 연산자인 @는 해당 값이 패턴과 매치되는지 확인하는 동시에 해당 값을 갖는 변수를 생성할 수 있도록 해줍니다. Listing 18-32는 `Message::Hello`의 `id` 필드가 3...7 범위 내에 있는지 테스트하는 예시입니다. 하지만 우린 그 값을 `id_variable` 값에 바인드하고, 그 갈래의 코드에서 사용하길 원합니다. 우린 이 변수를 필드명과 똑같이 `id`라는 이름으로 만들 수도 있지만, 이번 예제에선 다른 이름을 사용하겠습니다:

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3...7 } => {
        println!("Found an id in range: {}", id_variable)
    },
    Message::Hello { id: 10...12 } => {
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        println!("Found some other id: {}", id)
    },
}
```

Listing 18-32: 패턴 내에서 값을 테스트하는 동시에 @를 이용해 값 바인드하기

이 예제는 `Found an id in range: 5`를 출력할 겁니다. 3...7 범위 앞에 `id_variable @`를 지정하는 것으로 값이 범위 패턴과 매치되는지 테스트하면서 해당 값을 캡처할(capturing) 수 있습니다.

두 번째 갈래엔 범위 패턴만 존재합니다. 이 갈래의 코드에는 `id` 필드 값을 담는 변수가 없습니다. `id` 필드의 값은 10, 11, 12 중 하나가 될 수 있지만, 코드는 값이 몇인지 알지 못하고, `id` 값을 변수로 저장하지 않

있기 때문에 `id` 필드의 값을 사용할 수도 없습니다.

마지막 갈래에선 범위를 명시하지 않았고, `id`라는 이름의 변수를 해당 갈래의 코드 안에서 사용할 수 있습니다. 이유는 구조체 필드 약칭 구문을 사용했기 때문입니다. 하지만 첫 두 갈래와는 달리 `id` 필드에 대해 어떠한 테스트도 적용하지 않았습니다: 모든 값은 이 패턴과 매치될 겁니다.

`@` 를 사용하면 값을 테스트하고 변수로 저장하는 일을 한 패턴 내에서 할 수 있습니다.

정리

러스트의 패턴은 다른 종류의 데이터를 구별하는데 굉장히 유용합니다. `match` 표현 내에서 패턴을 사용하면 러스트가 여러분의 패턴이 모든 가능한 값을 커버할 수 있다는 것을 보장합니다. 아닐 경우에는 여러분의 프로그램은 컴파일 되지 않을 것입니다. `let` 구문과 함수 매개 변수에서의 패턴은 그들을 더 유용하게 구성할 수 있게 해주고, 변수의 할당과 동시에 작은 부분의 값들로 해체하는 것을 가능하게 해줍니다. 우린 우리의 필요에 맞게 간단하거나 복잡한 패턴을 만들 수 있습니다.

다음으로, 이 책의 끝에서 두번째 장에선 러스트의 고급 기능에 대해서 알아보도록 하겠습니다.

고급 기능들

지금까지, 여러분은 러스트 프로그래밍 언어의 가장 일반적으로 사용되는 부분들을 공부했습니다. 20장에서 한가지 프로젝트를 더 하기 전에, 우리는 여러분이 가끔 마주치게 될지 모를 이 언어의 몇 가지 측면들을 살펴볼 것입니다. 여러분이 러스트를 사용하다가 어떤 모르겠는 것을 마주칠 때 이 장을 참고자료로 쓸 수 있습니다. 이 장에서 여러분이 공부하게 될 기능들은 매우 특정한 경우에서만 유용합니다. 여러분이 종종 이것들에 근접하지 않게 될지라도, 우리는 여러분이 러스트가 제공해야 하는 모든 기능들에 대해 확실히 이해하기를 바랍니다.

이 장에서 우리가 다룰 것들입니다:

- 안전하지 않은 러스트: 어떻게 러스트가 보장하는 것들로부터 손을 떼고 수동으로 이러한 보장들을 책임질 수 있는가에 대하여
- 고급 라이프타임: 복잡한 라이프타임 상황에 대한 문법
- 고급 트레이트: 연관 타입, 기본 타입 파라미터, 완전 정규화 (fully qualified) 문법, 슈퍼트레이트 (supertrait), 그리고 트레이트와 관련된 신종 패턴
- 고급 타입: 신종 타입 패턴, 타입 별칭 (alias), *never* 타입, 동적 크기 조절 타입에 대한 더 많은 것들
- 고급 함수 및 클로저: 함수 포인터와 클로저 반환하기

모두를 위한 것을 갖춘 러스트 기능들의 모음입니다! 뛰어들어 봅시다!

안전하지 않은 러스트

우리가 여지껏 논해온 모든 코드들은 컴파일 타임에 강제되는 러스트의 메모리 안전성 보장을 갖습니다. 그러나, 러스트는 이러한 메모리 안전성 보장을 강제하지 않는 숨겨진 내부의 두번째 언어를 갖고 있습니다: 이것을 **안전하지 않은 러스트 (unsafe Rust)**라고 부르며 그저 보통의 러스트와 비슷하게 동작하지만, 우리에게 추가적인 슈퍼파워를 제공합니다.

안전하지 않은 러스트는 정적 분석이 선천적으로 보수적이기 때문에 존재합니다. 컴파일러가 어떤 코드에 대한 안전성을 보장하는지 혹은 아닌지를 결정하는 시도를 할 때, 유효하지 않은 프로그램을 허용하는 것보다는 유효한 프로그램을 불허하는 편이 더 낫습니다. 그 코드가 괜찮았을지도라도, 러스트가 그렇게 말할 수 있을 때까지는 괜찮은게 아닙니다! 이러한 경우, 우리는 컴파일러에게 “날 믿어, 내가 뭘 하고 있는지 알고 있어”라고 말하기 위해서 안전하지 않은 코드를 이용할 수 있습니다. 이것의 단점이라면 우리가 고스란히 위험성은 떠안고 이를 사용해야 한다는 점입니다: 만일 안전하지 않은 코드를 부정확하게 사용한다면, 널 포인터 역참조와 같은 메모리 불안전성으로 인한 문제가 발생할 수 있습니다.

러스트가 안전하지 않은 또 다른 자아를 갖고 있는 또 하나의 이유는 밑바탕이 되는 컴퓨터 하드웨어가 선천적으로 안전하지 않기 때문입니다. 만일 러스트가 안전하지 않은 연산을 허용하지 않았다면, 우리는 특정한 작업을 수행할 수 없었을 겁니다. 러스트는 우리가 저수준의 시스템 프로그래밍, 예를 들면 운영체제와 직접 상호작용을 하거나 심지어 우리만의 운영체제를 작성하는 등을 하는 것을 허용하고 싶어합니다. 저수준의 시스템 프로그래밍 작업은 이 언어의 목표 중 하나입니다. 안전하지 않은 러스트를 가지고 무엇을 할 수 있으며 또 어떻게 하는지에 대해서 탐구해봅시다.

안전하지 않은 슈퍼파워

안전하지 않은 러스트로 전환하기 위해서는 **unsafe** 키워드를 이용하며, 그 다음 안전하지 않은 코드를 감싸주는 새 블록을 시작합니다. 우리는 안전하지 않은 러스트 내에서 4개의 행동을 할 수 있는데, 이를 **안전하지 않은 슈퍼파워**라고 부르며, 안전한 러스트 내에서는 할 수 없는 것들입니다. 이 슈퍼파워들은 다음과 같은 것들을 하는 능력입니다:

- 로우 포인터 (raw pointer) 를 역참조하기
- 안전하지 않은 함수 혹은 메소드 호출하기
- 가변 정적 변수 (mutable static variable) 의 접근 혹은 수정하기
- 안전하지 않은 트레잇 구현하기

unsafe 가 빌림 검사기 혹은 다른 어떤 러스트의 안전성 검사 기능을 끄는 게 아니라는 것을 이해하는 것은 중요합니다: 만일 여러분이 안전하지 않은 코드 내에서 참조자를 이용한다면, 이것은 여전히 검사될 것입니다. **unsafe** 키워드는 메모리 안전성을 위해 컴파일러에 의해 검사될 수 없는 위의 네가지 기능을 사용할 수 있는 능력만을 제공할 뿐입니다. 안전하지 않은 블록 내에서도 우리는 여전히 어느 정도의 안전성을 갖습니다.

더불어 `unsafe` 는 블록 내의 코드가 필연적으로 위험하다던가 절대적으로 메모리 안전성 문제를 가지고 있음을 의미하는 것이 아닙니다: 그 의도는 `unsafe` 블록 내의 코드가 올바른 방법으로 메모리에 접근할 것임을 우리가 프로그래머로서 확실히 해두는 것입니다.

사람은 실수를 할 수 있고, 실수는 일어날 것이지만, 위의 네 가지 안전하지 않은 연산이 `unsafe` 이라고 명시된 블록 내에 있도록 요구함으로써 우리는 메모리 안전성과 관련된 어떠한 에러라도 틀림없이 `unsafe` 블록 내에 있을 것임을 알게 될 것입니다. `unsafe` 블록을 작게 유지하세요; 후에 여러분이 메모리 버그를 찾으나갈 때 감사함을 느낄 것입니다.

안전하지 않은 코드를 최대한 격리하기 위해서는 안전하지 않은 코드를 안전한 추상화 내에 있도록 감싸서 안전한 API를 제공하는 것이 최상인데, 이는 우리가 이 장의 뒷편에서 안전하지 않은 함수와 메소드를 시험해 볼 때 다루겠습니다. 표준 라이브러리의 일부분은 검사가 수행된 안전하지 않은 코드 위에 안전한 추상화로 구현되어 있습니다. 안전한 추상화로 안전하지 않은 코드를 감싸는 것은 여러분 혹은 여러분의 사용자가 `unsafe` 코드로 구현된 기능을 이용하고자 하는 모든 장소에 `unsafe` 라고 쓰는 것을 방지할 수 있는데, 안전한 추상화 코드를 사용하는 것은 안전하기 때문입니다.

네 가지 안전하지 않은 슈퍼파워 각각을 차례로 살펴봅시다: 또한 안전하지 않은 코드에 대한 안전한 인터페이스를 제공하는 몇몇 추상화도 살펴볼 것입니다.

로우 포인터를 역참조하기

4장의 “댕글링 참조자”절에서 우리는 참조자들이 언제나 유효함을 컴파일러가 보장한다고 언급했습니다. 안전하지 않은 러스트는 **로우 포인터** (*raw pointer*) 라고 불리는 참조자와 유사한 두 가지 새로운 타입을 갖습니다. 참조자를 이용하는 것처럼 로우 포인터도 불변 혹은 가변이 될 수 있으며 각각 `*const T` 와 `*mut T` 라고 씁니다. 이 애스터리스크는 역참조 연산자가 아닙니다; 이것은 타입 이름의 일부입니다. 로우 포인터의 맥락 내에서 “불변”이란 해당 포인터가 역참조된 후에 직접 대입될 수 없음을 의미합니다.

참조자나 스마트 포인터와는 다르게, 아래와 같은 로우 포인터의 성질을 명심하세요:

- 로우 포인터는 빌림 규칙 무시가 허용되어 불변 및 가변 포인터 양쪽 모두를 갖거나 같은 위치에 여러 개의 가변 포인터를 갖을 수 있습니다.
- 로우 포인터는 유효한 메모리를 가리키고 있음을 보장하지 않습니다.
- 로우 포인터는 널이 될 수 있습니다.
- 로우 포인터는 자동 메모리 정리가 구현되어 있지 않습니다.

러스트가 이러한 보장을 강제하도록 하는 것으로부터 손을 떼도록 함으로써, 우리는 보장된 안전성을 포기하고, 개선된 성능이나 러스트의 보장이 적용되지 않는 타 언어 혹은 하드웨어와의 상호작용 능력을 얻는 기회 비용을 얻을 수 있습니다.

Listing 19-1은 참조자로부터 불변 및 가변 로우 포인터를 만드는 방법을 보여줍니다.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Listing 19-1: 참조자로부터 로우 포인터 생성하기

이 코드에서 `unsafe` 키워드를 포함하지 않았음을 주목하세요. 우리는 로우 포인터를 안전한 코드 내에서 생성할 수 있습니다; 여러분이 잠시 후에 보게될 것처럼, 우리는 그저 안전하지 않은 블록 밖에서는 로우 포인터를 역참조할 수 없을 뿐입니다.

우리는 불변 및 가변 참조자를 관련된 로우 포인터 타입으로 캐스팅하기 위해 `as`를 사용함으로써 로우 포인터를 생성하였습니다. 우리가 유효성이 보장된 참조자로부터 직접 이것들을 만들었기 때문에, 우리는 이 특정한 로우 포인터가 유효함을 알지만, 임의의 로우 포인터에 대해서는 이러한 가정을 내릴 수 없습니다.

다음으로, 우리가 유효성을 특정할 수 없는 로우 포인터를 만들어 보겠습니다. Listing 19-2는 메모리 내에 임의의 위치를 가리키는 로우 포인터를 만드는 방법을 보여줍니다. 임의의 메모리를 사용 시도하는 것은 정의되어 있지 않습니다: 해당 주소에 데이터가 있을 수도 있고 없을 수도 있으며, 컴파일러가 코드를 최적화해서 메모리 접근이 없을 수도, 혹은 프로그램이 세그먼테이션 폴트 (segmentation fault) 에러를 일으킬지도 모릅니다. 보통은 이러한 코드를 작성할 어떠한 좋은 이유도 없지만, 가능은 합니다:

```
let address = 0x012345usize;
let r = address as *const i32;
```

Listing 19-2: 임의의 메모리 주소를 가리키는 로우 포인터 생성하기

우리가 안전한 코드 내에서 로우 포인터를 생성할 수는 있지만, 로우 포인터를 역참조하여 해당 포인터가 가리키고 있는 데이터를 읽지는 못함을 상기하세요. Listing 19-3에서는 `unsafe` 블록을 필요로 하는 로우 포인터에 상에서의 역참조 연산자 `*`를 사용합니다.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Listing 19-3: `unsafe` 블록 내에서 로우 포인터 역참조하기

포인터를 생성하는 것은 어떠한 해도 끼지지 않습니다; 문제는 우리가 이 포인터가 가리키는 값에 접근을 시

도하여 유효하지 않은 값을 다루는 상황에 처할지도 모를 때입니다.

또한 Listing 19-1과 19-3에서 우리가 `num`이 저장되어 있는 동일한 메모리 장소를 가리키고 있는 `*const i32` 와 `*mut i32` 로우 포인터를 생성했음을 주목하세요. 만일 우리가 대신 `num`에 대한 불변 및 가변 참조자를 생성 시도했다면, 러스트의 소유권 규칙이 가변 참조자 와 불변 참조자를 동시에 허용하지 않기 때문에 코드는 컴파일 되지 않을 것입니다. 로우 포인터를 이용하면, 우리는 동일한 위치를 가리키는 가변 포인터 및 불변 포인터를 만들 수 있고, 가변 포인터를 통해 데이터를 바꿀수 있는데, 이는 데이터 레이스를 야기할 가능성이 있습니다. 조심하세요!

이러한 모든 위험을 가지고, 왜 우리는 로우 포인터를 사용하게 될까요? 한가지 주요 사용례는 여러분이 다음 절에 “안전하지 않은 함수 혹은 메소드 호출하기”에서 보실 것과 같이, C 코드와의 상호작용을 할 때입니다. 또다른 경우는 빌림 검사기가 이해하지 못하는 안전한 추상화를 만들 때입니다. 우리는 안전하지 않은 함수를 소개한 다음 안전하지 않은 코드를 사용하는 안전한 추상화의 예를 살펴보겠습니다.

안전하지 않은 함수 혹은 메소드 호출하기

안전하지 않은 블록을 필요로하는 연산의 두번째 타입은 안전하지 않은 함수의 호출입니다. 안전하지 않은 함수와 메소드는 보통의 함수와 메소드와 똑같이 생겼지만, 함수 정의의 앞부분에 추가적으로 `unsafe` 가 붙어 있습니다. 이 맥락 내에서의 `unsafe` 키워드는 우리가 이 함수를 호출할 때 우리가 유지시키고 싶어하는 요구사항을 가지고 있음을 나타내는데, 이는 우리가 이러한 요구사항을 만족시키는지를 러스트가 보장할 수 없기 때문입니다. `unsafe` 블록 내에서 안전하지 않은 함수를 호출함으로써, 우리가 이 함수의 문서를 읽었고 함수의 계약서를 준수할 책임을 가지고 있다고 말하는 것입니다.

아래는 본체에서 아무것도 하지 않는 `dangerous`라는 이름의 안전하지 않은 함수입니다:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

우리는 반드시 분리된 `unsafe` 블록 내에서 `dangerous`를 호출해야 합니다. 만일 `unsafe` 블록 없이 `dangerous`의 호출을 시도하면, 다음과 같은 에러를 얻게 됩니다:

```
error[E0133]: call to unsafe function requires unsafe function or block
-->
|
4 |     dangerous();
|     ^^^^^^^^^^ call to unsafe function
```

우리의 `dangerous` 호출 주변에 `unsafe` 블록을 집어넣음으로서, 우리는 이 함수의 문서를 읽었고, 이를

어떻게 적절히 이용하는지 이해했으며, 이 함수의 개약서에 서명하는 것임을 확인했음을 러스트에게 단언하는 중입니다.

안전하지 않은 함수의 본체는 사실상 `unsafe` 블록이므로, 안전하지 않은 함수 내에서 다른 안전하지 않은 연산을 수행하기 위해서 별도의 `unsafe` 블록을 추가할 필요는 없습니다.

안전하지 않은 코드 상에 안전한 추상화 생성하기

어떤 함수가 단지 안전하지 않은 코드를 담고 있다는 것이 함수 전체를 안전하지 않은 것으로 표시할 필요가 있음을 뜻하지는 않습니다. 사실, 안전한 함수 내에 안전하지 않은 코드를 감싸는 것은 일반적인 추상화입니다. 한가지 예로, 표준 라이브러리가 제공하는 함수 `split_at_mut`를 공부해봅시다. 이 함수는 몇몇 안전하지 않은 코드를 필요로 하고 우리가 어떻게 구현할 수 있을지 탐구해볼만 합니다. 이 안전한 메소드는 가변 슬라이스 상에서 정의됩니다: 이것은 하나의 슬라이스를 취해서 인자로 주어진 인덱스에서 슬라이스를 쪼개서 둘로 만들어줍니다. Listing 19-4는 `split_at_mut`를 사용하는 방법을 보여줍니다.

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

Listing 19-4: 안전한 `split_at_mut` 함수의 사용

안전한 러스트만 사용해서는 이 함수를 구현할 수 없습니다. 그 시도는 Listing 19-5와 같은 형태처럼 되겠으나, 컴파일되지 않을 것입니다. 단순하게 하기 위해서, 우리는 `split_at_mut`를 메소드가 아닌 함수로서 구현하고 제네릭 타입 `T`의 슬라이스를 위한 것보다는 `i32` 값의 슬라이스를 위한 것으로 구현하겠습니다.

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

Listing 19-5: 안전한 러스트만 사용하여 `split_at_mut`를 구현하는 시도

이 함수는 먼저 슬라이스의 총 길이를 얻은 다음, 매개변수로 주어진 인덱스가 총 길이보다 작거나 같음을 검사함으로서 슬라이스 내에 있음을 단언(assert)합니다. 이 단언은 우리가 넘긴 인덱스가 슬라이스를 쪼개기

위한 인덱스보다 클 경우, 이 함수가 그 인덱스의 사용 시도를 하기 전에 패닉을 일으킬 것임을 의미합니다.

그 다음 우리는 두 개의 가변 슬라이스를 튜플 안에 넣어 반환합니다: 하나는 원본 슬라이스의 시작부터 `mid` 인덱스까지이고 다른 하나는 `mid`부터 원본 슬라이스의 끝까지입니다.

Listing 19-5의 코드의 컴파일을 시도하면, 다음과 같은 에러를 얻습니다:

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
-->
|
6 |     (&mut slice[..mid],
|         ----- first mutable borrow occurs here
7 |     &mut slice[mid..])
|         ^^^^^^ second mutable borrow occurs here
8 | }
| - first borrow ends here
```

러스트의 빌림 검사기는 우리가 슬라이스의 서로 다른 부분을 빌리는 중임을 이해할 수 없습니다; 러스트는 우리가 같은 슬라이스로부터 두번 빌리는 중인것만을 알고 있습니다. 슬라이스의 서로 다른 부분을 빌리는 것은 이 두 슬라이스가 서로 겹치지 않기 때문에 근본적으로 괜찮지만, 러스트는 이를 알 정도로 똑똑하진 않습니다. 우리가 이 코드가 괜찮은 것임을 알지만 러스트는 그렇지 못하므로, 안전하지 않은 코드를 이용할 시간입니다.

Listing 19-6은 `split_at_mut`의 구현체가 동작하도록 만들기 위해서 `unsafe` 블록, 로우 포인터, 그리고 몇몇 안전하지 않은 함수의 호출을 사용하는 방법을 보여줍니다.

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

Listing 19-6: `split_at_mut` 함수의 구현체 내에서 안전하지 않은 코드 사용하기

4장의 “슬라이스 타입”절에서 슬라이스는 어떤 데이터를 가리키는 포인터와 슬라이스의 길이로 되어있음을 상기하세요. 우리는 `len` 메소드를 사용하여 슬라이스의 길이를 얻고 `as_mut_ptr` 메소드를 사용하여 슬라이스의 로우 포인터에 접근합니다. 위의 경우, 우리가 `i32` 값들의 가변 슬라이스를 갖고 있으므로,

`as_mut_ptr`은 `*mut i32` 타입을 갖는 로우 포인터를 반환하는데, 이는 `ptr` 변수에 저장됩니다.

`mid` 인덱스가 슬라이스 내에 있다는 단어는 유지합니다. 그 다음 안전하지 않은 코드에 왔습니다:

`slice::from_raw_parts_mut` 함수는 로우 포인터와 길이를 받아서 슬라이스를 생성합니다. 이 함수를 이용하여 `ptr`로 시작하고 `mid` 만큼의 아이템을 가진 슬라이스를 생성합니다. 그다음 우리는 `ptr` 상에서 `offset` 메소드를 인자 `mid`와 함께 호출하여 `mid`에서부터 시작하는 로우 포인터를 얻고, 이 포인터와 `mid` 뒤에 남은 아이템의 개수를 길이로 하는 슬라이스를 생성합니다.

함수 `slice::from_raw_parts_mut`는 로우 포인터를 인자로 사용하고 이 포인터가 유효함을 반드시 믿어야 하므로 안전하지 않습니다. 로우 포인터의 `offset` 메소드 또한 안전하지 않은데, 그 이유는 오프셋 위치 또한 유효한 포인터임을 반드시 믿어야 하기 때문입니다. 따라서, 이들을 호출할 수 있도록 하기 위해 우리의 `slice::from_raw_parts_mut`와 `offset` 호출 주변에 `unsafe` 블록을 넣어야 했습니다. 이 코드를 살펴보고 `mid`가 반드시 `len`보다 작거나 같다는 단언을 추가함으로써, 우리는 `unsafe` 블록 내에서 사용된 모든 로우 포인터들이 슬라이스 내의 데이터를 가리키는 유효한 포인터가 될 것임을 말할 수 있습니다. 이는 받아들일만하고 `unsafe`의 적절한 사용입니다.

결과적으로 나온 `split_at_mut` 함수를 `unsafe`로 표시할 필요가 없으며, 이 코드를 안전한 리스트로부터 호출할 수 있음을 주목하세요. 우리는 `unsafe` 코드를 안전한 방법으로 사용하는 함수의 구현체를 가지고 안전하지 않은 코드에 대한 안전한 추상화를 만들었는데, 이는 이 함수가 접근하는 데이터로부터 오직 유효한 포인터만을 생성하기 때문입니다.

반면, Listing 19-7의 `slice::from_raw_parts_mut` 사용은 슬라이스에 사용될 때 크래시를 일으키기 쉽습니다. 이 코드는 임의의 메모리 위치를 얻어서 만개의 아이템 길이를 갖는 슬라이스를 생성합니다:

```
use std::slice;

let address = 0x012345usize;
let r = address as *mut i32;

let slice = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

Listing 19-7: 임의의 메모리 위치로부터 슬라이스 생성하기

우리는 이 임의의 위치에서 메모리를 소유하지 않았으며, 이 코드가 만들어낸 슬라이스가 유효한 `i32` 값을 담고 있음에 대한 보장은 없습니다. `slice`를 마치 유효한 슬라이스인 것처럼 사용하는 시도는 정의하지 않은 동작 (undefined behaviour) 을 야기합니다.

extern 함수를 사용하여 외부 코드 호출하기

가끔, 여러분의 러스트 코드는 다른 언어로 작성된 코드와 상호작용하고 싶어할지도 모릅니다. 이를 위해서 러스트는 외국 함수 인터페이스 (*Foreign Function Interface, FFI*) 의 생성과 사용을 가능케 하는 `extern` 키워드를 가지고 있습니다. FFI는 프로그래밍 언어가 함수를 정의하고 다른 (외국의) 프로그래밍 언어가 해당 함수를 호출 가능하게 하는 방법입니다.

Listing 19-8은 C 표준 라이브러리의 `abs` 함수와의 통합을 설정하는 방법을 보여줍니다. `extern` 블록 내에 선언된 함수는 언제나 러스트 코드로부터 호출하기에 안전하지 않습니다. 그 이유는 타 언어들이 러스트의 규칙과 보장들을 강제하지 않으며, 러스트가 이들을 검사할 수도 없으므로, 따라서 안전성을 보장하기 위한 책임은 프로그래머에게 떨어집니다.

Filename: src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

Listing 19-8: 다른 언어에 정의된 `extern` 함수의 선언 및 호출

`extern "C"` 블록 내에서, 우리가 호출하고 싶은 다른 언어로부터 온 외부 함수의 이름과 시그니처를 나열합니다. `"C"` 부분은 해당 외부 함수가 어떤 ABI (*application binary interface*) 를 사용하는지를 정의합니다: ABI는 어셈블리 수준에서 함수를 어떻게 호출하는지를 정의합니다. `"C"` ABI는 가장 흔하며 C 프로그래밍 언어의 ABI를 준수합니다.

다른 언어로부터 러스트 함수 호출하기

우리는 또한 `extern` 을 사용하여 다른 언어들이 러스트 함수를 호출할 수 있도록 하는 인터페이스를 만들 수 있습니다. `extern` 블록 대신, `fn` 키워드 전에 `extern` 키워드를 추가하고 사용할 ABI를 명시합니다. 우리는 또한 `#[no_mangle]` 어노테이션을 추가하여 러스트 컴파일러가 이 함수의 이름을 맹글링하지 않도록 할 필요가 있습니다. 맹글링 (*mangling*) 이란 우리가 함수에게 준 이름을 컴파일 과정의 다른 부분에서 사용하기 위한 더 많은 정보를 담고 있지만 사람이 읽기엔 별로 안 좋은 이름으로 컴파일러가 바꾸는 과정입니다. 모든 프로그래밍 언어 컴파일러가 약간씩 다르게 이름을 맹글링 하므로, 러스트 함수가 다른 언어에 의해 이름을 불릴 수 있도록 하기 위해, 우리는 반드시 러스트 컴파일러의 이름 맹글링 기능을 꺼야 합니다.

아래의 예제에서, 우리는 `call_from_c` 를 공유 라이브러리로 컴파일하고 C로 링크한 다음, 이 함수

를 C 코드에서 접근 가능하게 만들었습니다:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

이러한 `extern`의 사용에는 `unsafe`가 필요 없습니다.

가변 정적 변수의 접근 혹은 수정하기

지금까지 우리는 전역 변수 (*global variable*)에 대하여 이야기한 적이 없는데, 이는 러스트가 지원하기는 하지만 러스트의 소유권 규칙에 문제를 일으킬 수 있습니다. 만일 두 스레드가 동일한 가변 전역 변수에 접근하는 중이라면, 이는 데이터 레이스를 야기할 수 있습니다.

러스트에서 전역 변수는 정적 (*static*) 변수라고 불립니다. Listing 19-9는 스트링 슬라이스를 값으로 갖는 정적 변수의 정의 및 사용의 예를 보여줍니다.

Filename: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

Listing 19-9: 불변 정적 변수의 정의 및 사용

정적 변수는 상수와 유사한데, 이는 우리가 3장의 “변수와 상수의 차이점” 절에서 논의했었습니다. 정적 변수의 이름은 관례에 따라 `SCREAMING_SNAKE_CASE` 형식을 따르며, 우리는 반드시 변수의 타입을 명시해야 하는데, 위의 예제에서는 `&'static str`입니다. 정적 변수는 `'static` 라이프타임을 갖는 참조자만을 저장할 수 있는데, 이는 러스트 컴파일러가 라이프 타임을 알아낼 수 있음을 의미합니다; 우리는 이를 명시적으로 작성할 필요가 없습니다. 불변 정적 변수에의 접근은 안전합니다.

상수와 불변 정적 변수는 비슷해 보일지도 모르겠으나, 정적 변수의 값이 메모리 내의 고정된 주소값을 갖는다는 점에서 미묘한 차이점이 있습니다. 값을 사용하면 언제나 동일한 데이터에 접근하게 될 것입니다. 반면 상수는 사용될 때마다 데이터가 복사되는 것이 허용됩니다.

상수와 정적 변수 간의 또 다른 차이점은 정적 변수가 가변일 수 있다는 점입니다. 가변 정적 변수에 접근하고 수정하는 것은 안전하지 않습니다. Listing 19-10은 `COUNTER`라는 이름의 가변 정적 변수를 선언하고, 접

근하고, 수정하는 방법을 보여줍니다.

Filename: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Listing 19-10: 가변 정적 변수를 읽거나 쓰는 것은 안전하지 않습니다

보통의 변수처럼, 우리는 `mut` 키워드를 사용하여 가변성을 명시합니다. `COUNTER`를 읽거나 쓰는 어떠한 코드라도 `unsafe` 블록 내에 있어야 합니다. 이 코드는 컴파일 되고 우리가 기대한 바와 같이 `COUNTER: 3`을 출력하는데, 그 이유는 이 프로그램이 단일 스레드이기 때문입니다. 여러 스레드가 `COUNTER`에 접근하도록 하는 것은 데이터 레이스를 일으키기 쉽습니다.

전역적으로 접근 가능한 가변 데이터를 이용하는 것은 데이터 레이스가 없음을 확신하기 힘들게 만드는데, 이것이 리스트가 가변 정적 변수를 안전하지 않은 것으로 간주하는 이유입니다. 가능하다면 우리가 16장에서 논의했던 동시성 기술과 스레드-안전한 스마트 포인터를 이용하여, 컴파일러가 서로 다른 스레드로부터 접근되는 데이터가 안전하게 사용됨을 검사하도록 하는 편이 좋습니다.

안전하지 않은 트레이트 구현하기

`unsafe`에서만 동작하는 마지막 기능은 안전하지 않은 트레이트 구현하기입니다. 트레이트는 적어도 메소드 중 하나가 컴파일러가 검사할 수 없는 몇몇 불변성 (invariant)을 갖고 있을 때 안전하지 않게 됩니다. 우리는 `trait` 전에 `unsafe`를 추가함으로써 어떤 트레이트이 `unsafe` 함을 선언할 수 있습니다; 그 다음 트레이트의 구현체 또한 Listing 19-11에서 보는 바와 같이 `unsafe`로 표시되어야 합니다.

```
unsafe trait Foo {  
    // methods go here  
}  
  
unsafe impl Foo for i32 {  
    // method implementations go here  
}
```

Listing 19-11: 안전하지 않은 트레이트의 정의 및 구현

`unsafe impl`을 이용함으로써 우리는 컴파일러가 검증할 수 없는 불변성을 우리가 유지할 것임을 약속하고 있습니다.

한 가지 예로서, 16장의 “`Sync` 와 `Send` 트레이트를 이용한 확장 가능한 동시성” 절에서 논했던 `Sync` 와 `Send` 마커 트레이트를 상기해보세요: 우리의 타입이 전체적으로 `Send` 되고 `Sync` 한 타입으로 구성되어 있다면 컴파일러는 이 트레이트를 자동적으로 구현합니다. 만일 우리가 로우 포인터와 같이 `Send` 되지 않거나 `Sync` 하지 않은 타입을 포함한 타입을 구현하고, 이 타입을 `Send` 되거나 `Sync` 한 것으로 표시하고 싶다면, 우리는 `unsafe`를 이용해야 합니다. 러스트는 우리의 타입이 스레드 사이로 안전하게 보내지거나 여러 스레드로부터 안전하게 접근되는 것에 대한 보장을 유지하는 것을 검사할 수 없습니다; 따라서, 우리는 손수 이를 검사하고 `unsafe`를 이용하여 이러한 사항을 나타낼 필요가 있습니다.

언제 안전하지 않은 코드를 이용할까요?

방금까지 논했던 네 가지 행동 (슈퍼파워) 을 얻기 위해 `unsafe`를 사용하는 것은 잘못된 것도 아니고, 심지어 눈살을 찌푸릴 일도 아닙니다. 하지만 `unsafe` 코드를 올바르게 이용하는 것은 좀 더 힘든데 그 이유는 컴파일러가 메모리 안전성을 유지하는데 도움을 줄 수 없기 때문입니다. 여러분이 `unsafe` 코드를 사용할 이유를 갖게 될 때, 여러분은 그렇게 할 수 있고, 명시적인 `unsafe` 어노테이션을 갖는 것이 문제가 일어났을 때 그 근원을 추적해 나가는 것을 더 수월하게 만들어 줍니다.

고급 라이프타임

10장의 “라이프타임을 이용한 참조자 유효화”절에서, 여러분은 러스트에게 서로 다른 참조자의 라이프타임이 어떻게 연관되는지를 알려주기 위하여 참조자에 대한 라이프타임 파라미터의 명시 방법을 배웠습니다. 여러분은 모든 참조자가 라이프타임을 갖지만, 거의 대부분의 경우 러스트가 어떻게 이 라이프타임을 생략시켜 주는지도 봤습니다. 이제 우리는 아직 다루지 못했던 라이프타임의 세가지 고급 기능을 살펴볼 것입니다:

- 라이프타임 서브타이핑 (subtyping): 한 라이프타임이 다른 라이프타임보다 오래 사는 것을 보장하기
- 라이프타임 바운드: 제네릭 타입을 가리키는 참조자를 위한 라이프타임 명시하기
- 트레이트 객체 라이프타임의 추론: 컴파일러는 어떻게 트레이트 객체의 라이프타임을 추론하며 언제 이들을 명시할 필요가 있는지에 대하여

라이프타임 서브타이핑은 하나의 라이프타임이 다른 것보다 오래 사는 것을 보장합니다

라이프타임 서브타이핑은 하나의 라이프타임이 다른 라이프타임보다 오래 살아야 함을 명시합니다. 라이프타임 서브타이핑을 탐구하기 위해서, 우리가 파서를 작성하길 원한다고 상상해 보세요. 우리가 파싱하는 중인 스트링에 대한 참조자를 가지고 있는 **Context**라는 이름의 구조체를 사용하겠습니다. 이 스트링을 파싱하고 성공 혹은 실패를 반환하는 파서를 작성할 것입니다. 이 파서는 파싱을 하기 위해 **Context**를 빌릴 필요가 있을 것입니다. Listing 19-12는 이 파서 코드를 구현한 것인데, 필요한 라이프타임 명시가 제외되어 있고, 따라서 컴파일되지 않습니다.

Filename: src/lib.rs

```
struct Context(&str);

struct Parser {
    context: &Context,
}

impl Parser {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-12: 라이프타임 명시 없이 파서를 정의하기

이 코드를 컴파일하면 에러를 내는데 그 이유는 러스트가 **Context**의 스트링 슬라이스와 **Parser** 내의 **Context**의 참조자에 대한 라이프타임 파라미터를 기대하기 때문입니다.

단순함을 위해서, 이 **parse** 함수는 **Result<(), &str>**를 리턴합니다. 즉, 이 함수는 성공시에 아무것도

하지 않고, 실패시에는 파싱이 올바르기 되지 않은 스트링 슬라이스 부분을 반환할 것입니다. 실제 구현은 더 많은 에러 정보를 제공하고 파싱이 성공하면 구조화된 데이터 타입을 반환할 것입니다. 우리는 이러한 상세 부분은 다루지 않을 것인데, 이 예제의 라이프타임 부분과는 관련이 없기 때문입니다.

이 코드를 계속 단순하게 유지하기 위해, 우리는 어떠한 파싱 로직도 작성하지 않고 있습니다. 하지만, 유효하지 않은 입력을 처리하기 위하여 파싱 로직의 어딘가에서 잘못된 입력 부분을 참조하는 에러를 반환하기란 매우 가능성이 큽니다; 이 참조자가 코드 예제를 라이프타임에 대한 관점에서 흥미롭게 만들어주는 것입니다. 우리 파서의 로직이 첫번째 바이트 이후의 입력은 유효하지 않다고 판단했다고 가정해봅시다. 첫번째 바이트가 유효한 문자 범위 상에 있지 않으면 이 코드는 패닉을 일으킬 수도 있음을 주의하세요; 다시 한번, 우리는 수반되는 라이프타임에 집중하도록 예제를 단순화하는 중입니다.

이 코드를 컴파일하기 위해서는 `Context` 내의 스트링 슬라이스와 `Parser` 내의 `Conext`를 가리키는 참조자에 대한 라이프타임 파라미터를 채워줄 필요가 있습니다. 이를 위한 가장 직관적인 방법은 Listing 19-13에서 보시는 것과 같이 모든 곳에 동일한 라이프타임 이름을 사용하는 것입니다. 10장의 “구조체 정의 상에서의 라이프타임 명시”절에서 본 것처럼 각각의 `struct Context<'a>`, `struct Parser<'a>`와 `impl<'a>`는 새로운 라이프타임 파라미터를 선언중이라는 점을 상기하세요. 그 이름들이 모두 동일하게 등장한 반면, 예제에서 선언된 이 3개의 라이프타임 파라미터는 모두 연관되어 있지 않습니다.

Filename: src/lib.rs

```
struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-13: `Context`와 `Parser`의 모든 참조자에 라이프타임 파라미터 명시하기

이 코드는 잘 컴파일됩니다. 이 코드는 러스트에게 `Parser`가 라이프타임이 '`'a`'인 `Context`를 가리키는 참조자를 가지고 있고, `Context`는 `Parser` 내의 `Context` 참조자만큼 오래 사는 스트링 슬라이스를 가지고 있다고 말해줍니다. 러스트의 컴파일러 에러 메세지는 이 참조자들에게 라이프타임 파라미터가 필요하다고 기술했었고, 우리가 방금 그 라이프타임 파라미터를 추가했습니다.

다음으로, Listing 19-14에서 우리는 `Context`의 인스턴스를 받아서, 이 콘텍스트를 파싱하기 위해 `Parser`를 사용하고, `parse`가 반환하는 것을 반환하는 함수를 추가할 것입니다. 아래 코드는 잘 동작하지 않습니다:

Filename: src/lib.rs

```
fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}
```

Listing 19-14: `Context`를 받아서 `Parser`를 사용하는 `parse_context` 함수 추가 시도

`parse_context` 함수를 추가하고 컴파일 시도를 하면 두 개의 장황한 에러를 얻게 됩니다:

```
error[E0597]: borrowed value does not live long enough
--> src/lib.rs:14:5
|
14 |     Parser { context: &context }.parse()
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ does not live long enough
15 | }
| - temporary value only lives until here
|
note: borrowed value must be valid for the anonymous lifetime #1 defined on
the function body at 13:1...
--> src/lib.rs:13:1
|
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
| |_ ^
|
error[E0597]: `context` does not live long enough
--> src/lib.rs:14:24
|
14 |     Parser { context: &context }.parse()
|     ^^^^^^^ does not live long enough
15 | }
| - borrowed value only lives until here
|
note: borrowed value must be valid for the anonymous lifetime #1 defined on
the function body at 13:1...
--> src/lib.rs:13:1
|
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
| |_ ^
```

이 에러들은 만들어진 `Parser` 인스턴스와 `context` 파라미터가 `parse_context` 함수의 끝까지만 산다고 기술하고 있습니다. 그러나 이 둘 모두 함수의 전체 라이프타임보다 더 살아야 할 필요가 있습니다.

바꿔 말하면, `Parse` 와 `context` 는 전체 함수보다 오래 살아야 할 필요가 있고 이 코드의 모든 참조자들이 항상 유효하기 위해서 함수가 끝날 때는 물론 함수가 시작될 때도 유효해야 할 필요가 있습니다. 우리가 만든 `Parser` 와 `context` 파라미터는 함수 끝에서 스코프 밖으로 벗어나는데, 그 이유는 `parse_context` 가 `context` 의 소유권을 갖기 때문입니다.

이 에러가 왜 발생하는지 알아내기 위해, Listing 19-13에 있는 정의 부분 중 특히 `parse` 메소드의 시그니처에 있는 참조자들을 다시 살펴봅시다:

```
fn parse(&self) -> Result<(), &str> {
```

생략 규칙 기억하시죠? 만일 참조자의 라이프타임을 생략하지 않고 명시했다면, 시그니처는 다음과 같을 것입니다:

```
fn parse<'a>(&'a self) -> Result<(), &'a str> {
```

즉, `parse` 의 반환값의 에러 부분은 `Parser` 인스턴스의 라이프타임에 묶여 있는 라이프타임을 갖고 있다는 것입니다 (`parse` 메소드 시그니처 내의 `&self` 의 것이지요). 이는 타당합니다: 반환되는 스트링 슬라이스는 `Parser` 가 가지고 있는 `Context` 인스턴스의 스트링 슬라이스를 참조하고, `Parser` 구조체의 정의는 `Context` 의 참조자의 라이프타임과 `Context` 가 가지고 있는 스트링 슬라이스의 라이프타임이 동일해야 함을 기술하고 있습니다.

문제는 `parse_context` 함수가 `parse` 로부터 값을 반환하고 있으므로, `parse_context` 의 반환값의 라이프타임 또한 `Parser` 의 라이프타임과 묶여 있다는 것입니다. 그러나 `parse_context` 함수 내에서 만들어진 `Parser` 인스턴스는 함수 끝을 벗어나 살 수 없을 것이고 (일시적인 객체입니다), `context` 는 함수의 끝에서 스코프 밖으로 벗어날 것입니다 (`parse_context` 가 이것의 소유권을 가지고 있습니다).

러스트는 우리가 함수의 끝에서 스코프 밖으로 벗어나는 값의 참조자를 반환 시도를 하는 중이라고 생각하는데, 이는 우리가 모든 라이프타임을 동일한 라이프타임 파라미터로 명시했기 때문입니다. 이 어노테이션은 러스트에게 `Context` 가 가지고 있는 스트링 슬라이스의 라이프타임은 `Parser` 가 들고 있는 `Context` 를 가리키는 참조자의 라이프타임의 것과 동일하다고 말하고 있습니다.

`parse_context` 함수는 `parse` 함수의 내부에서 반환되는 스트링 슬라이스가 `Context` 와 `Parser` 보다 오래살 것이라는 것, 그리고 `parse_context` 가 반환하는 참조자가 `Context` 혹은 `Parser` 가 아닌 스트링 슬라이스를 참조하고 있다는 것을 알 수 없습니다.

`parse` 의 구현체가 무엇을 하는지 아는 것으로써, 우리는 `parse` 의 반환값이 `Parser` 에 묶여있는 유일한 이유가 이것이 스트링 슬라이스를 참조하고 있는 `Parser` 의 `Context` 를 참조하고 있기 때문이라는 것을 알게 되었습니다. 따라서, `parse_context` 가 다루고자 하는 것은 실은 스트링 슬라이스의 라이프타임인 것입니다. 우리는 `Context` 내의 스트링 슬라이스와 `Parser` 내의 `Context` 를 가리키는 참조자가 다른 라이프타임을 가지고 있고 `parse_context` 의 반환값은 `Context` 의 스트링 슬라이스의 라이프타임에 묶

여있음을 알려줄 방법이 필요합니다.

먼저 Listing 19-15에서 보시는 것처럼 **Parser** 와 **Context** 에게 서로 다른 라이프타임 파라미터를 주는 시도를 하겠습니다. 우리는 '**s**' 와 '**c**' 라는 라이프타임 파라미터 이름을 사용하여 어떤 라이프타임이 **Context** 내의 스트링 슬라이스에 포함되고 어떤 라이프타임이 **Parser** 내의 **Context** 를 가리키는 참조자에 포함되는지 명확히 할 것입니다. 이 해결책이 문제를 완전히 해결하지는 않겠지만, 이것이 시작점이라는 점을 주목하세요. 이 소스 코드 수정이 왜 컴파일 시도에 충분치 않은지 살펴보겠습니다.

Filename: src/lib.rs

```
struct Context<'s>(&'s str);

struct Parser<'c, 's> {
    context: &'c Context<'s>,
}

impl<'c, 's> Parser<'c, 's> {
    fn parse(&self) -> Result<(), &'s str> {
        Err(&self.context.0[1..])
    }
}

fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}
```

Listing 19-15: 스트링 슬라이스에 대한 참조자와 **Context**에 대한 참조자에 대해 서로 다른 라이프타임 파라미터 지정하기

우리가 Listing 19-13에서 명시했던 것과 모두 동일한 위치에 있는 참조자의 라이프타임을 명시했습니다. 하지만 이번에는 참조자가 스트링 슬라이스에 포함되는지 혹은 **Context**에 포함되는지 여부에 따라 다른 파라미터를 사용했습니다. 우리는 또한 **parse** 의 반환값의 스트링 슬라이스 부분에도 이것이 **Context** 내의 스트링 슬라이스의 라이프타임에 포함된다는 것을 나타내기 위해서 어노테이션을 추가했습니다.

이제 컴파일 시도를 하면, 다음과 같은 에러를 얻습니다:

```

error[E0491]: in type `&'c Context<'s>`, reference has a longer lifetime than
the data it references
--> src/lib.rs:4:5
 |
4 |     context: &'c Context<'s>,
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^
|
note: the pointer is valid for the lifetime 'c as defined on the struct at
3:1
--> src/lib.rs:3:1
 |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
| |_ ^
note: but the referenced data is only valid for the lifetime 's as defined on
the struct at 3:1
--> src/lib.rs:3:1
 |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
| |_ ^

```

러스트는 '`'c`'와 '`'s`' 사이에 어떠한 관계도 알지 못합니다. 이를 유효화하기 위해, '`'s`' 라이프타임을 가진 `Context` 내의 참조자 데이터는 '`'c`' 라이프타임을 가진 참조자보다 더 오래 산다는 것을 보장하기 위해 제한될 필요가 있습니다. 만일 '`'s`'가 '`'c`'보다 오래 살지 못한다면, `Context`의 참조자가 유효하지 않을 수도 있습니다.

이제 우리는 이 절의 요점을 얻었습니다: 러스트의 기능인 `라이프타임 서브타이핑`은 하나의 라이프타임 파라미터가 최소한 다른 것만큼 오래 산다는 것을 명시합니다. 우리가 라이프타임 파라미터를 선언하는 꺽쇠 괄호 내에서, 우리는 라이프타임 '`'a`'을 평소처럼 선언하고, 문법 '`'b: 'a`'를 사용하여 '`'b`'를 선언함으로써 라이프타임 '`'b`'가 최소 '`'a`' 만큼 오래 산다고 선언할 수 있습니다.

우리의 `Parser` 정의부에서, '`'s`' (스트링 슬라이스의 라이프타임) 가 최소한 '`'c`' (`Context`를 가리키는 참조자의 라이프타임) 만큼 오래 사는 것이 보장됨을 말하기 위해서, 아래와 같이 라이프타임 선언을 변경합니다:

Filename: src/lib.rs

```

struct Parser<'c, 's: 'c> {
    context: &'c Context<'s>,
}

```

이제 `Parser` 내에 있는 `Context`에 대한 참조자와 `Context` 내의 스트링 슬라이스를 가리키는 참조자

는 다른 라이프타임을 갖습니다; 우리는 스트링 슬라이스의 라이프타임이 **Context**를 가리키는 참조자보다 더 오래 살 것이란 보장을 했습니다.

참 길고 지루한 예제였습니다만, 이 장의 첫 부분에서도 언급했듯, 러스트의 고급 기능들은 매우 구체적입니다. 우리가 이 예제에서 묘사한 문법이 자주 필요치는 않겠지만, 특정한 상황에서 여러분이 참조해야 하는 무언가를 참조하는 방법을 알아둬야 할 것입니다.

제네릭 타입에 대한 참조자 상의 라이프타임 바운드

10장의 “트레이트 바운드”절에서, 우리는 제네릭 타입 상의 트레이트 바운드를 사용하는 것에 대해 논했습니다. 우리는 또한 제네릭 타입의 제약사항으로서 라이프타임 파라미터를 추가할 수 있습니다; 이를 **라이프타임 바운드 (lifetime bound)**라고 부릅니다. 라이프타임 바운드는 제네릭 타입 내의 참조자들이 참조하고 있는데 이터보다 오래 살지 못하도록 러스트가 확인하는 것을 돋습니다.

한 가지 예로, 참조자에 대한 래퍼 (wrapper) 인 타입을 고려해보세요. 15장의 “**RefCell<T>**와 내부 가변성 패턴”절에서 나온 **RefCell<T>** 타입을 상기해보세요: 이것의 **borrow** 및 **borrow_mut** 메소드는 각각 **Ref** 및 **RefMut** 타입을 반환합니다. 이 타입들은 런타임에 빌림 규칙을 계속 따르게 하는 참조자들의 레퍼입니다. **Ref** 구조체의 정의는 Listing 19-16과 같은데, 지금은 라이프타임 바운드 없이 쓰였습니다:

Filename: src/lib.rs

```
struct Ref<'a, T>(&'a T);
```

Listing 19-16: 시작을 위해 라이프타임 바운드 없이 쓰는 제네릭 타입에 대한 참조자를 감싼 구조체 정의하기

제네릭 타입 **T**과의 관계에 대한 라이프타임 **'a**의 명시적 제약이 없으면, 러스트는 에러를 내게 되는데 그 이유는 제네릭 타입 **T**가 얼마나 오래 살 것인지를 모르기 때문입니다:

```
error[E0309]: the parameter type `T` may not live long enough
--> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  | ^^^^^^
  |
  = help: consider adding an explicit lifetime bound `T: 'a`...
note: ...so that the reference type `&'a T` does not outlive the data it
points at
--> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  | ^^^^^^
```

`T` 가 어떠한 타입도 될 수 있으므로, `T` 는 참조자 혹은 하나 이상의 참조자를 가지고 있는 타입이 될 수 있는데, 각각은 자신의 라이프타임을 가질 수 있습니다. 러스트는 `T` 가 '`a`' 만큼 오래 살 수 있는지 확신할 수 없습니다.

다행히도, 위의 경우 어려가 라이프타임 바운드를 어떻게 명시하는지에 대한 도움되는 조언을 제공합니다:

```
consider adding an explicit lifetime bound `T: 'a` so that the reference type
`&'a T` does not outlive the data it points at
```

Listing 19-17은 우리가 제네릭 타입 `T` 를 선언할 때 라이프타임 바운드를 명시함으로서 이 조언을 어떻게 적용하는지를 보여줍니다:

```
struct Ref<'a, T: 'a>(&'a T);
```

Listing 19-17: `T` 상의 라이프타임 바운드를 추가하여 `T` 내의 어떠한 참조자들도 최소한 '`a`' 만큼 오래 살 것임을 명시하기

이 코드는 이제 컴파일되는데, `T: 'a` 문법을 사용하면 `T` 가 어떤 타입이든 될 수 있지만, 만일 어떠한 참조자라도 포함하고 있다면, 그 참조자들은 최소한 '`a`' 만큼은 오래 살아야 함을 명시하고 있기 때문입니다.

Listing 19-18의 `StaticRef` 구조체 정의 부분에서 `T` 에 '`static`' 라이프타임 바운드를 추가한 것처럼, 우리는 이 문제를 다른 방식으로 해결할 수도 있습니다. 이는 만일 `T` 가 어떠한 참조자를 가지고 있다면, 이들은 반드시 '`static`' 라이프타임을 가져야 함을 의미합니다.

```
struct StaticRef<T: 'static>(&'static T);
```

Listing 19-18: '`static`' 라이프타임 바운드를 `T` 에 추가하여 `T` 가 오직 '`static`' 참조자만을 갖거나 아무런 참조자도 없도록 제한하기

`'static'` 이 전체 프로그램만큼 오래 살아야 함을 뜻하기 때문에, 아무런 참조자도 없는 타입도 모든 참조자들이 전체 프로그램 만큼 오래 사는 규정을 만족합니다 (왜냐면 아무런 참조자도 없으니까요). 참조자가 충분히 오래 사는지에 대해 염려하는 빌림 검사기를 위하여, 아무런 참조자도 없는 타입과 영원히 사는 참조자들을 가진 타입 간의 실질적 구분은 없습니다: 둘다 그것이 참조하고 있는 것보다 더 짧은 라이프타임을 가진 참조자인지 아닌지를 결정하는 관점에서는 같습니다.

트레이트 객체 라이프타임의 추론

17장의 “서로 다른 타입의 값을 허용하는 트레이트 객체를 사용하기”절에서, 우리는 동적 디스패치를 이용할 수 있게 해주는 참조자 뒤의 트레이트으로 구성된 트레이트 객체를 논했습니다. 우리는 아직 트레이트 객체 내의 트

레잇을 구현한 타입이 자신만의 라이프타임을 가지면 어떤일이 벌어지는지 논하지는 않았습니다. 트레잇 `Red` 와 구조체 `Ball` 를 가지고 있는 Listing 19-19을 고려해보세요. `Ball` 구조체는 참조자를 가지고 있고 (따라서 라이프타임 파라미터를 가지고 있죠) 또한 트레잇 `Red` 를 구현합니다. 우리는 `Ball` 의 인스턴스를 트레잇 객체 `Box<Red>` 로서 사용하기를 원합니다:

Filename: src/main.rs

```
trait Red { }

struct Ball<'a> {
    diameter: &'a i32,
}

impl<'a> Red for Ball<'a> { }

fn main() {
    let num = 5;

    let obj = Box::new(Ball { diameter: &num }) as Box<Red>;
}
```

Listing 19-19: 트레잇 객체와 함께 라이프타임 파라미터를 갖는 타입 사용하기

비록 우리가 아직 `obj` 과 관련된 라이프타임을 명시적으로 적지 않았으나, 이 코드는 예러 없이 컴파일됩니다. 이 코드는 동작하는데 그 이유는 라이프타임과 트레잇 객체가 함께 동작하는 규칙이 있기 때문입니다:

- 트레잇 객체의 기본 라이프타임은 `'static` 입니다.
- `&'a Trait` 혹은 `&'a mut Trait` 을 쓴 경우, 트레잇 객체의 기본 라이프타임은 `'a` 입니다.
- 단일 `T: 'a` 구절을 쓴 경우, 트레잇 객체의 기본 라이프타임은 `'a` 입니다.
- 여러 개의 `T: 'a` 같은 구절들을 쓴 경우, 기본 라이프타임은 없습니다; 우리가 명시적으로 써야합니다.

우리가 명시적으로 써야 할 때, `Box<Red>` 같은 트레잇 객체에 대해 `Box<Red + 'static>` 혹은 `Box<Red + 'a>` 같은 문법을 써서 라이프타임 바운드를 추가할 수 있는데, 이는 참조자가 전체 프로그램 동안 사는지 혹은 그렇지 않은지에 따라 달려 있습니다. 다른 바운드를 사용할 때처럼, 라이프타임 바운드를 추가하는 문법은 타입 내에 참조자를 가진 어떠한 `Red` 트레잇의 구현체라도 그 타입의 참조자처럼 트레잇 객체 내에 명시된 동일한 라이프타임을 가져야 한다는 뜻입니다.

다음으로, 트레잇을 관리하는 다른 고급 기능을 살펴봅시다.

고급 트레이트

우리는 10장의 “트레이트: 공유 동작 정의하기”절에서 먼저 트레이트를 다루었지만, 라이프타임 사용처럼 더 고급 수준의 상세한 내용을 논하지는 않았습니다. 이제 여러분이 러스트에 대해 더 많은 것을 알고 있으니, 우리는 핵심으로 다가갈 수 있습니다.

연관 타입은 트레이트 정의 내에서 플레이스홀더 타입을 명시합니다

연관 타입 (*associated type*)은 타입 플레이스홀더와 트레이트를 연결하여 트레이트 메소드 정의를 할 때 이 플레이스홀더 타입을 시그니처 내에서 이용할 수 있도록 합니다. 트레이트를 구현하는 사람은 이 빈칸의 타입이 특정 구현을 위해 사용될 수 있도록 구체 타입을 명시하게 됩니다. 이러한 방법으로, 우리는 트레이트가 구현되기 전까지 어떠한 타입이 필요한지 정확히 알 필요 없이 임의의 타입을 사용하는 트레이트를 정의할 수 있습니다.

우리는 이 장에서 거의 필요하지 않은 고급 기능의 대부분을 기술했습니다. 연관 타입은 그 중간 어딘가에 있습니다: 이것은 이 책의 나머지 부분에서 설명하는 기능보다 더 희귀하게 사용되지만, 이 장에서 논의하는 많은 수의 다른 기능들보다는 더 흔하게 쓰입니다.

연관 타입을 가진 트레이트의 한 예는 표준 라이브러리에서 제공하는 `Iterator` 트레이트입니다. 그 연관 타입은 `Item`이라는 이름이 붙어있고 `Iterator` 트레이트를 구현하는 타입이 반복하는 값의 타입을 대신합니다. 13장의 “`Iterator` 트레이트와 `next` 메소드”절에서, 우리는 `Iterator` 트레이트의 정의가 Listing 19-20에서 보는 바과 같다고 언급했었습니다.

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Listing 19-20: 연관 타입 `Item`을 가진 `Iterator` 트레이트의 정의

타입 `Item`은 플레이스홀더 타입이고, `next` 메소드의 정의는 `Option<Self::Item>` 타입으로 된 값을 반환할 것임을 보여주고 있습니다. `Iterator` 트레이트를 구현하는 사람은 `Item`의 구체적인 타입을 명시할 것이고, `next` 메소드는 해당하는 구체적 타입의 값을 담고 있는 `Option`을 반환할 것입니다.

연관 타입 vs. 제네릭

연관 타입이 함수를 정의할 때 어떤 타입을 다룰지 특정하지 않고서도 정의할 수 있게 해준다는 점에서, 연관 타입은 제네릭과 유사한 개념같아 보일지도 모르겠습니다. 그럼 왜 연관 타입을 이용할까요?

13장에서 `Counter` 구조체에 대한 `Iterator` 트레이트를 구현했던 예제를 가지고 두 개념 사이의 차이점을 시험해봅시다. Listing 13-21에서, 우리는 `Item` 타입을 `u32`로 명시했었죠:

Filename: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
```

이 문법은 제네릭과 비슷해 보입니다. 그럼 왜 Listing 19-21처럼 그냥 제네릭을 사용하여 `Iterator` 트레이트를 정의하지 않을까요?

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Listing 19-21: 제네릭을 사용한 `Iterator` 트레이트의 가상 정의

그 차이점은 Listing 19-21에서처럼 제네릭을 이용할 경우, 우리는 각 구현마다 타입을 명시해야 한다는 점입니다. 그 이유는 `Iterator<String> for Counter`이나 어떠한 다른 타입도 구현할 수 있는데, 이는 `Counter`에 대한 `Iterator`의 복수 구현을 얻을 수 있게 됩니다. 바꿔 말하면, 트레이트가 제네릭 파라미터를 가지게 될 때, 이것이 하나의 타입에 대해서 매번 제네릭 타입 파라미터의 구체적 타입을 변경해가면서 여러번 구현이 가능해진다는 것입니다. 우리가 `Counter`의 `next` 메소드를 이용할 경우, 우리는 어떤 `Iterator`의 구현체를 이용하고자 하는지를 나타내기 위해 타입 명시를 제공해야만 할 것입니다.

연관 타입을 이용하면 하나의 트레이트에 대해 여러번의 구현을 할 수 없게 되므로 타입 명시를 할 필요가 없어집니다. 연관 타입을 이용하는 Listing 19-20에서의 정의에서, 우리는 `Item`의 타입이 무엇이 될지를 한번만 선택할 수 있는데, 이는 `impl Iterator for Counter`가 한번만 나타나게 될 것이기 때문입니다. 우리는 `Counter`의 `next`를 호출하는 것마다 `u32` 값의 반복자를 요구한다고 명시할 필요가 없습니다.

기본 제네릭 타입 파라미터와 연산자 오버로딩

우리가 제네릭 타입 파라미터를 사용할 때, 해당 제네릭 타입에 대한 기본 구체 타입을 명시할 수 있습니다. 이는 기본 타입이 동작할 경우 트레이트를 구현할 사람이 구체 타입을 명시해야 하는 수고를 덜어줍니다. 제네릭 타입에 대한 기본 타입의 명시 문법은 제네릭 타입을 선언할 때 `<PlaceholderType=ConcreteType>` 꼴입니다.

이 테크닉이 유용한 경우 중 좋은 예가 연산자 오버로딩과 함께 쓰이는 경우입니다. [연산자 오버로딩](#)

(*operator overloading*) 은 특정한 상황에서 (+ 같은) 연산자의 동작을 커스터마이징 하는 것입니다.

러스트는 여러분 만의 연산자를 만들거나 임의의 연산자를 오버로딩하는 것을 허용하지는 않습니다. 하지만 여러분은 `std::ops`에 나열되어 있는 연산자와 연관된 구현하는 것으로서 연산자 및 관련된 트레이트를 오버로딩 할 수 있습니다. 예를 들어, Listing 19-22에서는 두 개의 `Point` 인스턴스를 함께 더하기 위해서 `+` 연산자를 오버로딩 하였습니다. 이는 `Point` 구조체 상에 `Add` 트레이트를 구현하는 것으로 되었습니다:

Filename: src/main.rs

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
              Point { x: 3, y: 3 });
}
```

Listing 19-22: `Point` 인스턴스에 대한 `+` 연산자 오버로딩을 위하여 `Add` 트레이트 구현하기

`add` 메소드는 새로운 `Point`를 생성하기 위해 두 `Point` 인스턴스의 `x` 값과 `y` 값을 각각 더합니다. `Add` 트레이트는 `Output`이라는 연관 타입을 가지고 있는데 이는 `add` 메소드로부터 반환되는 타입을 결정합니다.

이 코드에서 기본 제네릭 타입은 `Add` 트레이트 내에 있습니다. 아래는 이 트레이트의 정의입니다:

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

이 코드가 일반적으로 친숙하게 보여야 합니다: 하나의 메소드와 연관 타입을 가진 트레이트입니다. 새로운 부분은 꺽쇠 괄호 내에 있는 **RHS=Self** 부분입니다: 이 문법을 **기본 타입 파라미터**라고 부릅니다. **RHS** 제네릭 타입 파라미터 (“right hand side”(우변)의 줄임말)은 **add** 메소드의 **rhs** 파라미터의 타입을 정의합니다. 만일 우리가 **Add** 트레이트를 구현할 때 **RHS**의 구체 타입을 지정하지 않는다면, **RHS**의 타입은 기본적으로 **Self**가 될 것인데, 이는 곧 우리가 **Add**를 구현하고 있는 그 타입이 될 것입니다.

Point에 대하여 **Add**를 구현했을 때, 우리는 두 **Point** 인스턴스를 더하고 싶었기 때문에 **RHS**에 대한 기본 타입을 사용했습니다. 기본 타입보다 **RHS** 타입을 커스터마이징 하고 싶은 경우에서의 **Add** 트레이트 구현 예제를 살펴봅시다.

우리는 **Millimeters**와 **Meters**라는, 서로 다른 단위의 값을 가지고 있는 두 개의 구조체를 가지고 있습니다. 우리는 밀리미터 단위의 값과 미터 단위의 값을 더하고 **Add**의 구현체가 변환을 올바르게 하기를 원합니다. Listing 19-23에서 보시는 것처럼, **RHS**로 **Meters**를 사용하여 **Millimeters**에 대한 **Add**의 구현을 할 수 있습니다.

Filename: src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Listing 19-23: **Millimeters**와 **Meters**를 더하기 위해 **Millimeters** 상에 **Add** 트레이트 구현하기

Millimeters와 **Meters**를 더하기 위해, **impl Add<Meters>**라고 명시하여 기본값 **Self** 대신 **RHS** 타입 파라미터를 지정합니다.

우리는 두 가지 주요 방식 내에서 기본 타입 파라미터를 사용합니다:

- 기존 코드를 깨는 일 없이 타입을 확장하기 위해
- 대부분의 유저는 원하지 않을 특정한 상황에 대한 커스터마이징을 허용하기 위해

표준 라이브러리의 **Add** 트레이트는 두 번째 목적에 맞는 예입니다: 보통 여러분은 비슷한 타입 두 개를 더할 것 있지만, **Add** 트레이트는 이를 뛰어넘어서 커스터마이징 할 수 있는 기능을 제공합니다. **Add** 트레이트 정의에 있는 기본 타입 파라미터를 사용한다는 것은 대부분의 경우 여러분이 추가적인 파라미터를 명시할 필요가 없

음을 뜻합니다. 바꿔 말하면, 약간의 구현 보일러 플레이트가 필요 없어서, 트레이트의 구현을 좀 더 간편하게 해준다는 말입니다.

첫번째 목적은 두번째 것과 유사하지만 방향이 반대입니다: 만일 우리가 이미 있던 트레이트에 타입 파라미터를 추가하고자 한다면, 우리가 기존 구현 코드를 깨트리는 일 없이 트레이트의 기능을 확장할 수 있도록 하기 위해 기본 파라미터를 제공할 수 있습니다.

모호성 방지를 위한 완전 정규화 (fully qualified) 문법: 동일한 이름의 메소드 호출하기

러스트에서는 어떤 트레이트이 다른 트레이트의 메소드와 동일한 이름의 메소드를 갖는 것을 방지할 수단이 없고, 두 트레이트을 모두 한 타입에 대해 구현 하는 것을 방지할 방법도 없습니다. 또한 어떤 타입에 대해 트레이트의 메소드와 동일한 이름을 가진 메소드를 직접 구현하는 것도 가능합니다.

동일한 이름의 메소드를 호출할 때, 우리가 어떤 걸 사용하길 원하는지 러스트에게 말해줄 필요가 있습니다.

`fly`라는 이름의 메소드를 가지고 있는 `Pilot`과 `Wizard`라는 두 개의 트레이트을 정의한 Listing 19-24의 코드를 보세요. 그 다음에는 이미 `fly`라는 이름의 메소드를 가지고 있는 `Human` 타입에 대하여 두 트레이트 모두 구현하였습니다. 각각의 `fly` 메소드는 다른 일을 합니다.

Filename: src/main.rs

```

trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}

```

Listing 19-24: `fly` 메소드를 갖도록 정의된 두 트레이트와 `fly` 메소드를 직접 가지고 있는 `Human` 타입 상에서의 해당 트레이트들의 구현

우리가 `Human` 인스턴스 상에서 `fly`를 호출할 때, Listing 19-25에서 보시는 것처럼 컴파일러는 기본적으로 그 타입에 직접 구현된 메소드를 호출합니다.

Filename: src/main.rs

```

fn main() {
    let person = Human;
    person.fly();
}

```

Listing 19-25: `Human` 인스턴스 상에서 `fly` 호출하기

이 코드를 실행시키면 `*waving arms furiously*` 가 출력되는데, 이는 러스트가 `Human` 상에 직접 구현된 `fly` 메소드를 호출했음을 보여줍니다.

Pilot 트레잇 혹은 **Wizard** 트레잇으로부터 **fly** 메소드를 호출하기 위해서는 우리가 어떤 **fly** 메소드를 뜻한 것인지를 특정하기 위하여 좀 더 명시적인 문법을 사용할 필요가 있습니다. Listing 19-26은 이 문법의 예시를 보여줍니다.

Filename: src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Listing 19-26: 호출하길 원하는 트레잇의 **fly** 메소드 특정하기

메소드 이름 앞에 트레잇 이름을 특정하는 것은 우리가 어떤 **fly** 구현체를 호출하고 싶어하는지에 대해서 러스트를 명료하게 해줍니다. 우리는 **Human::fly(&person)**이라고도 작성할 수 있는데, 이는 Listing 19-26에서 사용된 **person.fly()**와 동일한 것이나 모호하지 않기를 원할 경우 좀 더 길게 작성한 것입니다.

이 코드를 실행하면 다음과 같이 출력됩니다:

```
This is your captain speaking.
Up!
*waving arms furiously*
```

fly 메소드가 **self** 파라미터를 쓰므로, 만약 하나의 트레잇을 구현한 두 개의 타입을 가지고 있다면, 러스트는 **self**의 타입에 기초하여 어떤 트레잇의 구현체인지를 알아낼 수 있습니다.

그러나, 트레잇의 일부인 연관 함수는 **self** 파라미터를 가지고 있지 않습니다. 같은 스코프 내의 두 타입이 해당 트레잇을 구현하고 있을 때, 우리가 완전 정규화 문법을 사용하지 않는 이상 러스트는 어떤 타입을 뜻한 것인지를 알아낼 수 없습니다. 예를 들어, Listing 19-27에는 **baby_name**이라는 연관 함수를 가지고 있는 **animal** 트레잇, **Dog** 구조체에 대한 **Animal**의 구현체, 그리고 **Dog**에 바로 정의된 **baby_name** 연관 함수가 있습니다.

Filename: src/main.rs

```

trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}

```

Listing 19-27: 연관 함수를 가지고 있는 트레잇과 이 트레잇을 구현하면서 동시에 동일한 이름의 연관 함수를 가지고 있는 타입

이 코드는 모든 강아지 이름을 스팟 (Spot)이라고 짓길 원하는 동물 보호처를 위한 것인데, 이 이름은 `Dog` 상에 정의된 `baby_name` 연관 함수 내에 구현되어 있습니다. `Dog` 타입은 또한 `Animal` 트레잇을 구현하는데, 이는 모든 동물이 가지는 특성을 기술합니다. 아기 개는 강아지 (puppy)라고 불는데, 이는 `Animal` 트레잇과 연관된 `baby_name` 함수 내에서 `Dog` 상에 `Animal` 트레잇을 구현한 구현체 내에 적혀 있습니다.

`main`에서는 `Dog::baby_name` 함수를 호출했는데, 이는 `Dog`에 직접 정의된 연관 함수를 호출합니다. 이 코드는 다음과 같이 출력합니다:

A baby dog is called a Spot

이 출력은 우리가 원하던게 아니었습니다. 우리는 `Dog` 상에 구현된 `Animal` 트레잇에 속하는 `baby_name` 함수를 호출하여 코드가 `A baby dog is called a puppy`라고 출력하길 원합니다. Listing 19-26에서 사용했던 트레잇 이름 명시 기법이 여기서는 도움이 되지 않습니다; 만일 우리가 `main`을 Listing 19-28의 코드로 변경하면, 컴파일 에러를 얻을 것입니다.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

Listing 19-28: `Animal` 트레이트으로부터의 `baby_name` 함수 호출 시도이지만, 러스트는 어떤 구현체를 사용하는 알지 못합니다

`Animal::baby_name`이 메소드가 아닌 연관 함수이기 때문에, 그런고로 `self` 파라미터가 없기 때문에, 러스트는 `Animal::baby_name`의 어떤 구현체를 우리가 원하는 것인지 알아낼 수 없습니다. 우리는 다음과 같은 컴파일 에러를 얻게 됩니다:

```
error[E0283]: type annotations required: cannot resolve `_: Animal`
--> src/main.rs:20:43
   |
20 |     println!("A baby dog is called a {}", Animal::baby_name());
   |                                     ^^^^^^^^^^^^^^^^^^
   |
   = note: required by `Animal::baby_name`
```

모호성을 방지하고 러스트에게 `Dog`에 대한 `Animal` 구현체를 사용하고 싶다고 알려주기 위해서는 완전 정규화 문법을 사용할 필요가 있는데, 이는 함수를 호출할 때 할 수 있는 한 가장 명시적인 것입니다. Listing 19-29는 완전 정규화 문법을 어떻게 사용하는지를 보여줍니다.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

Listing 19-29: 완전 정규화 문법을 사용하여 `Dog` 상에 고현된 `Animal` 트레이트의 `baby_name` 함수를 호출하고 싶다고 명시하기

우리는 러스트에게 꺼쇠 괄호 내에 타입 명시를 제공하고 있는데, 이는 이번 함수를 호출할 때 `Dog` 타입을 `Animal`처럼 다루길 원한다고 말하는 것으로서 `Dog` 상에 고현된 `Animal` 트레이트의 `baby_name` 메소드를 호출하고 싶음을 나타냅니다. 이제 이 코드는 우리가 원하는 것을 출력할 것입니다:

```
A baby dog is called a puppy
```

일반적으로, 완전 정규화 문법은 다음과 같이 정의됩니다:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

연관 함수에서는 `receiver` 가 없을 것입니다: 즉 다른 인자들의 리스트만 있을 것입니다. 우리는 함수 혹은 메소드를 호출하는 모든 곳에서 완전 정규화 문법을 이용할 수도 있습니다. 그러나, 이 문법 내에서 러스트가 프로그램 내의 다른 정보로부터 알아낼 수 있는 부분은 생략이 허용됩니다. 우리는 이렇게 좀더 장황한 문법을 오직 동일한 이름을 사용하는 여러 개의 구체가 있고 러스트가 이중 어떤 것을 호출하길 원하는지를 식별하기 위해 도움이 필요할 경우만 사용하길 원합니다.

슈퍼트레이트 (supertrait) 을 사용하여 어떤 트레이트 내에서 다른 트레이트의 기능 요구하기

종종, 우리는 어떤 트레이트이 다른 트레이트의 기능을 이용하길 원할런지도 모릅니다. 이런 경우, 우리는 종속된 트레이트이 구현되어 있음에 의존할 필요가 있습니다. 우리가 의존 중인 트레이트이 우리가 구현하는 트레이트의 슈퍼트레이트입니다.

예를 들어, 어떤 값을 애스터리스크로 감싸서 출력하는 `outline_print` 라는 메소드를 가지고 있는 `OutlinePrint` 트레이트을 만들기를 원한다고 해봅시다. 즉, `(x, y)` 라는 결과를 내도록 `Display` 를 구현한 `Point` 구조체가 주어졌을 때, `x`에 `1`과 `y`에 `3`을 가지고 있는 `Point` 인스턴스 상에서 `outline_print` 를 호출하면, 다음과 같이 출력되어야 합니다:

```
*****  
*      *  
* (1, 3) *  
*      *  
*****
```

`outline_print`의 구현체 내에서, 우리는 `Display` 트레이트의 기능을 사용하길 원합니다. 그러므로, 우리는 `OutlinePrint` 트레이트이 `Display` 또한 구현하여 `OutlinePrint` 가 필요로 하는 기능을 제공하는 타입에서만 동작할 것임을 명시할 필요가 있습니다. 이는 트레이트 정의 부분에서 `OutlinePrint: Display` 라고 명시하는 것으로 할 수 있습니다. 이 기법은 트레이트에게 트레이트 바운드 추가하는 것과 유사합니다. Listing 19-30은 `OutlinePrint` 트레이트의 구현체를 보여줍니다:

Filename: src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

Listing 19-30: `Display`의 기능을 요구하는 `OutlinePrint` 트레이트 구현하기

`OutlinePrint`가 `Display` 트레이트를 요구한다고 명시했으므로, 우리는 `Display`를 구현한 어떤 타입이든 자동으로 구현되어 있는 `to_string` 함수를 사용할 수 있습니다. 만일 트레이트 이름 뒤에 `:` `Display`를 추가하지 않고 `to_string`의 이용을 시도하면, 현재 스코프 내에 `&Self` 타입을 위한 `to_string` 메소드가 없다는 에러를 얻게 됩니다.

아래 `Point` 구조체처럼 `Display`를 구현하지 않은 타입에 대해 `OutlinePrint`를 구현 시도하면 어떤 일이 벌어지는지 봅시다:

Filename: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

우리는 `Display`가 요구되었으나 구현되지 않았다고 말하는 에러를 얻습니다:

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
--> src/main.rs:20:6
   |
20 | impl OutlinePrint for Point {}
   | ^^^^^^^^^^^^^ `Point` cannot be formatted with the default
formatter;
try using `{:?}` instead if you are using a format string
   |
= help: the trait `std::fmt::Display` is not implemented for `Point`
```

이를 고치기 위해서는 아래와 같이 `Point` 상에 `Display`를 구현하여 `OutlinePrint` 가 요구하는 제약 사항을 만족시켜줍니다:

Filename: src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

그러면 `Point` 상의 `OutlinePrint` 트레이트 구현은 성공적으로 컴파일될 것이고, 우리는 애스터리스크로 감싸진 값을 출력하기 위해 `Point` 인스턴스 상에서 `outline_print`를 호출할 수 있습니다.

외부 타입에 대해 외부 트레이트를 구현하기 위한 뉴타입 패턴 (newtype pattern)

10장의 “타입 상에 트레이트 구현하기”절에서, 우리는 트레이트를 구현하려면 타입 혹은 트레이트 둘 중 최소 하나는 우리의 크레이트 내의 것이어야 한다고 기술하는 고아 규칙에 대해 언급했습니다. 이러한 제약은 **뉴타입 패턴** (*newtype pattern*)을 사용하여 우회할 수 있는데, 이는 튜플 구조체 내에 새로운 타입을 만드는 것입니다. (튜플 구조체에 대해서는 5장의 “새로운 타입을 만들기 위한 이름있는 항목 없는 튜플 구조체”절에서 다루었습니다.) 튜플 구조체는 하나의 필드를 가지게 될 것이고 우리가 트레이트를 구현하길 원하는 타입을 얇게 감싸는 래퍼가 될 것입니다. 그러면 이 래퍼 타입은 우리 크레이트 내에 있게 되고, 이 래퍼에 대하여 트레이트를 구현할 수 있습니다. **뉴타입**이란 하스켈 프로그래밍 언어로부터 기원한 용어입니다. 이 패턴을 사용하는데 있어 런타임 성능 패널티는 없으며, 래퍼 타입은 컴파일할 때 생략됩니다.

한가지 예로서, 우리가 `Vec`에 대하여 `Display`를 구현하고 싶다고 가정해보면, 이는 `Display` 트레이트과 `Vec` 타입이 우리 크레이트 밖에서 정의되어 있기 때문에 고아 규칙이 이를 할 수 없게끔 방지합니다. 우리는 `Vec`의 인스턴스를 가지고 있는 `Wrapper` 구조체를 만들 수 있습니다; 그런 다음 Listing 19-31에서 보시는 것처럼 `Wrapper` 상에 `Display`를 구현하고 `Vec` 값을 이용할 수 있습니다.

Filename: src/main.rs

```

use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}

```

Listing 19-31: `Display`를 구현하기 위해서 `Vec<String>`을 감싼 `Wrapper` 타입 만들기

`Display`의 구현체는 내부의 `Vec`에 접근하기 위해 `self.0`를 사용하는데, 이는 `Wrapper` 가 튜플 구조체이고 `Vec`이 이 튜플의 0번째 아이템이기 때문입니다. 그러면 우리는 `Wrapper` 상에서 `Display` 타입의 기능을 사용할 수 있습니다.

이 기법의 부정적인 면은 `Wrapper` 가 새로운 타입이므로, 들고 있는 원래 값의 메소드를 가지지 못한다는 점입니다. `Wrapper` 가 정확히 `Vec`처럼 다뤄질 수 있게 하려면, `Wrapper` 상에 `Vec`의 모든 메소드들을 직접 구현하여 이를 `self.0`에게 위임할수 있게 해야할 것입니다. 만일 새로운 타입이 내부 타입이 가지고 있는 모든 메소드를 갖길 원한다면, `Wrapper` 상에 `Deref` 트레잇을 구현하는 것이 해결책이 될 수 있습니다. (`Deref` 트레잇은 15장의 “`Deref` 트레잇을 사용하여 스마트 포인터를 보통의 참조자처럼 다루기”절에서 논했었습니다.) 만일 `Wrapper` 타입이 내부 타입의 모든 메소드를 가질 필요는 없다면, 예를 들어 `Wrapper` 타입의 동작을 제약하기 위해서는, 우리가 원하는 메소드만 수동으로 구현해야 할 것입니다.

이제 여러분은 트레잇과 관련하여 뉴타입 패턴이 어떻게 사용되는지 알게 되었습니다; 이는 심지어 트레잇이 포함되어 있지 않을 때라도 유용한 패턴입니다. 초점을 바꿔서 러스트의 타입 시스템과 상호작용하는 몇가지 고급 기법을 살펴봅시다.

고급 타입

러스트의 타입 시스템은 이 책에서 언급은 했지만 아직 논의하지는 않았던 몇가지 기능들을 가지고 있습니다. 우리는 대개 왜 뉴타입이 타입으로서 유용한지를 시험함으로서 뉴타입에 대해 논하는 것으로 시작할 것입니다. 그 다음 뉴타입과 비슷한 기능이지만 약간 다른 의미를 가지고 있는 타입 별칭(type alias)으로 넘어가겠습니다. 또한 **!** 타입과 동적인 크기의 (dynamically sized) 타입에 대해 논할 것입니다.

노트: 다음 절은 여러분이 이전 절 “외부 타입에 대해 외부 트레이트를 구현하기 위한 뉴타입 패턴”을 읽었음을 가정합니다.

타입 안전성과 추상화를 위한 뉴타입 패턴 사용하기

뉴타입 패턴은 우리가 지금까지 논했던 것 이상으로 다른 작업에 대해서도 유용한데, 여기에는 어떤 값이 혼동되지 않도록 정적으로 강제하는 것과 어떤 값의 단위 표시로서의 기능을 포함합니다. 여러분은 Listing 19-23에서 단위를 나타내기 위해 뉴타입을 사용하는 예제를 봤습니다: `u32` 값을 뉴타입으로 감싼 `Millimeters`와 `Meters` 구조체를 상기하세요. 만일 우리가 `Millimeters` 타입의 파라미터를 가지고 함수를 작성했다면, 의도치않게 그 함수에 `Meters` 타입의 값이나 그냥 `u32` 값을 넣어서 호출 시도를 하는 프로그램의 컴파일을 하지 못하게 됩니다.

뉴타입 패턴의 또다른 사용례는 어떤 타입의 몇몇 자세한 구현 사항을 추상화 하는 것입니다: 예를 들어 우리가 가능한 기능을 제약하기 위해 뉴타입을 직접 사용했다면 뉴타입은 내부의 비공개 타입이 가진 API와 다른 공개 API를 노출할 수 있습니다.

뉴타입은 또한 내부 구현사항을 숨길 수 있습니다. 예를 들어, 우리는 사람의 ID와 그의 이름을 저장하는 `HashMap<i32, String>`을 감싸는 `People` 타입을 제공할 수 있습니다. `People`을 사용하는 코드는 오직 우리가 제공하는 공개 API만을 통해 상호작용할 것이며, 여기에는 `People` 컬렉션에 이름 문자열을 추가하는 메소드 같은게 있겠지요; 이 코드에서는 우리가 내부적으로 이름에 대해 `i32` ID를 할당한다는 점을 알 필요가 없을 것입니다. 뉴타입 패턴은 캡슐화를 하여 자세한 구현 사항을 숨기기 위한 가벼운 방식으로, 캡슐화에 대한 것은 17장의 “자세한 구현사항을 숨기는 캡슐화” 절에서 다루었습니다.

타입 별칭은 타입의 동의어를 만듭니다

뉴타입 패턴에 덧붙여서, 러스트는 존재하는 타입에게 다른 이름을 부여하기 위한 *타입 별칭*(type alias) 선언 기능을 제공합니다. 이를 위해서는 `type` 키워드를 사용합니다. 예를 들어, 우리는 아래와 같이 `i32`에 대한 별칭 `Kilometers`를 생성할 수 있습니다:

```
type Kilometers = i32;
```

이제 별칭인 `Kilometers`는 `i32`와 동의어입니다; 우리가 Listing 19-23에서 만들었던 `Millimeters` 및 `Meters`와는 달리, `Kilometers`는 분리된, 새로운 타입이 아닙니다. `Kilometers` 타입의 값은 `i32` 타입의 값과 동일한 것으로 취급될 것입니다:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

`Kilometers`와 `i32`가 동일한 타입이기 때문에, 우리는 두 타입의 값을 더할 수 있고 `i32` 파라미터를 갖는 함수에게 `Kilometers` 값을 넘길 수 있습니다. 그러나, 이 방법을 사용하면 우리는 앞서 논의했던 뉴타입 패턴이 제공하는 타입 검사의 이점을 얻지 못합니다.

타입 동의어의 주요 사용 사례는 반복 줄이기입니다. 예를 들어, 우리는 아래와 같이 길다란 타입을 가질지도 모릅니다:

```
Box<Fn() + Send + 'static>
```

이러한 길다란 타입을 함수 시그니처 혹은 타입 명시로 코드의 모든 곳에 작성하는 것은 성가시고 에러를 내기도 쉽습니다. Listing 19-32와 같은 코드로 가득한 프로젝트가 있다고 상상해보세요.

```
let f: Box<Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<Fn() + Send + 'static> {
    // --snip--
}
```

Listing 19-32: 수많은 곳에 긴 타입을 사용하기

타입 별칭은 반복을 줄임으로서 이 코드의 관리를 더 잘하게끔 만들어줍니다. Listing 19-33에서 우리는 이 장황한 타입에 대해 `Thunk`라는 이름의 별칭을 도입해서 이 타입이 사용되는 모든 부분을 짧은 별칭인 `Thunk`로 대체할 수 있습니다.

```
type Thunk = Box<Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

Listing 19-33: 반복을 줄이기 위해 타입 별칭 `Thunk`을 도입하기

이 코드가 훨씬 읽고 쓰기 쉽습니다! 타입 별칭을 위한 의미없는 이름을 고르는 것은 또한 여러분의 의도를 전달하는 데에 도움을 줄 수 있습니다 (`thunk`는 이후에 실행될 코드를 위한 단어로, 저장되는 클로저를 위한 적절한 이름입니다.)

타입 별칭은 또한 `Result<T, E>` 타입의 반복을 줄이기 위해 흔하게 사용됩니다. 표준 라이브러리의 `std::io` 모듈을 고려해 보세요. I/O 연산들은 작동에 실패하는 상황을 다루기 위해서 자주 `Result<T, E>`을 반환합니다. 이 라이브러리는 모든 가능한 I/O 에러를 표현하는 `std::io::Error` 구조체를 가지고 있습니다. `std::io` 내의 많은 함수들이 `E`가 `std::io::Error`인 `Result<T, E>`을 반환합니다. `Write` 트레이트의 아래 함수들 같이 말이죠:

```
use std::io::Error;
use std::fmt;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

`Result<..., Error>`이 너무 많이 반복됩니다. 그렇기 때문에, `std::io`는 이 타입의 별칭 선언을 갖고 있습니다:

```
type Result<T> = Result<T, std::io::Error>;
```

이 선언이 `std::io` 모듈 내에 있으므로, 우리는 완전 정규화된 별칭 `std::io::Result<T>`을 사용할 수 있습니다; 이는 `E`가 `std::io::Error`로 채워진 `Result<T, E>`입니다. `Write` 트레이트 함수 시그니처는 결국 아래와 같이 보이게 됩니다:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: Arguments) -> Result<()>;
}
```

이 타입 별칭은 두 가지 방식으로 도움을 줍니다; 코드를 작성하기 더 편하게 해주고 **그러면서도 모든 std::io**에 걸쳐 일관된 인터페이스를 제공합니다. 이것이 별칭이기 때문에, 이것은 그저 또 다른 **Result<T, E>**일 뿐이고, 이는 우리가 **Result<T, E>**을 가지고 쓸 수 있는 어떠한 메소드는 물론, **?** 같은 특별 문법도 사용할 수 있음을 의미합니다.

결코 반환하지 않는 ! 부정 타입

러스트는 **!**로 칭하는 특별한 타입을 가지고 있는데 타입 이론 용어에서는 이 타입이 값을 가지지 않기 때문에 **빈 타입 (empty type)**으로 알려져 있습니다. 우리는 이를 **부정 타입 (never type)**이라고 부르는 편을 선호하는데, 그 이유는 어떤 함수가 결코 값을 반환하지 않을 때 반환 타입의 자리에 대신하기 때문입니다. 아래에 예제가 있습니다:

```
fn bar() -> ! {
    // --snip--
}
```

이 코드는 “함수 **bar**가 결코 반환하지 않는다”라고 읽힙니다. 결코 반환하지 않는 함수는 **발산 함수 (diverging function)**라고 부릅니다. 우리는 **!** 타입의 값을 만들수 없으므로 **bar**는 결코 반환이 가능하지 않습니다.

하지만 여러분이 값을 전혀 만들 수 없는 타입의 사용처는 무엇일까요? Listing 2-5의 코드를 상기해보세요; 여기 Listing 19-34에 재현해두었습니다.

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

Listing 19-34: `continue`로 끝나는 갈래를 가진 `match`

이 시점에서, 이 코드의 몇가지 세부 사항은 생략하겠습니다. 6장의 “`match` 흐름 제어 연산자” 절에서, 우리는 `match`의 갈래들이 동일한 타입을 반환해야 한다고 논했습니다. 따라서, 예를 들어 다음과 같은 코드는 동작하지 않습니다:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
}
```

이 코드의 `guess` 타입은 정수 및 문자열 이어야 할 것이고, 러스트는 `guess` 가 단 하나의 타입을 가져야 함을 요구합니다. 그러면 `continue` 가 반환하는 것은 무엇일까요? 어떻게 Listing 19-34에서 한 쪽의 갈래에서는 `u32` 를 반환하고 다른 갈래에서는 `continue` 로 끝나는 것이 허용되었을까요?

여러분이 짐작하셨던 것처럼, `continue` 는 `!` 값을 갖습니다. 즉, 러스트가 `guess` 의 타입을 계산할 때, 컴파일러는 매치의 두 갈래를 살펴보는데, 전자는 `u32` 의 값이고 후자는 `!` 값입니다. `!` 가 값을 가질 수 없으므로, 러스트는 `guess` 의 타입이 `u32` 이라고 결정합니다.

이 동작을 기술하는 정규적인 방법은 타입 `!` 의 표현식이 어떠한 다른 타입으로도 강제될 수 있다는 것입니다. `continue` 가 값을 반환하지 않으므로 이 `match` 의 갈래를 `continue` 로 끝내는 것이 허용됩니다; 대신 실행 지점이 루프의 상단으로 이동되므로, 우리는 `guess` 에 결코 값을 대입할 수 없습니다.

부정 타입은 또한 `panic!` 매크로에서도 유용하게 쓰입니다. `Option<T>` 값 상에서 값을 생산하거나 패닉을 일으키기 위해 호출한 `unwrap` 함수 기억하시죠? 여기 그 정의가 있습니다:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

이 코드에서 Listing 19-34의 `match` 와 동일한 일이 일어납니다: 러스트는 `val` 이 `T` 타입을 갖고 `panic!` 이 `!` 타입을 가지므로 전체 `match` 표현식의 결과값은 `T` 라고 봅니다. 이 코드는 `panic!` 이 값을 생산하지 않기 때문에 동작합니다; 패닉은 프로그램을 끝내죠. `None` 케이스에서는 `unwrap` 으로부터의 값을 반환하지 않을 것이므로, 이 코드는 유효합니다.

`!` 타입을 갖는 마지막 하나의 표현식은 `loop` 입니다:

```
print!("forever ");

loop {
    print!("and ever ");
}
```

여기서 루프는 결코 끝나지 않으므로, `!` 가 이 표현식의 값입니다. 그러나, `break` 을 포함시키면 이는 참이

아니게 되는데, 이는 루프가 `break`에 도달했을 때 멈추게 될 것이기 때문입니다.

동적인 크기의 타입과 `Sized`

특정 타입의 값을 할당하기 위한 공간의 크기 등 특정한 세부사항을 알기 위한 러스트의 요구로 인하여, 타입 시스템에서 혼란할 수 있는 구석이 있습니다: 바로 **동적인 크기의 타입** (*dynamically sized type*)에 대한 개념입니다. 이따금 *DST* 혹은 **크기 없는 타입** (*unsized type*)이라고도 불리는 이 타입은 우리가 오직 런타임에서만 그 크기를 알 수 있는 값을 이용하는 코드를 작성할 수 있게 해줍니다.

우리가 이 책을 통틀어 사용해온 `str`이라고 불리우는 동적인 크기의 타입의 세부사항을 파헤쳐봅시다. 그렇습니다. `&str`이 아니라 `str` 그 자체가 바로 DST입니다. 우리는 그 문자열이 얼마나 긴지 런타임이 될 때까지 알 수 없는데, 이는 우리가 `str` 타입의 변수를 만들수도, `str` 타입의 인자를 가질수도 없음을 의미합니다. 아래의 동작하지 않는 코드를 고려해보세요:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

러스트는 특정한 타입의 어떤 값을 위해 얼마나 많은 메모리를 할당해야 하는지 알 필요가 있으며, 하나의 타입의 모든 값은 동일한 크기의 메모리를 사용해야 합니다. 만일 러스트가 위의 코드의 작성을 허용한다면, 위의 두 `str` 값은 동일한 크기의 공간을 차지할 필요가 있을 것입니다. 그러나 이 둘은 서로 다른 길이를 가지고 있습니다: `s1`은 12 바이트의 저장소가 필요하고 ``s2`는 15가 필요하군요. 이것이 바로 동적인 크기의 타입을 보유하는 변수를 만들수 없는 이유입니다.

그러면 우리는 뭘 할까요? 위의 경우, 여러분은 이미 해답을 알고 있습니다: 우리는 `s1`과 `s2`의 타입을 `str`이 아닌 `&str`로 만듭니다. 4장의 “스트링 슬라이스” 절에서 슬라이스 데이터 구조는 슬라이스의 시작 위치와 길이를 저장한다고 얘기했던 것을 상기하세요.

따라서 `&T`는 `T`가 위치한 곳의 메모리 주소값을 저장한 단일값임에도 불구하고, `&str`는 두 개의 값입니다: `str`의 주소와 길이 말이죠. 그런 점에서, 우리는 `&str` 값의 크기를 컴파일 시점에 알 수 있습니다: 길이상 `usize`의 크기의 두 배가 되지요. 즉, 참조하고 있는 문자열의 길이가 얼마든 상관없이, 우리는 언제나 `&str`의 크기를 알 수 있습니다. 대개의 경우 이것이 러스트 내에서 동적인 크기의 타입이 사용되는 방식입니다: 이들은 동적인 정보의 크기를 저장하는 추가적인 메타데이터를 가지고 있습니다. 동적인 크기의 타입의 황금률은 우리가 언제나 동적인 크기의 타입의 값을 어떤 종류의 포인터에 저장해야 한다는 것입니다.

우리는 `str`을 모든 종류의 포인터와 결합할 수 있습니다: 예를 들어, `Box<str>` 혹은 `Rc<str>` 같은 것들 말이죠. 사실, 여러분은 다른 동적인 크기의 타입을 통해 이미 이를 보셨습니다: 바로 트레잇입니다. 모든 트레잇은 그 트레잇의 이름을 사용함으로서 참조할 수 있는 동적인 크기의 타입입니다. 17장의 “서로 다른 타입의 값을 허용하기 위한 트레잇 객체 사용하기” 절에서, 트레잇을 트레잇 객체로 사용하기 위해서는 이를 `&Trait` 혹은 `Box<Trait>`와 같은 식으로 포인터에 넣어야 한다고 언급했었습니다 (`Rc<Trait>` 또한

동작할 것입니다).

DST를 가지고 작업하기 위해서, 러스트는 어떤 타입의 크기를 컴파일 타임에 알 수 있는지 혹은 없는지를 결정하기 위해 **Sized**라는 이름의 특별한 트레이트를 가지고 있습니다. 이 트레이트는 크기가 컴파일 타임에 알려진 모든 것들에 대해 자동으로 구현됩니다. 추가적으로, 러스트는 암묵적으로 모든 제네릭 함수들에게 **Sized**를 바운드로 추가합니다. 즉, 아래와 같은 제네릭 함수의 정의는:

```
fn generic<T>(t: T) {  
    // --snip--  
}
```

실제로는 우리가 아래와 같이 작성한 것처럼 취급됩니다:

```
fn generic<T: Sized>(t: T) {  
    // --snip--  
}
```

기본적으로, 제네릭 함수는 컴파일 타임에 크기를 알 수 있는 타입에 대해서만 작동할 것입니다. 그러나, 여러분은 이 제한사항을 느슨하게 하기 위해 다음과 같은 특별 문법을 사용할 수 있습니다:

```
fn generic<T: ?Sized>(t: &T) {  
    // --snip--  
}
```

?Sized 트레이트 바운드는 Sized 트레이트 바운드의 반대 개념입니다: 우리는 이를 “T가 Sized 일 수도 있고 아닐 수도 있다”라고 읽을 수 있습니다. 이 문법은 다른 트레이트들 말고 오직 Sized에 대해서만 사용 가능합니다.

또한 t 파라미터가 T에서 &T로 바뀐 점을 주목하세요. 이 타입이 Sized가 아닐지도 모르기 때문에, 우리는 이를 어떤 종류의 포인터 뒤에 놓고 사용할 필요가 있습니다. 위의 경우에는 참조자를 선택했습니다.

다음으로는 함수와 클로저에 대해 다루겠습니다!

고급 함수와 클로저

마지막으로, 우리는 함수와 클로저와 관련된 몇 가지 고급 기능들을 탐구할 것이며, 여기에 함수 포인터 및 클로저 반환이 포함됩니다.

함수 포인터

우리는 어떻게 클로저를 함수 인자로 넘기는지에 대해 이야기 했었습니다; 여러분은 또한 일반 함수를 함수 인자로 넘길 수 있습니다! 이 기법은 새로운 클로저를 정의하는 것보다는 우리가 이미 정의해둔 함수를 넘기고 싶을 때 유용합니다. 우리가 함수를 다른 함수의 인자로서 사용하게끔 하기 위해서는 함수 포인터를 이용합니다. 함수는 (소문자 f를 써서) 타입 `fn`이 되는데, `Fn` 클로저 트레이트와 혼동하면 안됩니다. `fn` 타입을 함수 포인터라 부릅니다. 어떤 파라미터가 함수 포인터임을 명시하기 위한 문법은 클로저의 그것과 비슷한데, Listing 19-35에서 보시는 것과 같습니다.

Filename: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

Listing 19-35: `fn` 타입을 사용하여 함수 포인터를 인자로서 허용하기

이 코드는 `The answer is: 12`를 출력합니다. 여기서는 `do_twice` 내의 파라미터 `f`가 타입 `i32`을 파라미터로 받아서 `i32`를 반환하는 `fn`이라고 명시하였습니다. 그러면 우리는 `do_twice`의 본체 안에서 `f`를 호출할 수 있습니다. `main` 내에서는 `do_twice`의 첫 번째 인자로서 함수 이름인 `add_one`을 넘길 수 있습니다.

클로저와 달리 `fn`은 트레이트가 아니고 타입이므로, 우리는 `fn`을 트레이트 바운드로 `Fn` 트레이트 중 하나를 사용한 제네릭 타입 파라미터를 정의하기보다는 직접 파라미터 타입으로 특정합니다.

함수 포인터는 클로저 트레이트 세 종류 (`Fn`, `FnMut`, 그리고 `FnOnce`) 모두를 구현하므로, 우리는 언제나 클로저를 인자로서 기대하는 함수에게 함수 포인터를 넘길 수 있습니다. 제네릭 타입과 클로저 트레이트 중 하

나를 사용하는 함수를 작성하여 여러분의 함수가 함수 혹은 클로저를 허용할 수 있게 하는 것이 가장 좋습니다.

여러분이 오직 `fn`만 허용하고 클로저는 허용하지 않고 싶을 수 있는 예는 클로저를 가지고 있지 않은 외부 코드와의 인터페이싱을 할 때입니다: C 함수는 함수를 인자로서 허용하지만, 클로저를 가지고 있지 않지요.

우리가 인라인으로 정의된 클로저 혹은 이름을 가진 함수 중 하나를 사용할 수 있는 경우의 예제로서, `map`의 사용을 살펴봅시다. `map` 함수를 사용하여 숫자 벡터를 스트링 벡터로 전환하기 위해서는 아래와 같이 클로저를 이용할 수 있습니다:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(|i| i.to_string())
    .collect();
```

혹은 아래와 같이 클로저 대신 `map`의 인자로서 함수 이름을 쓸 수도 있습니다:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(ToString::to_string)
    .collect();
```

`to_string`이라는 이름의 사용 가능한 함수가 여러개이므로, 앞서 “고급 트레이트” 절에서 이야기했던 완전 정규화 문법을 사용해야 하는 점을 주목하세요. 여기서 우리는 `ToString` 트레이트 내에 정의된 `to_string` 함수를 사용하는데, 이는 표준 라이브러리가 `Display`를 구현한 어떤 타입에 대해서든 구현체를 가지고 있습니다.

어떤 이들은 이런 스타일을 선호하고, 어떤 이들은 클로저의 사용을 선호합니다. 이들은 컴파일되어 결국 같은 코드가 되므로, 어떤 스타일이든 여러분에게 더 깔끔해보이는 스타일로 이용하세요.

클로저 반환하기

클로저는 트레이트에 의해 표현되는데, 이는 우리가 클로저를 직접 반환할 수 없음을 의미합니다. 우리가 트레이트를 반환하고 싶어하는 대부분의 경우에는 함수의 반환값으로서 그 트레이트를 구현한 구체 타입을 대신 이용할 수 있습니다. 그러나 클로저에 대해서는 그렇게 할 수 없는데, 이는 클로저가 반환 가능한 구체타입을 가지고 있지 않기 때문입니다; 예를 들면 함수 포인터 `fn`을 반환 타입으로 사용하는 것은 허용되지 않습니다.

아래의 코드는 클로저를 직접 반환 시도를 하지만, 컴파일되지 않을 것입니다:

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

컴파일 에러는 다음과 같습니다:

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static: std::marker::Sized` is not satisfied
-->
|
1 | fn returns_closure() -> Fn(i32) -> i32 {
|           ^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32 + 'static` does not have a constant size known at compile-time
|
= help: the trait `std::marker::Sized` is not implemented for `std::ops::Fn(i32) -> i32 + 'static`
= note: the return type of a function must have a statically known size
```

에러가 또 **Sized** 트레이트를 참조하는군요! 러스트는 클로저를 저장하기 위해 얼만큼의 공간이 필요한지 알지 못합니다. 이 문제에 대한 해결책은 이전에 봤습니다. 우리는 트레이트 객체를 사용할 수 있습니다:

```
fn returns_closure() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

이 코드는 그냥 잘 컴파일 될 것입니다. 트레이트 객체에 대한 더 자세한 내용은 17장의 “서로 다른 타입의 값을 허용하기 위한 트레이트 객체 사용하기” 절을 참고하세요.

정리

휘유! 이제 여러분은 자주 사용하지는 않겠지만 매우 특정한 환경에서는 필요한 것임을 알게될 러스트의 몇 가지 기능들을 여러분의 도구함에 챙겼습니다. 우리가 몇몇 복잡한 주제를 소개했으므로 여러분이 이들을 여러 메세지 제안이나 다른 사람들의 코드에서 마주쳤을 때는 이 개념들과 문법을 인식할 수 있을 것입니다. 이 장을 여러분에게 해결책을 안내할 참고자료로서 사용하세요.

다음으로, 우리는 책 전체에 걸쳐 논의한 모든 것을 실전 예제에 넣어서 한가지 프로젝트를 더 해볼 것입니다!

마지막 프로젝트: 멀티 스레드 웹 서버 만들기

오랜 여정이었습니다만 이제 우린 이 책의 마지막에 도달했습니다. 이번 챕터에서는 여태까지의 내용을 요약하고 마지막 챕터의 내용을 정리하기 위해 프로젝트를 하나 더 만들것입니다.

"hello"를 나타내는 웹 서버를 우리의 마지막 프로젝트로 만들어 봅시다. 완성하면 웹 브라우저에서는 그림 20-1과 같은 모습으로 보일 것입니다.



Hello!

Hi from Rust

그림 20-1: 마지막 프로젝트

웹서버를 만들 계획은 아래와 같습니다.

1. TCP와 HTTP에 대해 간단히 배우기
2. TCP 소켓 연결요청을 수신하기
3. HTTP 요청의 일부를 분석하기
4. 적절한 HTTP 응답 만들기
5. 스레드 풀을 이용해 서버의 응답속도를 개선하기

시작하기전에 한가지 알려드릴게 있습니다. 우리가 사용할 방법이 러스트를 이용해 웹 서버를 만드는 최고의 방법은 아닙니다. 다수의 크레이트가 <https://crates.io/>에 등록되어 있으며 이들은 우리가 만들것보다 뛰어나게 웹 서버와 스레드 풀을 구현했습니다.

어쨌든, 이번 챕터에서 우리가 원하는건 배우는 것이지, 쉬운길로 돌아가는것이 아닙니다. 이는 러스트가 시스템 프로그래밍 언어이며, 우리는 다른 언어로는 불가능 하거나, 하기 힘든 저레벨 작업을 할 수 있기 때문이기도 합니다. 우린 기본적인 HTTP 서버와 스레드 풀을 직접 구현할 것이며, 이를 통해 여러분이 나중에 사용

하게 될 크레이트들의 기반이 되는 일반적인 기술들에 대해 배울 수 있습니다.

싱글스레드 기반 웹 서버 만들기

싱글 스레드 기반의 웹 서버가 작동하는것을 알아보는것 부터 시작하겠습니다. 시작하기 전에, 웹서버를 구성하는 프로토콜들에 대해 빠르게 훑어봅시다. 프로토콜들에 대한 자세한 설명은 이 책의 범주를 넘어가지만, 간단한 설명은 여러분에게 도움이 될 것입니다.

웹서버의 두 주요 프로토콜은 *HTTP(Hypertext Transfer Protocol)* 과 *TCP(Transmission Control Protocol)* 입니다. 이 두 프로토콜은 요청-응답(*request-response*) 프로토콜입니다. 요청-응답은 클라이언트가 요청을 생성하면, 서버는 요청을 받고 클라이언트에게 응답하는 과정을 뜻합니다. 요청과 응답의 내용은 각 프로토콜에 의해 정의됩니다.

TCP는 저레벨 프로토콜로, 한 서버에서 다른 서버로 정보를 요청할때 사용하지만, 해당 정보가 무엇인지는 특정하지 않습니다. HTTP는 TCP 상위에서 만들어졌으며, 요청과 응답의 내용을 정의하고 있습니다. HTTP 가 TCP 이외의 프로토콜을 사용하는것은 기술적으로 불가능하지 않지만, 일반적으로 HTTP통신은 TCP프로토콜 위에서 이루어집니다. 이번 장에선 TCP를 이용한 바이트통신과 HTTP를 이용한 요청과 응답을 실습해 볼 것입니다.

TCP 연결에 대한 처리

우리가 만들 웹 서버는 TCP 연결 요청에 대한 처리를 해야하기 때문에, TCP 연결 요청을 수신하는 것 부터 작업하도록 하겠습니다. 이 작업은 표준 라이브러리에서 제공하는 `std::net` 모듈을 이용해 진행할 수 있습니다. 하던대로 새 프로젝트를 만들어 봅시다.

```
$ cargo new hello --bin
     Created binary (application) `hello` project
$ cd hello
```

이제 20-1번 예제의 `src/main.rs` 코드를 입력합시다. 이 코드는 `127.0.0.1:7878` 주소로 TCP 연결 요청에 대해 수신 대기할 것입니다. 만약 요청이 들어온다면, `Connection established!` 가 출력될 것입니다.

파일명: `src/main.rs`

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!");
    }
}
```

예제 20-1: 수신 스트림 대기와 수신시 메시지 출력

우린 `TcpListener` 를 사용하여 `127.0.0.1:7878` 주소로 TCP연결을 수신할 수 있습니다. 이 주소의 `:` 앞부분은 여러분의 컴퓨터의 IP주소를 뜻합니다. (`127.0.0.1` 은 loopback IP로, 현재 컴퓨터를 가리키는 IP 주소입니다) 그리고 `7878` 은 포트를 뜻합니다. 여기서 이 포트를 사용한 이유는 두가지입니다. HTTP는 일반적으로 이 포트에서 요청되며, 7878은 "rust"를 전화기에서 입력했을때의 숫자이기 때문입니다.

Note: 80포트를 이용한 연결은 관리자 권한이 필요하다는 점을 유의하세요; 비 관리자는 1024 이상의 포트 번호만 사용 가능합니다.

위 코드에서 `bind` 함수는 `new` 함수처럼 동작하며 `TcpListener` 의 새 인스턴스를 반환합니다. 이 함수가 `bind` 라는 이름을 가진 이유는 네트워크 관련에서 포트를 수신대기하는 과정을 "포트를 binding한다"라고 부르기 때문입니다.

`bind` 함수는 바인딩의 성공여부를 나타내는 `Result<T, E>` 를 반환합니다. 예를들어, 우리가 만약 80포트를 관리자가 아닌 상태에서 연결하려고 시도할 경우나 같은 포트를 사용하는 프로그램을 여러개 실행할 경우에 바인딩은 실패하게 됩니다. 우리는 학습을 목적으로 서버를 작성하고 있기 때문에, 이러한 에러에 대한 처리를 해줄 필요가 없습니다. 따라서, 우리는 `unwrap` 을 이용해 에러가 생길 경우 프로그램을 멈출것입니다.

`TcpListener` 의 `incoming` 메소드는 스트림의 차례에 대한 반복자를 반환합니다. (보다 정확히는, 여러 스트림의 종류 중 `TcpStream` 에 해당합니다) 각각의 `stream` 은 클라이언트와 서버간의 열려있는 커넥션을 의미합니다. `connection` 은 클라이언트가 서버와 연결하고, 서버가 응답을 생성하고, 서버가 연결을 끊는 요청과 응답 과정을 통틀어 의미합니다. 이와같이, `TcpStream` 은 클라이언트가 보낸 정보를 읽어들이고, 우리의 응답을 스트림에 작성할 수 있게 해줍니다. 전체적으로, 이 `for` 반복문은 각각의 연결을 처리하고 우리에게 일련의 스트림들을 다룰 수 있도록 해줍니다.

현재, 우리는 어떠한 오류가 있을경우 `unwrap` 을 호출하여 프로그램을 종료시키는 방식으로 스트림을 처리합니다. 만약 오류가 없을경우, 프로그램은 메시지를 출력합니다. 우린 다음 항목에서 오류가 없을 경우에 대

한 기능을 더 추가할 것입니다. 우리가 `incoming` 메소드를 통해서 에러를 받을때의 이유는, 클라이언트가 서버로 연결할때 우리가 실제적인 연결을 반복하는것이 아닌, 연결 시도를 반복하기 때문입니다. 연결은 몇가지 이유로 실패할 수 있는데, 대다수의 경우 운영체제의 특성 때문입니다. 예를들어, 대부분의 운영체제는 동시에 열어놓을 수 있는 연결 개수에 제한을 가지고 있는데, 제한 이상으로 연결을 시도할 경우 이미 열려있는 연결이 닫힐때까지 오류를 발생시킵니다.

한번 이 코드를 실행해 봅시다. `cargo run` 을 터미널에 입력하고, 브라우저에서 `127.0.0.1:7878` 로 접속해봅시다. 브라우저는 "연결 재시도"와 같은 에러를 보여줄 것입니다. 이 이유는 현재의 서버는 어떠한 데이터도 전송하지 않기 때문입니다. 하지만 터미널을 보면, 브라우저가 서버에 접속할때 출력된 메시지들을 볼 수 있습니다!

```
Running `target/debug/hello`  
Connection established!  
Connection established!  
Connection established!
```

종종, 브라우저로 한번 요청했을때 여러 메시지가 출력되는걸 보실겁니다. 이유는 브라우저가 페이지뿐만 아니라 다른 여러 리소스를 요청하기 때문입니다. 요청되는 다른 리소스들중 대표적인 것은 브라우저 탭에 표시되는 아이콘인 `favicon.ico` 가 있습니다.

또한 서버가 어떠한 데이터도 보내주지 않기 때문에 브라우저가 여러번 연결을 시도했기 때문일 수도 있습니다. `stream` 이 영역을 벗어날 경우와 반복이 끝날때, `drop` 이 실행되는것처럼 연결이 끊어집니다. 브라우저는 서버와의 연결 문제가 일시적일 수도 있다고 생각하여 끊어진 연결을 재시도하기도 합니다. 여기서 중요한건 우리가 성공적으로 TCP연결을 처리했다는 것입니다.

이전 버전의 코드가 실행되는 프로그램을 종료할때는 `ctrl-c`를 누르고, 여러분이 만든 새 버전의 코드를 실행하기 위해 `cargo run` 명령어를 입력하는것을 기억하세요

요청 데이터 읽기

브라우저로부터의 요청을 읽는 기능을 구현해봅시다! 부담갖지 말고 '연결하기', '연결을 이용해보기'로 나눠서 진행해봅시다. 연결을 처리하기 위해 새 함수를 만들어 봅시다. 여기선 `handle_connection` 이라는 함수를 새로 만들었습니다. TCP 스트림에서 데이터를 읽고 출력해보며 브라우저가 보낸 데이터를 직접 확인해봅시다. 코드를 예제 20-2와 같이 변경합니다.

파일명: `src/main.rs`

```

use std::io::prelude::*;
use std::net::TcpStream;
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    println!("Request: {}", String::from_utf8_lossy(&buffer[..]));
}

```

예제 20-2: `TcpStream`으로부터 데이터를 읽고 출력

우린 `std::io::prelude`를 가져와 스트림으로부터 읽고 쓰는것을 허용하는 특성에 접근할 수 있도록 합니다. `main` 함수 내부의 `for` 반복문 안에서는, 연결에 성공했다는 메시지를 출력하는 대신, 새로 만든 `handle_connection` 함수를 `stream`을 전달하여 호출합니다.

`handle_connection` 함수에선, `stream` 매개변수를 가변으로 만들어 줬습니다. 이유는 `TcpStream` 인스턴스가 내부에서 어떤 데이터가 우리에게 반환되는지 추적하기 때문입니다. 우리가 요청하는것에 따라 더 많은 데이터를 읽거나, 다음 요청때까지 데이터를 저장할 수 있습니다. 이와 같이 내부의 상태가 변경될 수 있기에 `mut` 이 되어야 합니다. 보통 "읽기"는 변화와 관련이 없다고 생각하지만 이 경우 `mut` 키워드가 필요합니다.

다음으로, 실제로 스트림으로부터 데이터를 읽어봅시다. 이는 두가지 과정으로 나뉘는데: 먼저, 우리는 `buffer` (버퍼)를 읽을 데이터를 저장할 스택에 선언해야 합니다. 여기선 버퍼를 기본적인 요청을 저장하는 것과 우리의 목적에 충분한 크기인 512바이트로 만들었습니다. 만약 임의의 크기를 가진 요청을 다룰땐 버퍼 관리는 좀 더 복잡해져야 할 테지만, 지금은 단순하게 생각합시다. 우린 버퍼를 `stream.read`로 전달했는데 이 함수는 `TcpStream`으로부터 읽어들인 바이트를 버퍼로 집어넣는 역할을 합니다.

두번째로, 버퍼 안에있는 바이트들을 문자열로 변환하고 출력합니다. `String::from_utf8_lossy` 함수는 `&[u8]`을 전달받고 `String`으로 바꿔서 제공해줍니다. 함수의 이름 중 "lossy"는 이 함수가 유효하지 않은 UTF-8 배열을 만났을때의 행동을 나타냅니다. 유효하지 않은 배열은 `U+FFFD REPLACEMENT CHARACTER`라는 ⓘ로 교체되는데, 여러분은 아마 이 문자를 버퍼중 요청 데이터로 채워지지 않은곳에서

볼겁니다.

한번 코드를 실행해 보죠. 프로그램을 시작하고 웹 브라우저로 요청을 다시 보내봅시다. 브라우저는 여전히 에러페이지를 띄우겠지만, 우리의 프로그램은 아래와 비슷한 내용을 터미널에 출력할겁니다.

여러분의 브라우저에 따라서 조금씩 다른 출력결과가 나올 수 있습니다. 이렇게 요청 데이터를 출력해보았습니다. 이제 여러분은 **Request: GET** 뒤의 경로를 보고 어째서 한 브라우저가 여러번 연결 요청을 보냈는지 알 수 있습니다. 만약 반복되는 요청들이 모두 **/** 를 요청하고 있다면, 알다시피 브라우저가 우리의 프로그램에게서 응답을 받지 못했기 때문에 **/** 를 가져오려고 하는 것입니다.

한번 이 요청 데이터를 분석하며 브라우저가 우리 프로그램에 뭘 물어보는지 이해해 봅시다.

HTTP 요청을 자세히 살펴보기

HTTP는 텍스트 기반 프로토콜이고, HTTP 요청은 아래와 같은 양식을 따릅니다.

```
Method Request-URI HTTP-Version CRLF  
headers CRLF  
message-body
```

첫번째 줄은 *request_line*이고 클라이언트가 무슨 요청을 하는지에 대한 정보를 담고 있습니다. 요청 라인의 첫번째 부분은 사용된 메소드를 나타냅니다. **GET**이나 **POST** 등을 말하는데, 이는 클라이언트가 어떻게 이 요청을 만들었는지 나타냅니다. 우리 클라이언트는 **GET** 요청을 사용했습니다.

요청 라인의 다음 부분은 **/** 입니다. 클라이언트가 요청한 *URI(Uniform Resource Identifier)* 를 나타내는데, 이는 *URL(Uniform Resource Locator)* 과 완전히는 아니지만 거의 똑같습니다. *URI*와 *URL*의 차이는 이번 장의 우리 의도와는 관계이 별로 없으므로 여기선 머릿속으로 *URL*로 *URI*를 대체하도록 합시다.

마지막 부분은 클라이언트가 사용하는 HTTP 버전입니다. 그 다음은 CRLF 시퀀스로 요청라인이 끝나게 됩니다. CRLF 시퀀스는 `\r\n` 으로도 쓰일 수 있습니다: 여기서 `\r` 부분은 *carriage return*이고 `\n`은 *line feed*입니다. (이 표현은 타자기 시절부터 이어져 온 것입니다) CRLF 시퀀스는 요청 라인을 나머지 요청 데이터로부터 분리시키는 역할을 합니다. 여기서 CRLF 시퀀스는 출력되었을 때 `\r\n` 이 아닌 줄바꿈이 되는 걸 기억하세요.

이제 우리 프로그램이 실행되는 동안 받은 요청 라인 데이터를 한번에 살펴봅시다. `GET` 이 요청 메소드, `/` 가 요청 URI, `HTTP/1.1` 은 버전을 뜻합니다.

요청 라인 이후의 남은 라인들 중 `Host:` 이후는 모두 헤더입니다. (일반적으로 `GET` 메소드를 통한 요청은 body를 가지지 않습니다)

한번 다른 브라우저나 다른 주소(`127.0.0.1:7878/test` 등)으로 요청을 보내보고, 요청 데이터가 어떻게 변화하는지도 살펴보세요.

이제 브라우저가 요청하는 내용이 무슨 뜻인지 알았으니, 역으로 데이터를 보내봅시다!

응답 작성하기

이제 클라이언트의 요청에 대응하는 응답을 보내 봅시다. HTTP 응답은 아래와 같은 양식을 가집니다.

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

첫번째 줄은 *status line*입니다. 이곳엔 응답에 사용된 HTTP 버전, 요청에 대한 결과를 나타내는 상태 코드, 상태 코드에 대한 설명 구문(텍스트)이 들어가 있습니다. CRLF 시퀀스 이후는 헤더, 또 다른 CRLF의 뒤는 응답의 body가 들어갑니다.

여기 HTTP 1.1 버전을 이용한 응답 예시가 있습니다. 상태 코드는 200, 설명 문구는 OK, 헤더와 body는 없습니다.

```
HTTP/1.1 200 OK\r\n\r\n
```

200 상태 코드는 응답 성공을 뜻하는 표준 응답 코드입니다. 이제 이것을 요청에 대한 응답으로 스트림에 작성해 봅시다. 요청 데이터를 출력하는데 사용했던 `handle_connection` 함수의 `println!` 을 지우고 아래 20-3 예제의 코드를 대신 써 넣으세요.

파일명: src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    let response = "HTTP/1.1 200 OK\r\n\r\n";
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

예제 20-3: 스트림에 간단한 HTTP 응답 성공 메시지 작성하기

첫번째 새 줄에선 응답 메시지를 저장할 `response` 변수를 선언했습니다. 그 뒤 `response`의 `as_bytes`를 호출하여 문자열 데이터를 바이트 배열로 변환합니다. `stream`의 `write` 메소드는 & `[u8]`을 전달받고 커넥션에 바이트 배열을 전송합니다.

`write` 작업이 실패할 수 있기 때문에, 우린 전처럼 `unwrap`을 사용합니다. 다시 말하지만, 실제 어플리케이션에선 이런 경우에 에러 처리를 해야합니다. 마지막으로, `flush` 는 모든 바이트들이 커넥션으로 쓰여질 때까지 프로그램을 대기시킵니다. `TcpStream` 은 운영체제의 기능 호출을 최소화하기 위해 내부적으로 버퍼를 사용하기 때문에, 커넥션으로 전송시키기 위해선 `flush` 를 이용해 이 버퍼를 비워야 합니다.

코드를 실행시키고, 요청을 만들어 봅시다. 어떤 데이터도 출력되지 않지만, 웹 브라우저로 `127.0.0.1:7878` 로 접속했을때, 에러 대신 빈 페이지를 볼 수 있을 것입니다. 여러분은 HTTP 요청과 응답을 직접 코딩해 보셨습니다!

실제 HTML로 응답하기

빈 페이지보다 더 많은걸 응답하는 기능을 만들어 봅시다. `src` 폴더가 아닌 여러분의 프로젝트 디렉토리의 루트 디렉토리에 `hello.html` 파일을 새로 만든 뒤, 예제 20-4처럼 HTML을 작성하세요 (여러분이 원하는대로 내용을 바꾸셔도 됩니다)

파일명: `hello.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

예제 20-4: 응답할 HTML 파일 예시

위는 간단한 내용을 가진 HTML5 문서입니다. 이를 서버에서 요청이 들어왔을 때 반환하도록 바꾸기 위해서, `handle_connection` 을 예제 20-5처럼 HTML 파일을 읽고, 응답의 body에 추가하고, 전송하도록 수정합니다.

파일명: src/main.rs

```
use std::fs::File;
// --생략--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let mut file = File::open("hello.html").unwrap();

    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();

    let response = format!(
        "HTTP/1.1 200 OK\r\nContent-Length: {}\r\n\r\n{}",
        contents.len(),
        contents
    );

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

예제 20-5: *hello.html* 의 내용을 응답의 body에 넣고 전송하기

맨 위줄에서 `File` 표준 라이브러리를 가져왔습니다. 파일을 열고 내용을 읽는 코드는 12장 (예제 12-4)의 I/O 프로젝트에서 파일 내용을 읽을 때 작성해봤으니 익숙하실 겁니다.

다음으로, 우린 `format!` 을 이용해 응답 데이터의 body부분에 파일의 내용을 추가했습니다. HTTP 응답이 유효함을 확실히 하기 위해 우리는 응답 body의 크기로 설정된 `Content-Length` 헤더를 추가했는데, 이 경우에는 `hello.html`의 크기입니다.

이 코드를 `cargo run` 을 이용해 실행하고 브라우저로 `127.0.0.1:7878` 로 접속해보세요, 여러분이 작성한 HTML이 화면에 나타날 것입니다!

하지만 우리는 현재 `buffer` 안에 있는 요청 데이터를 무시하고 무조건 HTML 파일의 내용을 전송합니다. 이 말은 여러분이 브라우저로 `127.0.0.1:7878/something-else` 에 접속해도 똑같은 HTML이 나타난다는 뜻입니다. 우리의 서버는 매우 제한적이고 일반적인 웹 서버와는 다릅니다. 우리는 요청에 따라서 응답을 지정하고자 하고, 오직 `/` 에 대한 정상적인 요청에 대해서만 HTML 파일을 전송하려 합니다.

요청을 확인하고 선택적으로 응답하기

현재 우리 웹 서버는 클라이언트가 무엇을 요청했는지에 관계없이 파일 안의 HTML을 반환합니다. 한번 HTML 파일을 반환하기 전에 브라우저가 `/` 를 요청하는지 확인하고, 만약 다른걸 요청한다면 에러를 반환하는 기능을 추가해봅시다. 이를 위해 우리는 `handle_connection` 을 예제 20-6과 같이 수정할 필요가 있습니다. 이 코드는 받은 요청 내용을 우리가 아는 `/` 로의 요청 내용과 비교하여 `if` 와 `else` 블록을 추가해 요청들을 다르게 처리하는 코드입니다.

파일명: src/main.rs

```
// --생략--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    if buffer.starts_with(get) {
        let mut file = File::open("hello.html").unwrap();

        let mut contents = String::new();
        file.read_to_string(&mut contents).unwrap();

        let response = format!(
            "HTTP/1.1 200 OK\r\nContent-Length: {}\r\n\r\n{}",
            contents.len(),
            contents
        );
        stream.write(response.as_bytes()).unwrap();
        stream.flush().unwrap();
    } else {
        // 기타 다른 요청
    }
}
```

예제 20-6: 요청을 비교하고 / 로의 요청을 다른 요청들과 다르게 처리하기

먼저, 우리는 / 로의 요청에 해당하는 데이터를 get 변수에 하드코딩했습니다. 우리가 버퍼에서 읽어들이는 것은 원시 바이트이기 때문에 get 에 바이트 문자열 구문인 b"" 를 추가해 바이트 문자열로 바꿔줍니다. 이후 if 블록에서 buffer 가 get 의 내용으로 시작하는지 체크합니다. 만약 그렇다면 우린 정상적인 / 로의 요청을 받았다는 뜻이니, 우리의 HTML 파일의 내용을 반환합니다.

만약 buffer 가 get 의 내용으로 시작하지 않는다면, 다른 요청을 받았다는 뜻입니다. 이 요청들을 처리할 else 블록의 코드는 잠시 후에 작성할 예정입니다.

이 코드를 실행시키고 127.0.0.1:7878 로 요청을 보내봅시다. 여러분은 hello.html 의 HTML을 받았을 겁니다. 만약 127.0.0.1:7878/somthing-else 등의 다른 요청을 보낸다면, 여러분이 예제 20-1과 예제 20-2를 실행했을 때처럼 연결 에러가 날 것입니다.

이제 예제 20-7의 코드를 else 블록에 추가하고 응답으로 404 상태코드를 보내봅시다. 여기서 404 상태 코드는 요청에 대한 내용을 찾을 수 없다는 뜻을 신호입니다. 또한 우린 최종 유저에게 브라우저에서 보여질 페이지를 위한 HTML도 반환해볼겁니다.

Filename: src/main.rs

```
// --생략--

} else {
    let status_line = "HTTP/1.1 404 NOT FOUND";
    let mut file = File::open("404.html").unwrap();
    let mut contents = String::new();

    file.read_to_string(&mut contents).unwrap();

    let response = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        status_line,
        contents.len(),
        contents
    );

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

예제 20-7: `/` 로의 요청이 아닐 경우 `404` 상태 코드와 에러 페이지를 응답

여기서, 우린 `404` 상태 코드와 `NOT FOUND` 상태 메시지를 가진 `status line`을 응답에 포함하고 있습니다. 헤더는 없고, `body`는 `404.html` 파일의 HTML 내용입니다. 여러분은 `hello.html` 옆에 에러 페이지에 사용 할 `404.html` 을 만들고, 자유롭게 HTML 을 입력하거나 예제 20-8의 예시를 사용하시기 바랍니다.

파일명: `404.html`

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Hello!</title>
    </head>
    <body>
        <h1>Oops!</h1>
        <p>Sorry, I don't know what you're asking for.</p>
    </body>
</html>
```

예제 20-8: `404` 응답에 전달될 페이지의 내용 예시

변경사항들을 포함하고 다시 서버를 실행시켜 보세요. `127.0.0.1:7878` 로의 요청은 `hello.html` 의 내용을 반환할 것이고, 그 외의 요청 (`127.0.0.1:7878/foo` 등)은 `404.html` 의 내용을 반환할 것입니다.

리팩토링

`if` 와 `else` 블록에는 중복되는 부분이 많습니다. 둘다 파일을 읽고, 파일의 내용을 스트림에 작성합니다. 차이점은 오직 status line과 파일명뿐입니다. 코드를 좀더 간결하게하기 위해 이 차이점만 `if` 와 `else` 줄로 분리하고 status line과 파일명을 변수로 맡기도록 합시다. 그럼 우린 이 변수들을 파일을 읽고 응답을 작성하는데 쓰기만 하면 됩니다. 예제 20-9에서 `if` 와 `else` 블록의 코드 대부분을 변경한 결과를 보실 수 있습니다.

파일명: src/main.rs

```
// --생략--

fn handle_connection(mut stream: TcpStream) {
    // --생략--

    let (status_line, filename) = if buffer.starts_with(b"GET") {
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let mut file = File::open(filename).unwrap();
    let mut contents = String::new();

    file.read_to_string(&mut contents).unwrap();

    let response = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        status_line,
        contents.len(),
        contents
    );

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

예제 20-9: `if` 와 `else` 블록을 각각이 처리하는 내용 중 다른부분만 포함하도록 리팩토링

이제 `if`와 `else` 블록은 오직 각각의 적절한 status line과 파일명을 튜플로 반환할 뿐입니다. 우린 이 두 값을 18장에 나온 `let` 의 표현 패턴을 이용해 분리하고 각각을 `status_line` 과 `filename` 변수로 대입합니다.

이전의 중복된 코드는 이제 `if`와 `else` 블록의 밖에 있습니다. 그리고 `status_line` 과 `filename` 변수를 사용함으로써, 두 경우의 차이를 쉽게 볼 수 있고, 만약 우리가 파일을 읽고 응답하는 과정을 개선하고

싶을때 한 곳만 수정해도 된다는 이점을 얻을 수 있습니다. 물론 예제 20-9에 나온 코드는 예제 20-8에 나온 것과 똑같이 동작할것입니다.

훌륭합니다! 우린 이제 대략 40줄 정도의 러스트 코드로 한 요청에 대해선 내용 있는 페이지로 응답하고 그 외의 요청은 **404** 를 응답하는 간단한 웹 서버를 만들어 보았습니다.

현재, 우리가 만든 서버는 싱글 스레드로 동작합니다. 즉 한번에 하나의 요청밖에 대응하지 못합니다. 이게 왜 문제가 되는지 느린 요청들이 들어온 상황을 시뮬레이팅하고, 우리 서버가 한번에 여러 요청을 처리할 수 있도록 고칠것입니다.

서버를 싱글 스레드에서 멀티 스레드로 바꾸기

현재, 서버는 한번에 하나의 요청만 처리할 것입니다. 즉 첫번째 요청에 대한 작업이 끝나기 전에 두번째 요청이 들어온다면 앞선 작업이 끝날때까지 대기하게 됩니다. 만약 서버가 훨씬 더 많은 요청을 받게 된다면, 처리는 점점 더 늦어지게 됩니다. 나중에 들어온 요청은 앞선 요청보다 더 빠르게 처리 될 수 있더라도 긴 시간을 기다려야 할 것입니다. 우린 이 문제를 해결해야 합니다만, 먼저 현재 우리의 문제를 살펴보도록 하죠.

현재 서버에서 느린 요청을 시뮬레이팅하기

우린 현재의 우리가 만든 서버에서 느린 요청이 어떻게 다른 요청들에게 영향을 미칠 수 있는지 살펴 볼 것입니다. Listing 20-10은 `/sleep` 요청을 처리할 때 응답하기 전에 5초간 서버를 멈추도록 하여 느린 요청을 시뮬레이션 합니다.

파일명: src/main.rs

```
use std::thread;
use std::time::Duration;
// --생략--

fn handle_connection(mut stream: TcpStream) {
    // --생략--

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    // --생략--
}
```

Listing 20-10: `/sleep` 요청을 인식할 시 5초간 멈춤으로써 느린 요청을 시뮬레이션 하기

이 코드는 좀 지저분하지만 시뮬레이션 용도로는 충분합니다. 우리는 우리 서버가 인식할 두번째 요청인 `sleep` 을 생성하고, `/sleep` 으로의 요청을 처리할 `else if` 를 `if` 블록 뒤에 추가했습니다. 만약 요청이 들어오면, 서버는 HTML 페이지를 렌더링 하기 전에 5초간 대기할 것입니다.

여러분은 우리 서버가 얼마나 부족한지 알 수 있습니다: 실제 라이브러리들은 훨씬 간단한 방법으로 여려개의 요청을 구분할 것입니다!

`cargo run` 를 이용해 서버를 실행시키고, 두 브라우저 창을 엽니다: 하나는

`http://localhost:7878/` 로 접속하고 다른 하나는 `http://localhost:7878/sleep` 으로 접속합니다. 만약 여러분이 `/` URI로 몇번 접속하시면 기존처럼 빠른 응답을 보실 수 있으실 테지만, `/sleep` 으로 접속하고 `/` 로 접속한다면 `sleep` 이 5초동안의 로딩을 끝내고 나서야 `/` 에 대한 응답을 보실 수 있을 겁니다.

우리 웹 서버가 모든 요청들을 느린 요청 뒤에 처리하도록 하는것을 피하는 방법은 여러가지가 있지만, 그중 우리가 사용할 방법은 스레드 풀(thread pool) 입니다.

스레드 풀을 이용한 처리량 증진

스레드 풀은 대기중이거나 작업을 처리할 준비가 되어 있는 스레드들의 그룹입니다. 프로그램이 새 작업을 받았을때, 스레드 풀은 작업을 풀(pool) 안에 있는 스레드중 하나에게 맡기고 해당 스레드가 작업을 처리하도록 합니다. 남은 스레드들은 첫번째 스레드가 처리중인 동안 들어온 작업을 언제든지 처리할 수 있도록 합니다. 첫번째 스레드가 작업을 끝마치면 풀로 돌아와 작업 대기상태가 됩니다. 스레드 풀은 우리가 여러 커넥션들을 동시에 처리할 수 있게 해주고 우리 서버의 처리량을 증가시킵니다.

우린 DoS (Denial of Service) 공격을 막기 위해 풀 안의 스레드 개수에 대한 제한을 작게 둘 것입니다; 만약 우리 프로그램이 각각의 요청이 들어올때마다 새 스레드를 생성한다면 누군가 우리 서버에 10만개의 요청을 보냈을때 우리 서버는 서버의 모든 리소스를 사용하고 모든 요청이 끝날때까지 처리가 계속될 것입니다.

우린 스레드를 제한없이 생성하는것이 아닌 풀 안에서 대기할 고정된 개수의 스레드를 가질 것입니다. 요청이 들어온다면, 요청들은 처리를 위해 풀로 보내지고, 풀에선 들어오는 요청들에 대한 큐(queue)를 유지할 것입니다. 풀 내의 각 스레드들은 이 큐에서 요청을 꺼내서 처리하고 또 다른 요청이 있는지 큐에 물어봅니다. 우린 이 형태를 이용해 동시에 N 개의 요청을 처리할 수 있습니다. 여기서 N 은 스레드의 개수입니다. 만약 각각의 스레드가 응답하는데 오래 걸리는 요청을 처리하게되면 그 다음의 요청들은 여전히 큐에 남아있게 됩니다만, 이전보다 처리할 수 있는 요청은 늘어났습니다

이 기술은 우리 웹서버의 처리량을 증가시킬 수많은 방법중 하나일 뿐입니다. 여러분이 찾으실 다른 방법들은 fork/join 모델과 싱글 스레드 기반 비동기 I/O 모델 등일 것입니다. 만약 여러분이 이러한 내용에 관심이 있으시다면, 다른 해결책들에 대해 좀 더 자세히 찾아보시고 Rust로 구현해 보세요; Rust같은 저레벨 언어로는 이와 같은 방법들이 전부 가능합니다.

스레드 풀을 구현하기 전에, 풀이 어떻게 쓰여야 할지 이야기 해 봅시다. 여러분이 코드를 디자인할때, 클라이언트 인터페이스를 먼저 작성해 보는건 여러분의 디자인에 도움이 될 수 있습니다. 코드의 API 를 작성하여 원하는 방식으로 구성한 다음 기능을 구현하고 공개 API를 디자인하는 대신 해당 구조 내에서 기능을 구현하세요.

12장의 프로젝트에서 테스트 주도 개발을 할때와 흡사하게, 우린 여기서 컴파일러 주도 개발을 할 것입니다. 이는 우리가 원하는대로 기능을 호출하는 코드를 작성하고, 컴파일러로부터의 에러를 조사하여 어떻게 코드를 변화시켜야 작동시킬 수 있을지 알아내는 과정을 말합니다.

요청마다 스레드를 생성할 수 있는 코드 구조

먼저, 모든 연결에 대해 스레드를 새로 생성했을때의 코드는 어떤 모습이 될지 알아봅시다. 물론 앞에서 말했듯이, 이는 스레드들을 무한대로 만들어낼 수 있기 때문에 문제를 해결하기 위한 최종적인 대책은 될 수 없습니다만, 그에 대한 출발점 정도로는 볼 수 있습니다. Listing 20-11은 `main` 함수의 `for` 반복문을 모든 요청에 대해 새 스레드를 생성하도록 변경한 모습을 보여줍니다.

파일명: src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
```

Listing 20-11: 매 요청마다 새 스레드 생성

여러분이 16장에서 배우신대로, `thread::spawn` 은 새 스레드를 생성하고, 내부에 있는 클로저의 코드를 실행합니다. 만약 여러분이 이 코드를 실행하고 브라우저로 `/sleep` 으로 접속하신 후, 둘 이상의 브라우저 탭으로 `/` 에 접속하신다면, `/` 로의 요청이 `/sleep` 이 끝나길 기다리지 않고 완료 되는 것을 보실 수 있을 것입니다. 하지만 말했듯이, 스레드를 무한정 생성하는 것은 결국 시스템의 과부하를 일으킬 것입니다.

유한 스레드 수를 위한 인터페이스 만들기

우린 스레드 풀을 비슷하고 익숙하게 작동하도록 만들어서 스레드 풀 방식으로 변경할때 우리 API를 사용하는 코드를 크게 변경할 필요가 없도록 하고자 합니다. Listing 20-12는 `thread::spawn` 대신 이용하고자 하는 `ThreadPool` 이라는 가상의 인터페이스를 보여줍니다.

파일명: src/main.rs

```

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}

```

Listing 20-12: 우리의 이상적인 `ThreadPool` 인터페이스

우린 새로운 스레드 풀을 만들때 `ThreadPool::new` 를 설정할 스레드의 개수를 나타내는 수(이 경우는 4)와 함께 사용했습니다. 그 후 `for` 반복문에선 `thread::spawn` 과 비슷한 인터페이스를 가진 `pool.execute` 에 풀이 각각의 스트림에 대해 실행해야 할 클로저를 넘겨줍니다. 우린 이제 `pool.execute` 를 클로저를 받고 풀 안의 스레드에게 넘겨주어서 실행하도록 구현해야 합니다. 이 코드는 아직 컴파일 되지 않지만 컴파일러가 문제를 해결하는 방법을 안내 할 수 있도록 노력할 것입니다.

ThreadPool 구조체를 컴파일러 주도 개발을 이용해 제작

`src/main.rs` 를 Listing 20-12와 같이 변경하고, `cargo check` 로 얻은 컴파일러 에러를 이용해 개발을 진행해 봅시다. 여기 우리가 얻은 첫번째 에러가 있습니다.

```

$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve. Use of undeclared type or module
`ThreadPool'
--> src\main.rs:10:16
 |
10 |     let pool = ThreadPool::new(4);
      ^^^^^^^^^^^^^^ Use of undeclared type or module
`ThreadPool'

error: aborting due to previous error

```

훌륭합니다. 이 에러는 우리가 `ThreadPool` 타입이나 모듈이 필요하다고 알려주고 있으니 지금 하나 만들어 봅시다. 우리가 만든 `ThreadPool` 은 우리의 웹 서버가 하는 일의 성향과는 독립되어 있어야 합니다. 그러니 `hello` 크레이트를 바이너리 크레이트에서 라이브러리 크레이트로 변경하여 `ThreadPool` 구현을 유지합시다. 라이브러리 크레이트로 변경한 뒤에는, 우린 분리된 스레드 풀 라이브러리를 웹 요청을 처리하는 것 만이 아닌 우리가 스레드 풀을 사용하길 원하는 어떤 작업에서든 사용할 수 있습니다.

가장 간단한 **ThreadPool** 구조체 정의가 포함된 *src/lib.rs* 를 생성합니다.

Filename: *src/lib.rs*

```
pub struct ThreadPool;
```

그 후 *src/bin* 이라는 새 디렉토리를 생성하고 *src/main.rs* 바이너리 크레이트를 *src/bin/main.rs* 의 위치로 이동시킵니다. 이로써 *hello* 디렉토리 안의 라이브러리 크레이트가 주요 크레이트가 될 것입니다; 우린 여전히 *src/bin/main.rs* 바이너리 크레이트를 **cargo run** 명령어를 이용해 실행시킬 수 있습니다. *main.rs* 파일을 이동시킨 후 라이브러리 크레이트를 가져와서 *src/bin/main.rs* 상단에 다음 코드를 추가하여 **ThreadPool** 을 스코프 내로 가져옵니다:

파일명: *src/bin/main.rs*

```
extern crate hello;
use hello::ThreadPool;
```

이 코드는 여전히 작동하지 않지만, 다음 오류를 확인하기 위해 다시 확인해 보겠습니다.

```
$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for type
`hello::ThreadPool` in the current scope
--> src/bin/main.rs:13:16
   |
13 |     let pool = ThreadPool::new(4);
   |             ^^^^^^^^^^^^^^^^^^^ function or associated item not found in
`hello::ThreadPool`
```

이 에러는 우리가 **ThreadPool** 의 **new** 함수를 생성해야 한다는 것을 나타냅니다. 우리는 **new** 가 **4** 를 인수로 받을 수 있도록 하나의 인자를 가져야 하고 **ThreadPool** 객체를 반환해야 한다는 것을 알고 있으니 해당하는 특성을 가진 가장 간단한 **new** 함수를 구현해 봅시다.

파일명: *src/lib.rs*

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

스레드의 개수가 음수라는 것은 말이 안되기 때문에 `size` 인자의 타입을 `usize`로 정했습니다. 3장의 "정수 타입" 절에서 설명했듯이 이 4라는 숫자를 `usize` 타입의 용도에 걸맞게 스레드 컬렉션 요소의 개수로 사용합니다.

코드를 다시한번 체크해 봅시다:

```
$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
--> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |           ^^^^
  |
= note: #[warn(unused_variables)] on by default
= note: to avoid this warning, consider using `_size` instead

error[E0599]: no method named `execute` found for type `hello::ThreadPool` in
the current scope
--> src/bin/main.rs:18:14
  |
18 |         pool.execute(|| {
  |             ^^^^^^
```

이제 경고와 에러가 발생합니다. 경고는 잠시 무시하고, 에러는 `ThreadPool`에 `execute` 메소드가 없기 때문에 발생한 것을 볼 수 있습니다. "유한 스레드 수를 위한 인터페이스 만들기" 절에서 우리가 만들 스레드 풀이 `thread::spawn`과 비슷한 인터페이스를 가져야 한다고 결정했던걸 기억하세요. 또한 `execute` 함수를 구현하여 전달된 클로저를 풀의 유휴 스레드로 전달할 것입니다.

`ThreadPool`에 `execute` 메소드를 매개변수로 클로저를 전달받도록 정의합시다. 13장의 "제네릭 파라미터와 Fn 트레이트를 사용하여 클로저 저장하기" 절에서 클로저를 매개변수로 받을 때 `Fn`, `FnMut`, `FnOnce` 3가지의 트레이트가 있다고 했던걸 상기하세요. 우린 여기서 어떤 종류의 클로저를 사용할지 결정해야 합니다. 우린 표준 라이브러리인 `thread::spawn` 구현체와 비슷하게 만들것이기 때문에 `thread::spawn`의 매개변수가 어떻게 되어 있는지 참고할 수 있습니다. 문서는 다음과 같은 내용입니다.

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static
```

`F` 타입 인자가 바로 우리가 찾는 녀석입니다. `T` 타입 인자는 반환값과 연관된 인자니 관심을 가지지 않아도 됩니다. 우린 `spawn`이 `F`의 트레이트으로 `FnOnce`을 사용하는 것을 알 수 있는데, 이게 바로 우리가 찾는 내용입니다. 왜냐하면 우린 결국 `spawn`에 `execute` 인수를 전달해야하니까요. 또한 스레드가 요청을 처

리할때 요청 클로저를 한번만 실행할 것이기 때문에 `Once`에 매치되는 `FnOnce`가 우리가 원하던 트레이트라고 확신할 수 있습니다.

`F` 타입 인자는 `Send` 트레이트과 `'static` 생명주기가 바인딩되어 있습니다. 한 스레드에서 다른 스레드로 클로저를 전달해야하기에 `Send`가 필요하고 스레드가 언제 파괴될지 모르기 때문에 `'static'`이 필요합니다. `execute` 메소드를 `ThreadPool`에 생성하고 이들이 바인딩된 `F` 타입 제네릭 인자를 받도록 합시다.

파일명: src/lib.rs

```
impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
    }
}
```

우린 클로저가 인자를 받지 않고 반환값도 없기 때문에 `FnOnce` 뒤에 `()`를 사용합니다. 이처럼 함수의 반환값은 생략될 수 있습니다. 하지만 인자가 없더라도 괄호는 필요합니다.

이는 `execute` 메소드의 가장 간단한 구현입니다. 이 코드는 아무것도 하지 않지만, 우리 코드를 컴파일 시도해 볼 수 있습니다. 다시 한번 체크해봅시다.

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
--> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |           ^
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

warning: unused variable: `f`
--> src/lib.rs:8:30
  |
8 |     pub fn execute<F>(&self, f: F)
  |           ^
  |
  = note: to avoid this warning, consider using `_f` instead
```

이제 경고만 받았으니, 컴파일에 성공했다는 뜻입니다! 하지만 여러분이 만약 `cargo run` 을 실행하고 브라우저로 요청을 보내보시면, 이 장의 초반에서 본 에러를 받게되실겁니다. 우리 라이브러리는 `execute` 로 전달된 클로저를 실행하지 않기 때문입니다.

Note: 여러분이 만약 하스켈이나 러스트같이 엄격한 컴파일러를 사용하는 언어를 사용하신다면, "코드가 컴파일이 되면, 작동한단 뜻입니다." 라는 말이 통용됩니다. 하지만 이게 항상 적용되는게 아닌것이, 우리 프로젝트는 컴파일 되었지만 아무것도 하지 않습니다. 만약 우리가 실제 완성을 목표로 프로젝트를 제작중이었다면 이 상황은 코드가 컴파일되는지 체크하는것에 대해서 우리가 원하는 기능이 구현됐는지 확인하기 위해 유닛테스트를 진행하기 시작할 좋은 기회가 될 것입니다.

new 의 스레드 개수에 대한 유효성 검사

`new` 와 `execute` 의 파라미터로 아무것도 하지 않기 때문에 여전히 경고가 나타납니다. 이제 우리가 원하는 기능을 이 함수의 몸체부분에 구현해봅시다. 시작하기 전에, `new` 에 대해서 생각해보죠. 이전에 우리가 스레드풀의 스레드 개수가 음수라는건 말이 안되기 때문에 `size` 인자를 양수형 타입으로 정한것을 기억하시나요? 어쨌든, 스레드가 하나도 없는것도 말이 안되는건 마찬가지입니다. `usize` 타입엔 0이 들어갈 수 있으므로, 우린 예시 20-13 처럼 `ThreadPool` 인스턴스를 반환하기 이전에 `size` 가 0보다 큰지 검사하고, 0일 경우 `assert!` 매크로를 이용해 프로그램 패닉을 일으키는 코드를 추가할 것입니다.

파일명: src/lib.rs

```
impl ThreadPool {
    /// 새 스레드풀 생성
    ///
    /// size 는 풀 안의 스레드 개수입니다.
    ///
    /// # Panics
    ///
    /// `new` 함수는 size 가 0일때 패닉을 일으킵니다
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --생략--
}
```

Listing 20-13: `ThreadPool::new` 를 `size` 가 0일 경우 패닉을 일으키도록 구현

`ThreadPool` 에 문서 주석(doc comments)을 좀 추가해 봤습니다. 14장에서 논의했듯이 우리 함수가

패닉을 일으킬 수 있는 상황을 설명하는 절을 추가함으로써 좋은 문서화방법을 따랐습니다. `cargo doc --open` 을 입력하고 `ThreadPool` 구조체를 클릭한 뒤 `new` 에 대한 문서가 어떻게 만들어 졌는지 확인해보세요!

위에서 한 것처럼 `assert!` 매크로를 추가하는 대신에, `new` 를 예제 12-9 의 I/O 프로젝트의 `Config::new` 처럼 `Result` 를 반환하도록 바꿔봅시다. 하지만 이처럼 스레드풀을 스레드 없이 생성하려 하는것은 회복할 수 없는(unrecoverable) 에러가 되어야 합니다. 만약 여러분이 오기가 생기신다면, `new` 를 다음과 같이 시그니처를 만든 새 버전을 작성해 보시고, 두 버전을 비교해보세요.

```
pub fn new(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

스레드를 보관하기 위한 공간 생성하기

이제 우린 스레드풀에 보관할 스레드의 개수가 유효하단 것을 확인했으니, 반환하기 전에 `ThreadPool` 구조체에 스레드들을 생성하고 보관해 놓을 수 있습니다. 그런데, 어떻게 스레드를 "보관" 할까요? `thread::spawn` 의 시그니처를 다시 살펴봅시다.

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static
```

`spawn` 함수는 `JoinHandle<T>` 를 반환합니다. 여기서 `T` 는 클로저가 반환할 타입입니다. `JoinHandle` 을 사용해보고 무슨 일이 일어나는지 살펴봅시다. 우리의 경우, 스레드풀로 전달된 클로저는 커넥션을 다루고 아무것도 반환하지 않을테니 `T` 는 `()` 가 되겠네요.

Listing 20-14의 코드는 컴파일엔 문제가 없지만 아직 아무 스레드도 만들지 않습니다.

`thread::JoinHandle<()>` 객체를 담는 벡터를 취급하도록 `ThreadPool` 의 정의를 변경했습니다. 벡터의 크기를 `size` 로 초기화하고 `for` 반복문에서 스레드들을 생성한뒤, 스레드들을 가진 `ThreadPool` 객체를 반환할 것입니다.

Filename: src/lib.rs

```

use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --생략--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // 스레드들을 생성하고 벡터 내에 보관합니다
        }

        ThreadPool {
            threads
        }
    }
    // --생략--
}

```

Listing 20-14: `ThreadPool` 에 스레드들을 보관하기 위한 벡터 만들기

`std::thread` 를 라이브러리 크레이트의 스코프 내로 가져왔습니다. 우리가 `thread::JoinHandle` 를 `ThreadPool` 내 벡터 요소의 타입으로 사용하고 있으니까요.

`ThreadPool` 은 유효한 숫자를 전달받을 경우 `size` 크기대로 새로운 벡터를 생성합니다. 이 책에선 아직 `Vec::new` 와 비슷한 기능을 하는 `with_capacity` 함수를 사용하진 않았습니다: 다만 이 두 함수는 중요한 차이가 있는데, `with_capacity` 함수는 벡터의 공간을 미리 할당합니다. 우린 벡터 안에 들어갈 요소의 개수를 알고 있기 때문에 사전에 공간을 할당 함으로써 요소의 삽입마다 재할당이 일어나는 `Vec::new` 를 사용할 때 보다 효율을 높일 수 있습니다.

`cargo check` 를 재실행 하시면 몇개의 경고는 발생하겠지만 문제 없이 성공하실겁니다.

ThreadPool에서 스레드로 코드를 보내는 Worker 구조체

Listing 20-14 의 `for` 반복문에 스레드 생성에 관해 주석을 남겨놨습니다. 여기서 우리가 스레드들을 실제로 만드는 방법을 알아볼 예정입니다. 표준 라이브러리는 `thread::spawn` 를 이용해 스레드를 생성할 수 있도록 제공하며, `thread::spawn` 은 스레드가 생성되는 즉시 스레드가 실행할 코드를 전달 받도록 되어 있습니다. 하지만 우린 스레드를 생성하고 나중에 코드를 전달받을 때까지 `기다리도록` 해야 합니다. 안타깝게

도 표준 라이브러리의 스레드 구현에는 이러한 방법을 지원하지 않아서 우리가 직접 구현해야합니다.

우린 이 기능을 `ThreadPool` 과 스레드들 사이에 새로운 데이터 구조를 도입하여 구현할 것입니다. 앞으로 이 데이터 구조를 `Worker` 라고 부르겠습니다. 이 용어는 풀링 구현에서 흔하게 사용됩니다. 한번 식당의 부엌에서 일하는 사람들을 예로 들어보죠. 이 `Worker` 들은 고객으로부터 주문을 받을 때까지 기다린 다음, 주문을 받고 일합니다. 뭐 대충 비슷하지 않나요?

스레드 풀 안에 `JoinHandle<()>` 인스턴스 벡터 대신, `Worker` 구조체의 인스턴스들을 내장하도록 해 봅시다. 이때 각각의 `Worker` 는 단일 `JoinHandle<()>` 인스턴스를 내장하게 됩니다. 그리고 실행할 코드의 클로저를 전달받고 스레드에게 전달해 실행하도록 하는 함수를 `Worker` 에 구현할 것입니다. 또한 우린 각각의 워커에 `id` 를 부여해 로그를 남기거나 디버깅을 할때 서로 다른 워커들을 구별할 수 있게 할 것입니다.

`ThreadPool` 을 생성할때 일어나는 일을 다음과 같이 변경해 보겠습니다. 다음과 같은 방법으로 `Worker` 를 설정하고 스레드에 클로저를 전송하는 코드를 구현합니다.

1. `id` 와 `JoinHandle<()>` 를 갖는 `Worker` 구조체를 정의 합니다.
2. `ThreadPool` 을 `Worker` 인스턴스들의 벡터를 갖도록 변경합니다.
3. `id` 숫자를 받고 전달받은 `Worker` 인스턴스를 반환하는 `Worker::new` 함수를 정의합니다. 반환된 `Worker` 인스턴스에는 `id` 와 빈 클로저로 생성된 스레드가 포함되어 있습니다.
4. `ThreadPool::new` 안에서, `for` 루프 카운터를 이용해 `id` 를 생성하고 생성된 `id` 를 이용해 새 `Worker` 를 생성한 뒤 해당 워커를 벡터안에 저장합니다.

도전해보실 분은 Listing 20-15 코드를 보기 전에 직접 구현해보시길 바랍니다.

준비 되셨나요? 여기 앞선 수정사항들을 구현한 방법중 하나로 Listing 20-15 를 가져와 보았습니다.

파일명: src/lib.rs

```

use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --생략--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers
        }
    }
    // --생략--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker {
            id,
            thread,
        }
    }
}

```

Listing 20-15: `ThreadPool` 을 스레드들을 직접 내장하는 대신 `Worker` 인스턴스들을 내장하게 변경

이제 `JoinHandle<()>` 인스턴스들이 아닌 `Worker` 인스턴스들을 내장하기 때문에 `ThreadPool` 의 필드 이름을 `threads`에서 `workers`로 변경했습니다. 우린 `for` 반복문으로 `Worker::new`에 전달된 인자만큼 카운트하고 각각의 새 `Worker`를 `workers` 벡터에 저장합니다.

외부 코드 (우리 서버의 `src/bin/main.rs` 같은) 예선 `ThreadPool` 내 `Worker` 구조체의 상세한 구현을 알 필요가 없기에 `Worker` 구조체와 `new` 함수를 `private`로 만듭니다. `Worker::new` 함수는 우리가

넘겨준 `id` 를 사용하고 새로 생성된 `JoinHandle<()>` 객체를 저장합니다. 이때 `JoinHandle<()>` 객체를 생성한 주체는 빈 클로저를 이용해 생성된 새 스레드입니다.

이 코드는 컴파일 되고 `Thread::new` 의 인자로서 지정된 `Worker` 인스턴스의 개수를 저장합니다. 하지만 우린 여전히 `execute`에서 전달받은 클로저를 처리하지 않고 있습니다. 다음에 그 작업을 수행하는 방법을 살펴 보겠습니다.

채널을 통해 스레드에 요청 보내기

이제 우린 `thread::spawn`에 주어진 클로저가 아무것도 하지 않는다는 문제점을 해결할 겁니다. 현재 우린 실행하고 싶은 클로저를 `execute` 메소드로 받고 있습니다. 하지만 우리가 `thread::spawn`에 전달할 클로저는 `ThreadPool`의 생성중 각각의 `Worker` 가 생성될 때 실행할 클로저여야 합니다.

우리는 방금 생성한 `Worker` 구조체가 `ThreadPool`에 들어있는 큐에서 실행될 코드를 가져오고 그 코드를 스레드로 보내 실행하도록 하고자 합니다.

16장에서, 간단히 두 스레드간에 통신하는 방법인 *channels*에 대해 배웠습니다. 이거 지금 상황에 딱이네요. 우린 채널을 작업 대기열로 사용하고 `execute`는 `ThreadPool`에서 `Worker` 인스턴스로 작업을 보냅니다. 그러면 작업이 스레드로 전송되겠죠. 계획은 다음과 같습니다:

1. `ThreadPool`은 채널을 생성하고 채널의 송신단을 유지합니다.
2. 각 `Worker`는 채널의 수신단을 유지합니다.
3. 우린 채널로 전송하려는 클로저를 저장할 새로운 `Job` 구조체를 생성할 겁니다.
4. `execute` 메소드는 채널의 송신단으로 실행하려는 작업을 전송합니다.
5. 스레드에선 `Worker`가 채널의 수신단에서 반복되며 수신되는 모든 작업의 클로저를 실행합니다.

`ThreadPool::new` 안에 채널을 생성하고 `ThreadPool` 객체에 송신단을 내장하도록 하는 것부터 시작해 봅시다. Listing 20-16에 보이는 것처럼 `Job` 구조체는 아직은 아무것도 들어있지 않지만, 우리가 채널에 보낼 요소의 타입이 될 것입니다.

파일명: src/lib.rs

```
// --생략--
use std::sync::mpsc;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --생략--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --생략--
}
```

Listing 20-16: `ThreadPool` 이 `Job` 객체를 전송하는 채널의 송신 측을 저장하도록 변경

`ThreadPool::new`에서 우린 새 채널을 만들고 풀에 송신단을 저장합니다. 이 소스는 경고 몇개와 함께 성공적으로 컴파일됩니다.

한번 스레드풀이 생성한 각각의 `worker`에 채널의 수신단을 넘겨봅시다. 우린 `worker`가 생성한 스레드에서 수신단을 사용할 수 있게 만들기 위해, 클로저의 `receiver` 매개변수를 참조 할 것입니다. Listing 20-17의 코드는 아직 컴파일 되지 않습니다.

파일명: `src/lib.rs`

```

impl ThreadPool {
    // --생략--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --생략--
}

// --생략--


impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker {
            id,
            thread,
        }
    }
}

```

Listing 20-17: *worker*에 채널의 송신단을 전달

작고 간단한 변경사항을 만들었습니다: `Worker::new`에 채널의 수신단을 전달하고, 클로저 안에서 사용합니다.

이 코드를 `check` 해보면, 다음과 같은 에러가 나타날 겁니다:

```
$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
--> src/lib.rs:27:42
   |
27 |         workers.push(Worker::new(id, receiver));
   |                           ^^^^^^^^ value moved here in
   | previous iteration of loop
   |
   = note: move occurs because `receiver` has type
   `std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
```

이 코드는 `receiver` 을 여러개의 `Worker` 객체에 넘기는 시도를 하는데, 이는 작동하지 않습니다. 16장에서 배운걸 떠올려보세요: 러스트가 제공하는 채널 구현은 여러 *producer*, 하나의 *consumer* 를 제공합니다. 즉 이 코드를 수정하기 위해 채널의 소비측 끝을 복제할 수는 없습니다. 만약 가능하더라도 우리가 원하는 기법은 아닙니다; 우린 대신에 하나의 `receiver` 을 모든 `worker` 들이 공유하도록 만들어 스레드간 작업을 분산하고자 합니다.

또한, 채널 큐에서 작업을 가져오는 작업은 `receiver` 을 이용하는데, 이 과정에서 `receiver` 이 변화할 수도 있습니다. 따라서 스레드들은 `receiver` 을 공유하고 수정하기 위한 안전한 방법이 필요합니다. 그렇지 않다면, 경쟁 조건 (관련 내용은 16장에서 다뤘습니다) 이 발생하게 될 것 입니다.

16장에서 설명한 스레드-안전 스마트 포인터를 생각해 보세요: 이는 여러 스레드 간에 소유권을 공유하고 스레드가 값을 변경하도록 허용합니다. 우린 `Arc<Mutex<T>>` 를 사용해야 하는데, 이 `Arc` 타입은 여러 `worker` 들이 `receiver` 를 소유하는 걸 허용해줍니다. 그리고 `Mutex` 는 한번에 하나의 `worker` 만이 `receiver`로부터 작업을 가져가도록 보장합니다. Listing 20-18 은 위 내용대로 수정한 모습입니다.

파일명: src/lib.rs

```

use std::sync::Arc;
use std::sync::Mutex;
// --생략--

impl ThreadPool {
    // --생략--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --생략--
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --생략--
    }
}

```

Listing 20-18: `Arc` 와 `Mutex` 를 이용해 채널의 를 `worker` 들이 공유하도록 변경

`ThreadPool::new` 에서, `Arc` 와 `Mutex` 를 이용해 채널의 수신단을 감싸고, 새로운 `worker` 각각에 `Arc` 를 복제해 참조 카운트를 늘려서 `worker` 들이 소유권을 공유할 수 있도록 합니다.

이렇게 변경하고 나면 컴파일이 될 겁니다! 우리가 해냈네요!

execute 메소드 구현

이제 마지막으로 `ThreadPool` 의 `execute` 메소드를 구현해 봅시다. 우린 `execute` 가 받는 클로저 타입을 포함할 트레이트 오브젝트를 위해 `Job` 을 구조체에서 타입 별칭으로 변경할 겁니다. 19장의 “Type Aliases Create Type Synonyms” 부문에서 이야기한 대로, 타입 별칭은 긴 이름을 가진 타입을 짧게 만

들 수 있게 해줍니다. Listing 20-19에서 확인해 보세요.

파일명: src/lib.rs

```
// --생략--

type Job = Box<FnOnce() + Send + 'static>;

impl ThreadPool {
    // --생략--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --생략--
```

예제 20-19: 각 클로저를 내장한 `Box`에 대한 타입 별칭인 `Job`을 만들고 채널에 작업 전송하기

`execute`로 얻은 클로저를 사용하여 새 `Job` 객체를 생성하고 나면, 생성된 작업을 채널의 송신 측으로 보냅니다. `send`가 실패할 경우엔 `unwrap`을 호출합니다. 예를 들어 실행중인 모든 스레드가 중지되면 수신단이 새로운 메시지를 수신하지 못하게 될 수 있습니다. 현재, 우린 실행중인 스레드들을 멈출 수 없습니다: 스레드들은 풀이 존재하는 한 계속 실행될 겁니다. `unwrap`을 사용하는 이유는 실패 사례가 발생하지 않을 것이란걸 우린 알고 있지만 컴파일러는 모르기 때문입니다.

아직 끝나지 않았습니다! `worker`에서, 우리 클로저는 `thread::spawn`에 전달되어 여전히 채널의 수신단만 참조 합니다. 대신에 클로저가 계속 반복되며 채널의 수신단에 작업을 요청하고 받은 작업을 실행해야 합니다. Listing 20-20에서 `Worker::new`를 봅시다.

파일명: src/lib.rs

```
// --생략--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {} got a job; executing.", id);

                (*job)();
            }
        });
        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-20: *worker* 스레드에서 작업을 수신하고 실행하기

맨 처음, 뮤텍스를 얻기 위해 `receiver` 의 `lock` 을 호출 하였고, 그 뒤 `unwrap` 을 이용해 어떤 에러든 패닉을 일으키도록 하였습니다. 만약 어떤 스레드에서 잠금을 걸고 나서 해제하기 전에 패닉 상태가 되어 뮤텍스가 *poisoned* 상태가 되었을 경우 뮤텍스를 얻는데 실패할 수 있기 때문에, 이 경우 `unwrap` 을 호출해 스레드 패닉을 발생시키는 것이 취해야 할 올바른 행동입니다. 원하실 경우 `unwrap` 을 여러분에게 의미있는 에러 메세지와 함께 `expect` 로 바꾸어 보세요.

만약 우리가 뮤텍스의 잠금을 얻게 된다면, 채널로부터 `Job` 을 얻기 위해 `recv` 를 호출합니다. 마지막 `unwrap` 은 송신단을 유지하는 스레드가 종료 되었을 경우 발생할 수 있는 에러를 지나쳐서 이동합니다. `send` 메소드가 수신단이 종료되면 `Err` 을 리턴하는 것과 비슷합니다.

아직 아무 작업도 없어서 `recv` 에 대한 호출이 막힌다면, 현재 스레드는 작업이 가능해질 때까지 대기합니다. `Mutex<T>` 가 한번에 오직 하나의 `Worker` 스레드가 작업을 요청할 수 있도록 보장합니다.

이론적으로 이 코드는 컴파일 되어야 합니다. 하지만 불행하게도 러스트 컴파일러는 아직 완벽하지 않습니다. 나타나는 에러는 다음과 같습니다:

```
error[E0161]: cannot move a value of type std::ops::FnOnce() +
std::marker::Send: the size of std::ops::FnOnce() + std::marker::Send cannot
be
statically determined
--> src/lib.rs:63:17
   |
63 |         (*job)();
   |         ^^^^^^
```

문제가 상당히 난해하기 때문에 이 오류는 상당히 수수께끼스럽습니다. `Box<T>` (`Job` 가 가리키는 그것) 안에 저장되어 있는 `FnOnce` 클로저를 호출하기 위해선, 클로저는 `Box<T>`에서 스스로 벗어나야 합니다. 우리가 호출할 때 클로저는 `self`의 소유권을 가지기 때문이죠. 보편적으로, 러스트는 `Box<T>`의 값을 옮기는 것을 허용하지 않습니다. 러스트에서 `Box<T>` 안에 얼마나 큰 값이 들어갈지 알 수 없기 때문입니다: 15장에서 우리가 박스에 저장하고자 하는 알수없는 크기의 무언가를 알고 있는 크기의 값으로 얻어내기 위해 `Box<T>`를 사용했던 걸 떠올려 보세요.

여러분이 Listing 17-15에서 보신 것처럼 우린 `self: Box<Self>` 구문을 이용하는 메소드를 작성할 수 있습니다. `Box<T>`에 저장된 `Self` 값의 소유권을 다룰 수 있도록 허용된 메소드 말이죠. 우리가 지금 하고 싶은 것 그 자체네요. 그런데 불행히도 러스트가 우릴 놓아주지 않네요: 러스트의 구현체중 클로저가 호출될 때의 구현체 부분은 `self: Box<Self>` 방식을 사용하지 않았습니다. 따라서 러스트는 이 상황에서 `self: Box<Self>`를 사용할 수 있다는 것을 아직 이해하지 못합니다.

러스트의 컴파일러는 여전히 개선중입니다. 따라서 언젠가 Listing 20-20의 코드는 정상적으로 작동할 거예요. 여러분 같은 사람들이 이런 문제를 해결하기 위해 노력중입니다! 여러분이 이 책을 끝내고 나서 참여하신다면 우린 환영할 겁니다.

하지만 지금 당장은, 편리한 트릭을 이용해 이 문제를 해결하도록 하겠습니다. 우린 러스트에게 명시적으로 이러한 경우에 우린 `self: Box<Self>`를 이용해 `Box<T>` 내부의 값에 대한 소유권을 가질 수 있다고 말할 수 있습니다; 클로저에 대한 소유권을 가진 뒤에는 호출할 수 있습니다. 이는 `call_box` 메소드로 새로운 트레이트인 `FnBox`를 정의하는 것입니다. `call_box`는 시그니처로 `self: Box<Self>`를 사용하고 `FnOnce()`를 구현하는 모든 타입에 `FnBox`를 정의하고 타입 별명을 새 트레이트로 변경하고 `Worker`를 `call_box` 메소드를 사용하도록 변경할 것입니다. 이 내용들을 Listing 20-21에서 보실 수 있습니다.

파일명: src/lib.rs

```

trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<FnBox + Send + 'static>;

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });
        Worker {
            id,
            thread,
        }
    }
}

```

Listing 20-21: `Box<FnOnce()>` 의 한계를 해결하기 위한 새 트레이트 `FnBox` 추가

먼저, `FnBox` 라는 이름의 새 트레이트를 생성합니다. 이 트레이트는 `call_box` 메소드를 하나 가집니다. 이 메소드는 `self` 의 소유권을 다루고 `Box<T>`에서 값을 제외하기 위해 `self: Box<Self>`를 다룬다는 점 외에는 다른 `Fn*` 트레이트들의 `call` 메소드와 흡사합니다.

다음으로, `FnOnce()` 트레이트를 구현하는 `F` 타입에 대한 `FnBox` 트레이트를 구현합니다. 효과적으로, 이는 `FnOnce()` 클로저가 우리의 `call_box` 메소드를 사용할 수 있다는 뜻입니다. `call_box`의 구현은 `(*self)()`를 사용하여 클로저를 `Box<T>` 밖으로 빼내고 호출합니다.

우리는 이제 `Job` 타입 별명이 새로운 트레이트인 `FnBox`를 구현하는 `Box`가 될 필요가 있습니다. 이는 클로저를 직접 호출하는 대신 `Job` 값을 얻을 때 `Worker`에서 `call_box`를 사용할 수 있게 해줍니다. 어떤 `FnOnce()` 클로저에 대한 `FnBox` 트레이트를 구현한다는 것은 채널을 보내고 있는 실제 값에 대해서는 아무

것도 변경할 필요가 없다는 것을 의미합니다. 이제 러스트는 우리가 하고자 하는 일이 문제 없단걸 정상적으로 인식할 수 있습니다.

이 트릭은 매우 복잡하고 교활합니다. 정확히 이해가 되지 않아도 걱정하지 마세요; 언젠가 완전히 필요 없어질 겁니다.

이 트릭을 구현함으로써, 우리의 스레드 풀은 작동합니다. `cargo run` 을 실행하고, 몇가지 요청을 해보세요.

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never used: `workers`
--> src/lib.rs:7:5
  |
7 |     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^
  |
= note: #[warn(dead_code)] on by default

warning: field is never used: `id`
--> src/lib.rs:61:5
  |
61|     id: usize,
  |     ^^^^^^^
  |
= note: #[warn(dead_code)] on by default

warning: field is never used: `thread`
--> src/lib.rs:62:5
  |
62|     thread: thread::JoinHandle<()>,
  |     ^^^^^^^^^^^^^^^^^^
  |
= note: #[warn(dead_code)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs
    Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

성공했습니다! 우린 이제 비동기적으로 커넥션을 실행하는 스레드 풀을 가지게 되었습니다. 스레드는 4개 이상 만들어지지 않을 겁니다. 따라서 우리 시스템은 서버가 수많은 요청을 받더라도 과부하 될 일이 없습니다. 만약 우리가 `/sleep` 요청을 하더라도, 다른 스레드가 작동함으로써 다른 요청에 문제 없이 작동합니다.

18장에서 `while let` 반복문을 배우셨다면, 아마 제가 왜 Listing 20-22와 같이 *worker* 스레드의 코드를 작성하지 않았는지 궁금해 하실 겁니다.

파일명: src/lib.rs

```
// --생략--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-22: `while let` 을 이용한 `Worker::new` 의 대안 구현체

이 코드는 컴파일 될 것이나 원하던 스레딩 동작은 하지 않습니다. 느린 요청은 여전히 다른 요청들이 처리되길 기다립니다. 이유는 다소 미묘합니다: `Mutex` 구조체는 공개된(public) `unlock` 메소드를 가지고 있지 않습니다. `lock`의 소유권은 `LockResult<MutexGuard<T>>` 에 있는 `MutexGuard<T>` 의 라이프타임에 기반을 두고 있기 때문입니다. 컴파일시, 빌림 검사기 (borrow checker) 는 잠금을 유지하지 않으면 `Mutex`에 의해 보호받는 리소스에 접근할 수 없다는 규칙을 적용할 수 있습니다. 그러나 이러한 구현은 `MutexGuard<T>`의 라이프타임을 주의 깊게 생각하지 않았을 경우 잠금이 의도보다 오래 지속되는 결과를 초래할 수 있습니다. `while` 식 안의 값은 블록의 지속 시간동안 남아있기 때문에, 잠금은 `job.call_box()` 호출 기간동안 유지되어 다른 *worker* 들이 작업을 수행할 수 없음을 뜻합니다.

대신에 `loop` 를 사용하고 잠금과 작업을 얻음으로써 외부가 아닌 블록 내에서 얻음으로써, `lock` 메소드에서 반환 된 `MutexGuard` 는 `let job` 문이 끝나자 마자 사라지게 됩니다. 이렇게하면 잠금이 `recv`에 대한 호출 중에 해제되지만, `job.call_box()` 호출 이전에 해제되어 여러 요청을 동시에 처리 할 수 있습니다.

우아한 종료와 정리

Listing 20-21 의 코드는 우리가 의도한대로 스레드 풀을 이용해 비동기적으로 요청에 응답합니다. 다만 우린 `workers`, `id`, `thread` 필드를 직접적으로 사용하지 않고 있다는 경고를 받는데, 이는 우리가 아무것도 정리하질 않았다는 것을 상기시킵니다. 예를 들어 우리가 `ctrl-c` 처럼 우아하지 않은 방식으로 메인 스레드를 정지 시킬 경우 모든 스레드는 즉시 정지됩니다. 만약 그 스레드가 요청을 처리하는 도중 이더라도요.

이제 우린 풀 안의 각 스레드 상에서 `join` 을 호출하여 스레드가 종료되기 전에 그들이 처리하던 요청을 먼저 처리할 수 있도록 하기 위하여 `Drop` 트레이잇을 구현할 겁니다. 그런 다음 스레드들에게 더 이상 새로운 요청을 받지 말고 종료하라고 알려주는 방법을 구현할 것입니다. 이 코드가 작동하는지 확인하기 위해, 정상적으로 스레드 풀을 종료하기 전에 오직 두개의 요청만 수락하도록 우리 서버를 수정합시다.

ThreadPool 에 Drop 트레이잇 구현하기

우리가 만든 스레드 풀에 `Drop` 을 구현하는 것부터 시작해봅시다. 풀이 드롭(dropped) 되었을 때, 스레드들은 모두 `join` 되어 자신의 작업을 마쳐야 합니다. Listing 20-23 은 `Drop` 을 구현한 첫 시도의 모습입니다; 이 코드는 아직 작동하지 않습니다.

파일명: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

Listing 20-23: 스레드 풀이 스코프를 벗어날때 각 스레드 종료

먼저, 스레드 풀의 `workers` 각 요소에 대한 반복문을 정의합니다. `self` 가 가변 참조자이고, 우리가 `worker` 를 변경할 수 있도록 해야 하므로 `&mut` 를 사용했습니다. 각각의 `worker`에 대해서는 이 `worker`가 종료된다는 메시지를 출력하고 해당 `worker`의 스레드에 `join` 을 호출합니다. 만약 `join` 을 호출하는데 실패하면, `unwrap` 을 이용해 Rust 패닉을 일으키고 강제 종료합니다.

이 코드를 컴파일 했을 때 나오는 에러는 다음과 같습니다

```
error[E0507]: cannot move out of borrowed content
--> src/lib.rs:65:13
   |
65 |         worker.thread.join().unwrap();
   |         ^^^^^^ cannot move out of borrowed content
```

이 에러는 우리가 `worker` 의 가변 형태로 빌리기만 했기 때문에 인수의 소유권을 필요로 하는 `join` 을 호출할 수 없다는 걸 알려줍니다. 이 이슈를 해결하기 위해, `join` 이 스레드를 사용할 수 있도록 `thread` 의 소유권을 `Worker` 인스턴스로부터 빼내야 합니다. 이전에 Listing 17-15 에서 한번 해봤었죠: `Worker` 가 `Option<thread::JoinHandle<()>>` 를 대신 갖도록 하면, `Option` 의 `take` 메소드를 사용하여 `Some` variant에서 값을 빼내고 `None` 으로 대체할 수 있습니다. 즉, 실행중인 `Worker` 는 `thread` 에 `Some` variant 를 갖게 되고, 우린 `worker` 를 종료하고자 할때 `Some` 을 `None` 으로 대체하여 `worker` 가 실행할 스레드를 없앨 수 있습니다.

그러니 `Worker` 의 정의를 다음과 같이 변경합시다:

파일명: src/lib.rs

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

변경해야 하는 나머지 부분은 컴파일러에 의지해서 찾아보도록 합시다. 코드를 `check` 해보니 두 에러가 나온네요:

```

error[E0599]: no method named `join` found for type
`std::option::Option<std::thread::JoinHandle<()>>` in the current scope
--> src/lib.rs:65:27
 |
65 |         worker.thread.join().unwrap();
 |             ^^^^

error[E0308]: mismatched types
--> src/lib.rs:89:13
 |
89 |         thread,
 |         ^^^^^^
 |         |
 |         expected enum `std::option::Option`, found struct
`std::thread::JoinHandle`
 |         help: try using a variant of the expected type:
`Some(thread)`
 |
= note: expected type `std::option::Option<std::thread::JoinHandle<()>>`
        found type `std::thread::JoinHandle<_>`

```

`Worker::new`의 끝에 위치한 두번째 에러부터 해결해 봅시다; `Worker`를 생성할 때 `thread`를 `Some`으로 감싸줘야 한다네요. 다음과 같이 변경해줍시다:

Filename: src/lib.rs

```

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --생략--
        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

첫번째 에러는 우리의 `Drop` 구현에서 발생했네요. 이전에 우리가 `worker`로 부터 `thread`를 빼내기 위해선 `Option`에서 `take`를 호출해야 한다고 언급했습니다. 이는 다음과 같이 변경해줍시다:

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

17장에서 의논한 대로, `Option`의 `take` 메소드는 `Some` variant를 빼내고 `None`으로 대체합니다. `Some`을 파괴하고 스레드를 얻기 위해 `if let`를 사용합니다; 그리고 나서 스레드의 `join`을 호출합니다. 만약 이때 `worker`의 스레드가 이미 `None`일 경우, `worker`가 자신의 스레드를 이미 정리했다는 뜻이므로 아무 일도 하지 않습니다.

스레드가 작업 리스닝을 중지하도록 신호하기

모두 수정하고 나면 경고 없이 컴파일이 잘 될 겁니다. 하지만 안 좋은 소식이 있는데, 이 코드는 아직 우리가 원하는 대로 작동하지 않는다는 겁니다. 이에 대한 핵심은 `Worker` 인스턴스의 스레드에 의해 실행되는 클로저에 있습니다: 우리가 `join`을 호출해도 스레드는 영원히 새 작업을 찾는 일을 반복할 것이기에 스레드는 종료되지 않습니다. 만약 우리가 현재 `drop`의 구현대로 `ThreadPool`을 `drop`한다면, 메인스레드는 첫번째 스레드가 끝나기만을 기다리는 상태로 영원히 멈춰있을 겁니다.

이를 해결하기 위해, 실행할 `Job`이나 리스닝을 멈추고 무한 반복문을 탈출하라는 신호를 기다리도록 스레드를 수정해야 합니다. 우리 채널은 `Job` 인스턴스 대신에 두 variant를 가진 열거형을 전달할 겁니다.

파일명: src/lib.rs

```
enum Message {
    NewJob(Job),
    Terminate,
}
```

이 `Message` 열거형은 스레드가 실행해야 할 `Job`을 담고 있는 `NewJob` variant가 되거나, 혹은 스레드를 중지시킬 `Terminate` variant가 될 겁니다.

우린 Listing 20-24처럼 `Job` 대신 `Message` 타입을 이용하도록 채널을 조정해야 합니다.

파일명: src/lib.rs

```
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

// --생략--

impl ThreadPool {
    // --생략--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(job)).unwrap();
    }
}

// --생략--


impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
        Worker {
        let thread = thread::spawn(move ||{
            loop {
                let message = receiver.lock().unwrap().recv().unwrap();

                match message {
                    Message::NewJob(job) => {
                        println!("Worker {} got a job; executing.", id);

                        job.call_box();
                    },
                    Message::Terminate => {
                        println!("Worker {} was told to terminate.", id);

                        break;
                    },
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

}

Listing 20-24: `Message` 값을 전달하고 받으며 `Worker` 가 `Message::Terminate` 를 받을 경우 반복문 탈출

`Meesage` 열거형을 통합하기 위해, `ThreadPool` 정의와 `Worker::new` 의 시그니처에서 `Job` 을 `Message` 로 변경해야 합니다: `ThreadPool` 의 `execute` 메소드는 `job` 을 `Message::NewJob` variant 로 감싸서 전달해야 합니다. 그리고 `Worker::new` 의 채널로부터 `Message` 를 받는 부분에선 전달 받은게 `NewJob` 일시 작업을 처리할 것이고, `Terminate` 일 경우 스레드는 루프를 탈출 할 겁니다.

변경하고 나면, 이 코드는 컴파일 되고 Listing 20-21 과 똑같이 작동 할 겁니다. 하지만 우리가 `Terminate` 메시지를 아무것도 만들지 않았기 때문에 경고가 나타납니다. 우리 `Drop` 구현체를 Listing 20-25 와 같이 수정해서 경고를 고쳐봅시다.

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");

        for _ in &mut self.workers {
            self.sender.send(Message::Terminate).unwrap();
        }

        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

Listing 20-25: 각 worker 스레드에 `join` 을 호출하기 전에 `Message::Terminate` 전달하기

이제 우린 각 worker 들을 두번 순회합니다: 한번은 각 worker 에 `Terminate` 메시지를 보내기 위해서고 한번은 각 워커의 스레드에 `join` 을 호출하기 위해서입니다. 만약 루프를 한번만 이용해서 메시지를 보내는 동시에 `join` 을 호출한다면 현재 반복되는 worker 가 채널에서 메시지를 새로 가져오려 하는 중이란 걸 보장할 수 없기에 별 효과를 볼 수 없습니다.

반복문을 두번으로 나눈 이유를 좀더 자세히 설명해 보겠습니다. 한번 두 worker 를 상상해 보세요. 만약 우

리가 반복문을 한번만 사용한다면, 첫번째 반복자에서 종료 메시지가 채널로 전송되고 첫 worker 의 스레드에서 **join** 이 호출될 겁니다. 만약 첫번째 worker 가 요청을 처리하느라 바쁠 경우, 두번째 worker 가 채널에서 종료 메시지를 가져와 종료합니다. 우린 첫번째 worker 가 종료되길 기다리지만, 두번째 스레드가 이미 종료 메시지를 가져가는 바람에 첫번째 worker 는 영원히 종료되지 않습니다. 교착상태(Deadlock)에 걸려버렸네요!

이 시나리오를 방지하기 위해서는 하나의 반복문으로 모든 **Terminate** 메시지를 채널에 넣어야 합니다; 그 뒤 다른 반복문으로 모든 스레드에 join 합니다. 각 worker 는 종료 메시지를 받으면 채널로부터의 요청 수신을 중지합니다. 따라서 우린 worker 의 수와 같은 수의 종료 메시지를 보내면 각 worker 는 자신의 스레드에 **join** 이 호출되기 전에 종료 메시지를 수신하게 될 거라고 확신할 수 있습니다.

이 코드가 작동하는 걸 보려면, **main** 을 Listing 20-26 에서 나오는 것처럼 우아하게 종료 되기 전에 오직 두 요청만 받도록 변경해야 합니다.

파일명: src/bin/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

Listing 20-26: 두 요청을 처리하고서 반복문을 탈출하게 하여 서버를 종료

여러분은 실제 웹 서버가 달랑 두개의 요청만 처리하고 종료되는걸 원하진 않을겁니다. 이 코드는 어디까지나 우아한 종료 및 정리 작업이 잘 작동하는지 보기위한 시연용 입니다.

take 메소드는 **Iterator** 트레이트에 정의되어 있으며 반복을 처음 두 항목으로 제한합니다.

ThreadPool 은 **main** 의 끝에서 스코프를 벗어나게 될 것이고, **drop** 이 실행 될 것입니다.

cargo run 으로 서버를 실행시키고, 요청을 3개 생성해 보세요. 세번째 요청은 에러가 날 것이고, 여러분은 터미널에서 다음과 비슷한 내용의 출력을 보게 될 겁니다.

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
  Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
    Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
Shutting down all workers.
Shutting down worker 0
Worker 1 was told to terminate.
Worker 2 was told to terminate.
Worker 0 was told to terminate.
Worker 3 was told to terminate.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

아마 여러분은 작업자와 메시지가 출력된 순서가 다르다는 걸 보셨을 겁니다. 우린 이 메시지로부터 이 코드가 어떻게 작동하는지 알 수 있습니다: worker 0 과 3 이 처음 두 요청을 받고, 그런 다음 3번째 요청에서 서버는 연결 수락(connection accept)을 중지했습니다. `ThreadPool` 이 `main` 의 끝에서 스코프를 벗어나게 되면 `Drop` 이 실행되고, 풀(pool)이 모든 worker 에게 종료 신호를 알립니다. 각 worker 는 자신이 종료 메시지를 받았을때 메시지를 출력하고, 스레드 풀은 각 worker 스레드에 `join` 을 호출합니다.

이 실행 결과의 한가지 흥미로운 점을 주목해보세요: `ThreadPool` 은 종료 메시지들을 채널로 전송하고, worker 가 메시지를 수신하기 전에 worker 0 에 `join` 을 시도합니다. worker 0 이 아직 종료 메시지를 받지 못했기에 메인 스레드는 worker 0 이 종료될때까지 멈추게 됩니다. 그동안 각 worker 들은 종료 메시지를 수신합니다. worker 0 이 종료되면, 메인 스레드는 나머지 worker 들이 종료될 때 까지 대기합니다. 이 때 그들은 이미 종료 메시지를 받았으므로 종료될 수 있습니다.

축하드립니다! 드디어 우리 프로젝트를 완성했습니다; 우린 스레드 풀을 이용해 비동기적으로 응답하고, 종료 될때 풀의 모든 스레드를 정리하는 우아한 종료를 가진 기초적인 웹 서버를 만들었습니다.

다음은 참고용 전체 코드입니다.

파일명: src/bin/main.rs

```
extern crate hello;
use hello::ThreadPool;

use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::fs::File;
use std::thread;
```

```
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let mut file = File::open(filename).unwrap();
    let mut contents = String::new();

    file.read_to_string(&mut contents).unwrap();

    let response = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        status_line,
        contents.len(),
        contents
    );

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Filename: src/lib.rs

```
use std::thread;
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;

enum Message {
    NewJob(Job),
    Terminate,
}

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<FnBox + Send + 'static>;

impl ThreadPool {
    /// 새 스레드풀 생성
    ///
    /// size 는 풀 안의 스레드 개수입니다.
    ///
    /// # Panics
    ///
    /// `new` 함수는 size 가 0일때 패닉을 일으킵니다
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
```

```

        sender,
    }
}

pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static
{
    let job = Box::new(f);

    self.sender.send(Message::NewJob(job)).unwrap();
}
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");

        for _ in &mut self.workers {
            self.sender.send(Message::Terminate).unwrap();
        }

        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
        Worker {
        let thread = thread::spawn(move ||{
            loop {
                let message = receiver.lock().unwrap().recv().unwrap();

                match message {
                    Message::NewJob(job) => {
                        println!("Worker {} got a job; executing.", id);
                    }
                }
            }
        });
    }
}

```

```
        job.call_box();
    },
    Message::Terminate => {
        println!("Worker {} was told to terminate.", id);

        break;
    },
}
});

Worker {
    id,
    thread: Some(thread),
}
}
```

여기서 더 많은걸 할 수도 있습니다! 만약 여러분이 이 프로젝트를 개선하고 싶으시다면, 여기 몇가지 아이디어를 참고하세요:

- **ThreadPool** 과 public 메소드에 문서 더 추가하기.
- 라이브러리의 기능 테스트 추가하기.
- **unwrap** 호출을 에러 처리가 더 뛰어난 에러 핸들링 호출로 변경하기.
- **ThreadPool** 을 웹 요청을 처리하는 것 외에 다른 작업을 수행하는데 사용해보기.
- <https://crates.io/>에서 스레드 풀 크레이트를 찾아보고 그를 이용해 유사한 웹 서버를 구현해보고 그 것의 API랑 견고성을 우리가 구현한 스레드 풀과 비교해 보기.

마치며

수고하셨습니다! 여러분은 이 책을 끝마치셨습니다! 이 러스트의 여정에 참여해주셔서 감사드립니다. 여러분은 이제 자신의 러스트 프로젝트를 구현하고 다른 사람들의 프로젝트를 도와줄 준비가 되셨습니다. 여러분이 앞으로 러스트를 사용하시면서 겪으실 어려움을 해결하는데 도움이 되길 원하는 다른 러스트 유저들이 모여 있는 커뮤니티가 언제나 여러분을 환영한다는 걸 잊지 마세요.

부록

이번 장에는 러스트를 사용하는데 유용한 참고자료들이 포함되어 있습니다.

부록 A: 키워드

다음 목록은 러스트에서 현재 사용중이거나 미래에 사용될 키워드들입니다. 따라서 이들은 식별자, 함수명, 변수, 매개변수, 구조체 필드, 모듈, 크레이트, 상수, 매크로, 정적 변수, 속성, 타입, 트레이트, 라이프타임에 사용할 수 없습니다.

현재 사용되고 있는 키워드

다음 키워드들은 현재 각각의 설명에 해당하는 기능으로 사용되고 있습니다.

- **as** - 캐스팅하거나, 항목을 포함하는 특정 트레이트를 명확히 하거나, **use** 와 **extern crate** 구문에서 항목의 이름을 변경
- **break** - 반복문 즉각 탈출
- **const** - 상수 혹은 상수 로우 포인터 정의
- **continue** - 다음 반복 루프로 넘어감
- **crate** - 외부 크레이트를 링크하거나 해당 매크로가 정의되어 있는 크레이트를 대표하는 매크로 변수를 생성합니다.
- **else** - **if** 와 **if let** 제어 흐름 구조에 대한 대비책
- **enum** - 열거형 정의
- **extern** - 외부 크레이트, 함수 혹은 변수를 링크
- **false** - Boolean 의 거짓(false)을 나타내는 상수
- **fn** - 함수 혹은 함수 포인터 타입 정의
- **for** - 반복자의 항목들을 반복하거나, 트레이트를 구현하거나, 더 높은 수준의 라이프타임을 명시
- **if** - 조건식 결과를 이용한 분기
- **impl** - 내재된 특성 혹은 트레이트 특성 구현
- **in** - **for** 반복문 문법의 일부
- **let** - 변수 바인딩
- **loop** - 무조건적인 반복
- **match** - 패턴에 값을 매치
- **mod** - 모듈 정의
- **move** - 클로저가 사용하는 모든 값에 대해 소유권을 갖도록 만듬
- **mut** - 레퍼런스, 로우 포인터, 배턴 바인딩에 대한 가변성 표시
- **pub** - 구조체 필드, **impl** 블록, 모듈의 public 가시성 표시
- **ref** - 레퍼런스로 바인딩
- **return** - 함수의 반환
- **Self** - 트레이트를 구현하고 있는 타입의 별칭
- **self** - 메소드의 주체 혹은 현재 모듈

- **static** - 글로벌 변수 혹은 전체 프로그램 실행에서 지속되는 라이프타임
- **struct** - 구조체 선언
- **super** - 현재 모듈의 부모 모듈
- **trait** - 트레이트 선언
- **true** - Boolean 의 참(true)을 나타내는 상수
- **type** - 타입 별칭 혹은 관련 타입 선언
- **unsafe** - 코드, 함수, 트레이트, 구현이 안전하지 않다는 것을 표시
- **use** - 심볼을 범위 내로 불러옴
- **where** - 특정 타입으로 제한하는 절을 나타냄
- **while** - 표현식의 결과에 따라 반복

추후 이용하도록 예약된 키워드들

다음 키워드들은 아무 기능도 갖지 않지만, 리스트가 장래에 이용하도록 예약되어 있습니다.

- **abstract**
- **alignof**
- **become**
- **box**
- **do**
- **final**
- **macro**
- **offsetof**
- **override**
- **priv**
- **proc**
- **pure**
- **sizeof**
- **typeof**
- **unsized**
- **virtual**
- **yield**

부록 B: 연산자 및 기호

이번 부록은 러스트 문법 이외에도, 경로, 제네릭, 트레이트 바운드, 매크로, 속성, 주석, 퓨즈, 팔호 등에 사용되는 연산자 및 기호가 수록되어 있습니다.

연산자

Table B-1에 러스트 연산자를 나열해 놓았습니다. 각 연산자가 컨텍스트 상에 나타나는 모습과 간단한 설명, 연산자 오버로드 가능 여부 및 오버로드 가능할 경우 사용할 수 있는 트레이트 순서로 이루어져 있습니다.

Table B-1: 연산자

연산자	예시	설명	오버로드 가능 여부
!	<code>ident!(...)</code> , <code>ident!{...}</code> , <code>ident![...]</code>	매크로 전개	
!	<code>!expr</code>	비트 및 논리 보수	Not
<code>!=</code>	<code>var != expr</code>	불일치 비교	PartialEq
%	<code>expr % expr</code>	나머지 연산	Rem
<code>%=</code>	<code>var %= expr</code>	나머지 연산 후 대입	RemAssign
&	<code>&expr</code> , <code>&mut expr</code>	빌림	
&	<code>&type</code> , <code>&mut type</code> , <code>&'a type</code> , <code>&'a mut type</code>	빌림 포인터	
&	<code>expr & expr</code>	비트 단위 AND 연산	BitAnd
<code>&=</code>	<code>var &= expr</code>	비트 단위 AND 연산 후 대입	BitAndAssign
<code>&&</code>	<code>expr && expr</code>	논리 AND	
*	<code>expr * expr</code>	곱하기 연산	Mul
<code>*=</code>	<code>var *= expr</code>	곱셈 후 대입	MulAssign
*	<code>*expr</code>	역 참조	
*	<code>*const type</code> , <code>*mut type</code>	Raw 포인터	
+	<code>trait + trait</code> , <code>'a + trait</code>	타입 제약 조건 조합	
+	<code>expr + expr</code>	더하기 연산	Add
<code>+=</code>	<code>var += expr</code>	더하기 연산 후 대입	AddAssign

,	<code>expr, expr</code>	인수 및 요소 구분자	
-	<code>- expr</code>	부정 연산	<code>Neg</code>
-	<code>expr - expr</code>	빼기 연산	<code>Sub</code>
-=	<code>var -= expr</code>	빼기 연산 후 대입	<code>SubAssign</code>
->	<code>fn(...) -> type, ... -> type</code>	함수와 클로저 반환 타입	
.	<code>expr.ident</code>	멤버 접근	
..	<code>..., expr..., ..expr, expr..expr</code>	상한을 제외한 범위 리터럴	
..	<code>..expr</code>	구조체 갱신법	
..	<code>variant(x, ..), struct_type { x, .. }</code>	“나머지” 패턴 바인딩	
...	<code>expr...expr</code>	상한을 포함한 범위 패턴 (패턴 내)	
/	<code>expr / expr</code>	나누기 연산	<code>Div</code>
/=	<code>var /= expr</code>	나누기 연산 후 대입	<code>DivAssign</code>
:	<code>pat: type, ident: type</code>	제약 조건	
:	<code>ident: expr</code>	구조체 필드 초기화	
:	<code>'a: loop {...}</code>	loop 표식	
;	<code>expr;</code>	구문 및 요소 종결자	
;	<code>[...; len]</code>	고정 크기 배열 문법의 일부	
<<	<code>expr << expr</code>	좌측 쉬프트 연산	<code>Shl</code>
<<=	<code>var <<= expr</code>	좌측 쉬프트 연산 후 대입	<code>ShlAssign</code>
<	<code>expr < expr</code>	대소 비교 (소)	<code>PartialOrd</code>
<=	<code>expr <= expr</code>	동등 및 대소 비교 (소)	<code>PartialOrd</code>
=	<code>var = expr, ident = type</code>	대입/등가	
==	<code>expr == expr</code>	동등 비교	<code>PartialEq</code>
=>	<code>pat => expr</code>	갈래 문법의 일부	
>	<code>expr > expr</code>	대소 비교 (대)	<code>PartialOrd</code>
		동등 및 대소 비교	

<code>>=</code>	<code>expr >= expr</code>	(대)	<code>PartialOrd</code>
<code>>></code>	<code>expr >> expr</code>	우측 쉬프트 연산	<code>Shr</code>
<code>>>=</code>	<code>var >>= expr</code>	우측 쉬프트 연산 후 대입	<code>ShrAssign</code>
<code>@</code>	<code>ident @ pat</code>	패턴 바인딩	
<code>^</code>	<code>expr ^ expr</code>	비트 단위 XOR 연산	<code>BitXor</code>
<code>^=</code>	<code>var ^= expr</code>	비트 단위 XOR 연산 후 대입	<code>BitXorAssign</code>
<code> </code>	<code>pat pat</code>	다중 패턴	
<code> </code>	<code>expr expr</code>	비트 단위 OR 연산	<code>BitOr</code>
<code> =</code>	<code>var = expr</code>	비트 단위 OR 연산 후 대입	<code>BitOrAssign</code>
<code> </code>	<code>expr expr</code>	논리 OR 연산	
<code>?</code>	<code>expr?</code>	에러 전파	

연산자 이외의 기호

다음은 연산자와는 다르게 동작하는 (함수나 메소드를 호출했을 때 일어나는 현상과 유사하지 않다는 의미입니다) 문자 목록입니다.

Table B-2에 다양한 곳에서 사용할 수 있는 기호를 나열해 놓았습니다.

Table B-2: 독립 문법

기호	설명
<code>'ident</code>	라이프라임 지정 및 loop 표식
<code>...u8</code> , <code>...i32</code> , <code>...f64</code> , <code>...usize</code> , 기타 등등	특정 타입 정수 리터럴
<code>"..."</code>	스트링 리터럴
<code>r"..."</code> , <code>r#"..."#</code> , <code>r##"..."##</code> , 기타 등등	Raw 스트링 리터럴, 이스케이프 문자를 처리하지 않음
<code>b"..."</code>	바이트 스트링 리터럴; 문자열 대신 <code>[u8]</code> 이용
<code>br"..."</code> , <code>br#"..."#</code> , <code>br##"..."##</code> , 기타 등등	Raw 바이트 스트링 리터럴, Raw 스트링과 바이트 스트링을 합친 것

'...'	문자 리터럴
b'...'	ASCII 바이트 리터럴
... expr	클로저
!	함수 분기를 위해 존재하는 의미를 갖지 않는 빈 타입
_	"무시된" 패턴 바인딩; 정수 링터럴의 가독성을 높이 는 데에도 사용됨

Table B-3 은 모듈 계층 구조의 경로를 나타내는 데 사용되는 기호를 나타냅니다.

Table B-3: 경로 관련 문법

기호	설명
ident::ident	네임스페이스 경로
::path	크레이트의 루트 디렉토리를 기준으로 한 상대 경로 (즉, 명시적인 절대 경로)
self::path	현재 모듈을 기준으로 한 상대 경로 (즉, 명시적인 상대 경로).
super::path	현재 모듈의 부모 모듈을 기준으로 한 상대 경로
type::ident, <type as trait>::ident	연관 상수, 함수 및 유형
<type>::...	직접 명명할 수 없는 타입에 연관된 항목 (예시 <&T>::..., <[T]>::..., 기타 등등)
trait::method(...)	해당 메소드를 정의한 트레이트 이름으로 메소드 호출을 명확화
type::method(...)	정의된 타입명을 이용해 메소드 호출을 명확화
<type as trait>::method(...)	타입과 트레이트 이름을 이용해 메소드 호출을 명확화

Table B-4 는 제네릭 타입 매개변수로 사용되는 기호를 나타냅니다.

Table B-4: 제네릭

기호	설명
path<...>	타입의 제네릭 매개변수 명시 (예시 Vec<u8>)
path::<...>, method::<...>	제네릭 타입, 함수, 메소드 등의 표현식에 매개변수 명시; turbofish 로도 불림 (예시 "42".parse::<i32>())
fn ident<...> ...	제네릭 함수 정의

<code>struct ident<...> ...</code>	제네릭 구조체 정의
<code>enum ident<...> ...</code>	제네릭 열거체 정의
<code>impl<...> ...</code>	제네릭 구현 정의
<code>for<...> type</code>	고 랭크 라이프타임 제약
<code>type<ident=type></code>	하나 이상의 관련 타입에 특정 할당을 갖는 제네릭 타입 (예시 <code>Iterator<Item=T></code>)

Table B-5 은 트레이트를 이용해 제네릭 매개변수의 제약 조건을 설정하는 데 사용되는 기호를 나타냅니다.

Table B-5: 트레이트 제약 조건

기호	설명
<code>T: U</code>	제네릭 매개변수 <code>T</code> 는 <code>U</code> 를 구현한 타입일 것
<code>T: 'a</code>	제네릭 타입 <code>T</code> 는 <code>a</code> 보다 긴 라이프타임을 가질 것 (해당 타입은 일시적으로 <code>'a</code> 보다 짧은 라이프타임을 갖는 레퍼런스를 포함할 수 없다는 의미입니다)
<code>T: 'static</code>	제네릭 타입 <code>T</code> 는 <code>'static</code> 이외의 빌림 참조자를 포함하지 않을 것
<code>'b: 'a</code>	제네릭 라이프타임 <code>'b</code> 는 <code>'a</code> 보다 긴 라이프타임을 가질 것
<code>T: ?Sized</code>	제네릭 매개변수가 동적 사이즈 타입이 되는 것을 허용
<code>'a + trait,</code> <code>trait +</code> <code>trait</code>	타입 제약 조건 조합

Table B-6 은 매크로를 호출 및 정의하거나 임의의 아이템에 대한 속성을 명시하는 데 사용되는 기호를 나타냅니다.

Table B-6: 매크로 및 속성

기호	설명
<code># [meta]</code>	외부 속성
<code>#! [meta]</code>	내부 속성
<code>\$ident</code>	매크로 치환
<code>\$ident:kind</code>	매크로 캡처
<code>\$(...)...</code>	매크로 반복

Table B-7 은 주석 기호를 나타냅니다. (소속 및 외부 항목이 무엇을 의미하는지는 14-2 에 나와있습니다)

Table B-7: 주석

기호	설명
//	한 줄 주석
//!	소속 항목 대상 한 줄 문서화 주석
///	외부 항목 대상 한 줄 문서화 주석
/*...*/	블록 주석
/*!...*/	소속 항목 대상 블록 문서화 주석
/**...*/	외부 항목 대상 블록 문서화 주석

Table B-8 은 튜플 문법에서 사용되는 기호를 나타냅니다.

Table B-8: 튜플

기호	설명
()	빈 튜플 (일명 '유닛'), 리터럴이자 타입임
(expr)	괄호 내 표현식
(expr,)	단일 개체 튜플 표현식
(type,)	단일 개체 튜플 타입
(expr, ...)	튜플 표현식
(type, ...)	튜플 타입
expr(expr, ...)	함수 호출 표현식; 튜플 <code>struct</code> 와 튜플 <code>enum</code> variants 를 초기화하는 데도 사용
ident!(...), ident!{...}, ident![...]	매크로 호출
expr.0, expr.1, 기타 등등	튜플 인덱싱

Table B-9 은 중괄호의 사용처를 나타냅니다.

Table B-9: 중괄호

사용처	설명
{...}	블록 표현식
Type {...}	<code>struct</code> 리터럴

Table B-10 은 대괄호의 사용처를 나타냅니다.

Table B-10: 대괄호

사용처	설명
[...]	배열 리터럴
[expr; len]	expr 을 len 만큼 복제한 배열 리터럴
[type; len]	type 인스턴스를 len 만큼 갖는 배열 타입
expr[expr]	배열 인덱싱. Index, IndexMut 을 이용해 오버로드 가능
expr[..], expr[a..], expr[..b], expr[a..b]	컬렉션 슬라이싱 모양의 컬렉션 인덱싱. 인덱스로 Range, RangeFrom, RangeTo, RangeFull 을 사용

부록 C: derive 가능한 트레잇

책의 여러 곳에서 구조체나 열거형 정의 시 적용할 수 있는 `derive` 속성을 다뤘습니다. `derive` 속성은 여러분이 `derive` 문법을 명시함으로써 생성할 수 있는 기본 트레잇 구현체를 생성해줍니다.

이 부록에선 표준 라이브러리에 존재하는 트레잇 중 `derive`로 이용 가능한 트레잇들의 레퍼런스를 제공합니다. 각 절에서 다루는 내용은 다음과 같습니다:

- 어떤 연산자와 메소드가 해당 트레잇에 derive 가능한지
- `derive`로 제공된 트레잇의 구현체가 하는 일
- 타입에 트레잇을 구현한다는 것의 의미
- 트레잇을 구현하도록 허용되거나 허용되지 않는 조건
- 트레잇이 필수적인 연산들의 예시

`derive` 속성을 통해 제공되는 것과 다른 동작을 원하신다면, 표준 라이브러리 문서에서 각 트레잇을 직접 구현하는 법을 찾아보시기 바랍니다.

표준 라이브러리의 나머지 트레잇들은 `derive`를 통해 여러분의 타입에 구현 될 수 없습니다. 이 트레잇들은 적절한 기본 동작을 갖지 않기 때문에, 여러분이 수행하려는 작업에 맞춰서 직접 구현해야 합니다.

`derive` 될 수 없는 트레잇의 대표적인 예는 `Display` 트레잇입니다. 이 트레잇의 역할은 최종 사용자(end user)들을 위한 포맷팅입니다. 다만 적절한 포맷팅을 만들기 위해선 어느 부분을 보여줘야 할지, 관련성 있는 부분은 어느 곳인지, 데이터의 형식은 어떤 것이 가장 적절할지 등을 여러분이 직접 끊임없이 고민해야 합니다. 러스트 컴파일러는 이런 식으로 적절한 포맷팅을 생성해낼 수 없고, 따라서 `derive`를 지원하지 않습니다.

이 부록에 나온 트레잇들이 `derive` 가능한 트레잇의 전부는 아닙니다: 라이브러리에서 자신들의 트레잇에 `derive`를 구현할 수도 있기 때문에, 여러분이 `derive`를 사용할 수 있는 트레잇은 사실상 무제한이라고 보셔도 됩니다. `derive`를 구현하는 법에 관해선 절차 매크로 사용을 포함해 부록 D에서 다루고 있습니다.

프로그래머 출력을 위한 `Debug`

`Debug` 트레잇을 사용하면 형식 문자열에서 디버그 포맷팅을 사용할 수 있습니다. 디버그 포맷팅은 형식 문자열의 `{}` 변경자 내에 `:?`를 추가해서 사용합니다.

`Debug` 트레잇을 사용하면 해당 타입의 인스턴스를 디버깅 목적으로서 출력 가능합니다. 이는 여러분들의 타입을 사용하는 다른 프로그래머들이 프로그램의 실행 도중 인스턴스를 점검할 수 있게 해줍니다.

Debug 트레이트이 필수적인 경우의 예는, `assert_eq` 매크로를 사용할 때입니다. 이 매크로는 동치 비교 결과가 거짓일 경우, 프로그래머가 두 인스턴스가 같지 않다는 것을 확인할 수 있도록 인수로 넘겨받은 인스턴스의 값을 출력하기 때문입니다.

동치 비교를 위한 **PartialEq** 와 **Eq**

PartialEq 트레이트을 사용하면 타입의 인스턴스를 동치 비교할 수 있고 `==` 와 `!=` 연산자를 사용할 수 있습니다.

PartialEq derive 는 `eq` 메소드를 구현합니다. **PartialEq** 가 구조체에 derive 된다면, 인스턴스를 비교할 때 각 인스턴스의 모든 필드가 서로 동일한 경우에만 두 인스턴스가 동일하다고 판별하며, 만약 서로 다른 필드가 하나라도 있다면 동일하지 않다고 판별합니다. 열거형에 derive 될 경우, 각각의 variant 는 자신과 동일하며 그 외의 variant 와는 동일하지 않습니다.

PartialEq 트레이트이 필수적인 경우는 `assert_eq!` 매크로를 사용할 때입니다. 타입의 두 인스턴스가 서로 동일한지 비교할 수 있어야 하기 때문입니다.

Eq 트레이트은 메소드를 갖지 않습니다. 그저 어노테이션 된 타입의 모든 값에 대해 값이 그 자체와 동일하다는 것을 알리는 것이 목적이기 때문입니다. **Eq** 트레이트은 **PartialEq** 를 구현한 타입에만 적용 가능합니다. 하지만 그렇다고 해서 **PartialEq** 를 구현한 모든 타입이 **Eq** 를 구현할 수 있는 것은 아닌데, 대표적인 예로 부동 소수점 타입이 있습니다: 부동 소수점 숫자의 구현체에 따르면, 두 비수(`NaN`, not-a-number) 의 인스턴스는 서로 같지 않습니다.

Eq 가 필수적인 예는 `HashMap<K, V>` 의 키값으로 사용될 경우입니다. `HashMap<K, V>` 에서 두 키값이 서로 같은지 확인해야 하기 때문입니다.

순서 비교를 위한 **PartialOrd** 와 **Ord**

PartialOrd 트레이트은 정렬 목적으로 타입의 인스턴스를 비교할 수 있게 해줍니다. 이를 구현한 타입은 `<`, `>`, `<=`, `>=` 등의 연산자를 사용할 수 있습니다. **PartialOrd** 트레이트은 **PartialEq** 트레이트을 구현한 타입에만 적용할 수 있습니다.

PartialOrd derive 는 `partial_cmp` 메소드를 구현해야 합니다. 이 메소드는 `Option<Ordering>` 을 반환하며, 주어진 값으로 순서를 비교할 수 없을 때 반환값은 `None` 이 됩니다. 해당 타입의 대부분의 값은 비교가 가능하지만, 순서를 비교할 수 없는 값의 예는 비수(`NaN`, not-a-number) 부동 소수점 값입니다. 아무 부동 소수점 값과 `NaN` 부동 소수점 값으로 `partial_cmp` 를 호출하면 `None` 이 리턴되는 걸 보실 수 있습니다.

구조체에 derive 될 경우 **PartialOrd** 는 두 인스턴스의 각 필드를 구조체 정의에 나타난 순서대로 비교합

니다. 열거형에 derive 될 경우는 해당 열거형 정의문에 먼저 선언한 variant 가 나중에 선언한 variant 보다 더 적게(less) 평가됩니다.

PartialOrd 트레이트이 필수적인 예는, 특정 범위 내에서 랜덤한 값을 생성해내는 **rand** 크레이트의 **gen_range** 메소드가 있습니다.

Ord 트레이트은 명시된 해당 타입에 있어서, 이 타입의 어떠한 두 값간에 순서를 비교하는 것이 가능하다는 것을 나타냅니다. **Ord** 트레이트은 **cmp** 메소드를 구현하고, 이 메소드는 **Ordering** 을 반환합니다. 어째서 **Option<Ordering>** 이 아닌 **Ordering** 을 반환하는가 함은, 언제나 순서 비교가 가능하다는 것을 보장하기 위해서입니다. **Ord** 트레이트은 **PartialOrd** 와 **Eq** (그리고 **Eq** 는 **PartialEq** 가 필수적이죠) 를 구현한 타입에만 적용 가능합니다. 구조체나 열거형에 derive 될 시에는 **cmp** 가 **PartialOrd** 의 **partial_cmp** 가 derive 되어 구현된 것과 똑같이 작동할 것입니다.

Ord 가 필수적인 예는, 값을 정렬해서 저장하는 자료구조인 **BTreeSet<T>** 에 값을 저장할 때입니다.

값 복제를 위한 **Clone** 와 **Copy**

Clone 트레이트은 명시적으로 값의 깊은 복사를 생성할 수 있게 해주며, 복제 과정은 임의의 코드 실행과 힙 데이터 복사가 포함될 수 있습니다. **Clone** 에 대한 자세한 내용을 원하시는 분은 4-1 장의 "변수와 데이터가 상호작용하는 방법: 클론" 절을 참고하시기 바랍니다.

Clone derive 는 **clone** 메소드를 구현합니다. 유의할 점은 타입의 모든 부분에 **clone** 메소드가 호출되기 때문에, 해당 타입의 모든 필드 혹은 값이 **Clone** 을 derive 하거나 구현해야 한다는 것입니다.

Clone 이 필수적인 예는 슬라이스에 **to_vec** 메소드를 호출할 경우입니다. 슬라이스는 자신이 포함하는 타입 인스턴스를 소유하지 않기 때문에, **to_vec** 메소드는 슬라이스의 각 항목에 **clone** 을 호출하여 자신이 반환할 벡터가 인스턴스들을 소유할 수 있도록 합니다.

Copy 트레이트은 값을 복제할 때 스택에 저장된 비트들을 복사할 과정만을 거칩니다; 어떠한 임의의 코드도 실행할 필요가 없습니다. **Copy** 에 대한 내용을 더 원하시는 분은 4-1장의 "스택에만 있는 데이터: 복사" 를 참고하시기 바랍니다.

Copy 트레이트은 어떠한 메소드도 정의하지 않음으로써 프로그래머가 메소드를 오버로딩해 임의의 코드를 실행시키는 경우를 방지합니다. 따라서 모든 프로그래머들은 값의 복사가 느려질 것을 염려하지 않아도 됩니다.

Copy derive 는 타입의 모든 부분이 **Copy** 를 구현한 타입에만 가능합니다. **Copy** 트레이트 적용은 **Clone** 을 구현하고 있는 타입에만 적용 가능합니다. 이는 **Copy** 를 구현하는 타입은 **Copy** 와 같은 작업을 하는 **Clone** 의 간단한 구현을 지니기 때문입니다.

`Copy` 트레이트를 요구하는 경우는 매우 드뭅니다; `Copy` 를 구현한 타입은 최적화가 가능한데, 즉 여러분이 `clone` 을 호출하지 않아도 된다는 의미이며, 이는 코드를 더 간결하게 만들어 줍니다.

`Copy` 로 할 수 있는 것은 `Clone` 으로도 할 수 있습니다만, 이 경우 코드가 좀 느려지거나 코드에서 `clone` 을 사용해야 할 수도 있습니다.

값을 고정된 크기의 값으로 맵핑하기 위한 Hash

`Hash` 트레이트는 해쉬 함수를 이용해 임의 크기 타입의 인스턴스를 고정된 크기의 값으로 맵핑할 수 있게 해줍니다. `Hash` derive 는 `hash` 메소드를 구현합니다. `hash` 메소드의 구현 상 타입의 각 부분에 `hash` 를 호출하도록 되어 있으므로, `Hash` 를 derive 하기 위해선 모든 필드나 값은 `Hash` 를 구현해야 합니다.

`Hash` 가 필수적인 예는 `HashMap<K, V>` 에 효율적으로 데이터를 저장하기 위해 key 값을 저장하는 경우입니다.

기본 값을 위한 Default

`Default` 트레이트는 타입의 기본 값을 생성할 수 있게 해줍니다. `Default` derive 는 `default` 함수를 구현합니다. `default` 메소드의 구현 상 타입의 각 부분에 `default` 를 호출하도록 되어 있으므로, `Default` 를 derive 하기 위해선 모든 필드나 값은 `Default` 를 구현해야 합니다.

`Default::default` 함수는 5-1 장의 "구조체 간신법을 이용하여 기존 구조체 인스턴스로 새 구조체 인스턴스 생성하기" 에서 다룬 구조체 간신법과 연계하여 사용하는 것이 일반적입니다. 여러분은 구조체의 일부 필드를 원하는대로 설정하고 나머지 필드는 `..Default::default()` 를 이용해 기본 값으로 설정할 수 있습니다.

`Default` 트레이트는 `Option<T>` 인스턴스에 `unwrap_or_default` 메소드를 사용할 때 필수적입니다. 예를 들어, `Option<T>` 가 `None` 일 경우 `unwrap_or_default` 메소드는 `Option<T>` 에 해당하는 `T` 형식의 `Default::default` 호출 결과를 반환합니다.

부록 D: 매크로

우린 이 책에서 `println!` 등의 매크로를 사용했습니다. 하지만 아직 매크로가 정확히 무엇이고, 어떻게 동작하는지는 알아보지 않았습니다. 이번 부록에선 매크로에 대해 다음과 같은 순서로 알아볼 것입니다:

- 매크로가 무엇이고, 함수와 다른 점
- 메타프로그래밍을 하기 위한 '선언적 매크로' 정의법
- `derive` 트레잇을 커스텀하기 위한 절차적 매크로 정의법

매크로에 대한 자세한 내용을 부록에서 다루는 이유는, 러스트의 매크로는 아직 진화중이기 때문입니다. 러스트 1.0 이래로, 매크로는 언어의 나머지 부분과 표준 라이브러리보다 빠르게 바뀌었고, 향후에도 그럴 것입니다. 따라서 이 절은 책의 다른 부분보다 시대에 뒤처질 가능성이 높습니다. 러스트는 안정성을 보증하므로 여기 나오는 코드는 이후 버전에서도 동작할 테지만, 그때쯤엔 이 책이 발행된 시점에선 사용할 수 없었던 추가 기능이나, 보다 쉽게 매크로를 작성하는 여러 방법이 존재할 것입니다. 만약 이 부록을 참고해 무언가 구현하려 하신다면 이 점을 염두해두시기 바랍니다.

매크로와 함수의 차이

근본적으로, 매크로는 다른 코드를 작성하는 코드입니다. 이 개념은 *메타프로그래밍*(metaprogramming)으로 잘 알려져 있죠. 우린 부록 C에서 트레잇에 다양한 구현체를 생성해주는 `derive` 속성에 대해 다뤘고, 책 중간중간 `println!` 과 `vec!` 매크로도 사용했습니다. 이 모든 매크로들은 수동으로 코드를 작성하지 않고도 많은 코드를 생산해낼 수 있게 합니다.

메타프로그래밍은 여러분이 작성하고 관리해야 할 코드의 양을 줄여줍니다. 물론 이는 함수의 역할이기도 합니다만, 매크로는 함수가 하지 못하는 일도 할 수 있습니다.

함수 시그니처는 해당 함수가 갖는 매개변수의 개수와 타입을 선언해야만 하는 반면, 매크로는 매개변수의 개수를 가변적으로 처리할 수 있습니다: 간단한 예로, `println!("hello")` 와 같이 1 개의 매개변수를 사용하거나, `println!("hello {}", name)` 처럼 2 개의 매개변수를 사용할 수 있습니다. 또한 매크로는 컴파일러가 코드의 의미를 해석하기 이전에 작동합니다. 따라서 주어진 타입에 트레잇을 구현하는 등, 런타임에 호출되는 함수로는 불가능한 일을 할 수 있습니다.

단, 함수 대신 매크로를 구현하는 것도 단점이 존재합니다. 매크로를 구현한다는 것은 러스트 코드를 만들어내는 코드를 작성한다는 것입니다. 이는 추상화 계층을 하나 더 만들어 낸다는 것이기 때문에, 매크로 정의는 일반적으로 함수 정의에 비해 읽고, 이해하고, 관리하기 어렵습니다. 요약해서, 매크로 정의의 단점은 함수 정의보다 복잡하다는 겁니다.

함수와 매크로의 또 다른 차이는, 매크로 정의는 함수 정의와는 달리 모듈의 네임스페이스에 소속되지 않는다는 것입니다. 이로 인한 외부 크레이트 사용 시 발생하는 예기치 않은 이름 충돌을 막기 위해선, 외부 크레이트를 스코프 내로 가져오는 동시에 `#[macro_use]` 어노테이션을 사용하여 가져올 매크로를 명시해야 합니다.

다. 다음은 `serde` 크레이트에 정의된 매크로를 현재 크레이트의 스코프로 가져오는 예제입니다:

```
#[macro_use]
extern crate serde;
```

만약 어노테이션을 명시하지 않더라도 `extern crate` 만으로 스코프 내에 매크로가 자동적으로 들어오게 됐더라면, 여러분은 같은 이름의 매크로가 정의된 크레이트를 동시에 사용하지 못했을 겁니다. 충돌이 실제로 자주 발생하는 건 아니지만, 많은 크레이트를 사용할수록 충돌이 발생할 확률은 높아집니다.

마지막으로, 매크로와 함수의 차이 중 중요한 한 가지가 남았습니다: 함수는 정의 위치에 관계 없이 아무 곳에서나 호출이 가능하지만, 매크로는 호출 하기 전에 반드시 해당 파일에 정의하거나 가져와야 합니다.

일반적인 메타프로그래밍을 위한 `macro_rules!` 를 사용하는 선언적 매크로

러스트에서 매크로는 선언적 매크로(*declarative macros*)의 형태로 가장 널리 사용됩니다. 이는 예제 별 매크로(*macros by example*), `macro_rules!` 매크로, 아니면 그냥 매크로라고 불리기도 합니다. 선언적 매크로란 것은, 러스트의 `match` 표현식과 유사하게 작성할 수 있습니다. `match` 는 표현식을 다루는 제어 구조입니다. 6 장에서 다뤘듯 `match` 는 패턴과 표현식의 결과값을 비교하고, 해당 패턴과 연관된 코드를 실행합니다. 매크로 또한 값과, 관련된 코드를 갖는 패턴을 비교합니다; 이때 값은 매크로에 넘겨진 러스트 소스 코드를 말하고, 패턴에는 소스코드의 구조가 해당되며, 각 패턴에 관련된 코드는 매크로로 전달되어 대체된 코드를 말합니다. 그리고, 이 모든 과정은 컴파일 중 일어납니다.

`vec!` 매크로가 어떻게 정의되어 있는지 살펴보도록 합시다. `vec!` 매크로는 특정 값을 이용해 새로운 벡터를 생성하는 매크로로, 8 장에서 다뤘습니다. 다음은 이 매크로를 사용해 세 정수 값으로 새로운 벡터를 생성하는 예시입니다:

```
let v: Vec<u32> = vec![1, 2, 3];
```

이외에도 `vec!` 매크로는 두 정수로 이루어진 벡터나 5 개의 스트링 슬라이스로 이루어진 벡터를 만드는데도 사용할 수 있습니다. 함수는 사전에 값의 개수나 타입을 알 수 없으므로 이러한 작업은 불가능합니다.

`vec!` 매크로의 정의를 간략화한 모습을 Listing D-1에서 한번 살펴봅시다.

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Listing D-1: `vec!` 매크로 정의를 간략화한 모습

Note: 표준 라이브러리 내 `vec!` 매크로의 실제 정의에는 메모리의 정확한 양을 미리 할당하는 코드가 포함되어 있습니다. 이 코드에선 예제를 간략화 하기 위해서 해당 부분을 제외했습니다.

`#[macro_export]` 어노테이션은 우리가 정의한 매크로가 들어 있는 크레이트를 누군가 임포트 했을 때, 해당 매크로를 사용할 수 있도록 해줍니다. 이 어노테이션을 사용하지 않을 경우, 이 크레이트를 디펜던시로 갖는 누군가가 `#[macro_use]` 어노테이션을 사용하더라도 해당 매크로는 스코프 내로 가져와지지 않습니다.

매크로 정의는 `macro_rules!` 와 느낌표가 붙지 않는 매크로의 이름으로 시작됩니다. 예시의 경우, 매크로명은 `vec`이며 뒤따르는 중괄호가 매크로 정의의 본문을 나타냅니다.

`vec!`의 본문 구조는 `match` 표현식 구조와 유사합니다. 여기선 `($($x:expr),*)` 패턴과 그 뒤로 따라오는 `=>`, 그리고 해당 패턴에 연관된 코드 블록으로 이루어진 갈래 하나를 갖습니다. 패턴이 매치될 경우, 해당 패턴에 연관된 코드가 반환 됩니다. 이 매크로는 하나뿐인 패턴을 갖기 때문에 매치되는 경우는 하나뿐이며, 다른 모든 경우는 에러가 발생할 것입니다. 물론 이보다 복잡한 매크로는 갈래를 하나 이상 갖겠죠.

매크로 정의에서 사용하는 패턴 문법은 18 장에서 다룬 패턴 문법과는 다릅니다. 그 이유는, 매크로는 러스트 코드 구조와 매치되기 위한 것인데 18 장의 패턴에서 사용하는 값과 러스트 코드 구조는 상당히 다르기 때문입니다. 이제 Listing D-1의 패턴을 하나씩 살펴보면서 알아보도록 합시다; 매크로 패턴의 모든 문법에 대한 내용은 [레퍼런스](#)를 참조하시기 바랍니다.

먼저 괄호 쌍이 전체 패턴을 둘러쌉니다. 그 다음 달려 기호(`$`)뒤에 괄호 쌍이 오며, 배치할 코드에서 사용하기 위해, 괄호 안 패턴과 일치하는 값을 캡처합니다. `$()` 내에는 `$x:expr` 가 있는데, 이는 어떤 러스트 표현식과도 매치되며, 그에 `$x`라는 이름을 부여하는 기능을 합니다.

`$()`에 따라오는 쉼표는 `$()` 내에 캡처되어 매치된 코드 뒤에 나타날 수도 있는 리터럴 쉼표 구분 문자를

나타냅니다. 쉼표 뒤 `*` 는 자신 앞에 위치한 0 개 이상 패턴을 지정합니다.

이 매크로를 `vec![1, 2, 3];` 으로 호출할 경우, `$x` 패턴은 `1, 2, 3` 세 표현식에 맞춰 세 번 매치됩니다.

이제 패턴 갈래와 연관된 본문 코드를 살펴봅시다: `$(())*` 부분 내의 `temp_vec.push()` 코드는 패턴에서 `$(())` 에 매치되는 횟수만큼 반복되어 생성되고, 코드 내 `$x` 는 각각의 매치된 표현식으로 대체됩니다. 따라서 `vec![1, 2, 3]` 으로 이 매크로를 호출할 경우, 매크로로부터 생성되어 매크로 호출문을 대체할 코드는 다음과 같습니다:

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

우린 인수 개수가 어느만큼이건, 어떤 타입이건 가리지 않고 특정한 요소들을 포함할 벡터를 만들어내는 코드를 생성할 수 있는 매크로를 선언했습니다.

대부분의 러스트 프로그래머는 매크로를 작성하기 보다는 사용하는 일이 더 많을 겁니다. 따라서 여기선 `macro_rules!` 에 대해서 더 이상 다루지 않습니다. 매크로를 작성하는 법에 대해 더 많은 것을 배우고 싶으신 분은 “[The Little Book of Rust Macros](#)” 등의 온라인 문서를 찾아보세요.

커스텀 `derive` 를 위한 절차적 매크로

두번째 매크로 형식은 함수(프로시저(procedure) 유형)에 가깝기 때문에 절차적 매크로(*procedural macro*)라고 불립니다. 선언적 매크로는 코드를 패턴과 매치시키고 다른 코드로 대체하는 반면, 절차적 매크로는 어떤 러스트 코드를 입력 받고, 코드를 연산하여 러스트 코드를 생성합니다. 이 내용이 작성된 시점엔 `derive` 어노테이션에 특정 트레이트 이름을 지정하여, 타입에 해당 트레이트를 구현하도록 하는 데에만 절차적 매크로를 정의할 수 있습니다.

우린 `hello_macro` 크레이트를 생성할 것이며, 이 크레이트는 `hello_macro` 라는 연관 함수 하나를 가진 `HelloMacro` 트레이트를 정의할 것입니다. 다만, 이 크레이트를 사용하는 사람이 각 타입마다 `HelloMacro` 트레이트를 구현할 필요 없도록 절차적 매크로를 제공하여, 사용자가 자신의 타입에 `#[derive(HelloMacro)]` 를 어노테이트 하는 것 만으로도 `hello_macro` 함수의 기본 구현체를 이용할 수 있도록 해봅시다. 이 기본 구현체는 `Hello, Macro! My name is TypeName` (`TypeName` 엔 이 트레이트가 정의된 타입의 이름이 들어갈 겁니다.) 을 출력할 것입니다. 쉽게 말해서, 우리 크레이트를 이용하는 다른 프로그래머가 Listing D-2 처럼 코드를 작성할 수 있도록 할 것입니다.

Filename: src/main.rs

```
extern crate hello_macro;
#[macro_use]
extern crate hello_macro_derive;

use hello_macro::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Listing D-2: 우리가 만든 크레이트의 사용자가 우리 절차적 매크로를 이용해 작성 가능할 코드

완성됐을 때, 이 코드는 `Hello, Macro! My name is Pancakes!` 를 출력할 겁니다. 이제 새 라이브러리 크레이트를 만드는 첫 과정을 진행해보죠:

```
$ cargo new hello_macro --lib
```

다음으로, `HelloMacro` 트레이트와 연관 함수를 정의합시다.

Filename: src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

이제 트레이트와 함수를 만들었습니다. 또한 현재 시점에서도 이 크레이트를 이용해 다음과 같은 방식으로 우리가 원하던 기능을 구현할 수는 있습니다.

```

extern crate hello_macro;

use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}

```

하지만 지금으로서는 `hello_macro` 와 같이 사용하려는 타입마다 구현 내용을 직접 작성해줘야 합니다; 이 작업은 생략할 수 있도록 하는 편이 좋을 것입니다.

허나, 우린 아직 `hello_macro` 함수의 기본 구현체를 제공할 수 없습니다. 우리가 원하는 기능은 자신이 구현된 트레이트 이름을 알아내어 출력하는 것인데, 러스트엔 리플렉션 기능이 없어서 런타임 중에는 타입명을 알아낼 수 없기 때문입니다. 따라서 매크로를 이용해 컴파일 타임에 코드를 생성해야 합니다.

다음 단계는 절차적 매크로를 정의하는 것입니다. 이 내용이 작성된 시점엔 절차적 매크로가 자신의 크레이트 내부에 위치해야만 하지만, 이 제약은 언젠가 사라질 겁니다. 크레이트 및 매크로 크레이트의 구조화 규칙은 다음과 같습니다: 예를 들어 `foo` 크레이트의 경우, derive 절차적 매크로 크레이트명은 `foo_derive` 가 됩니다. 이제 `hello_macro` 프로젝트 내에 `hello_macro_derive` 라는 새 크레이트를 만들어 봅시다:

```
$ cargo new hello_macro_derive --lib
```

이 두 크레이트는 밀접히 연관되어 있고, 따라서 절차적 매크로 크레이트를 `hello_macro` 크레이트 디렉토리 내에 생성하였습니다. 이는 우리가 만약 `hello_macro` 내 트레이트 정의를 변경할 경우,

`hello_macro_derive` 의 절차적 매크로 구현 또한 변경하도록 강제합니다. 이 두 크레이트는 각각 따로 배포될 것이고, 이를 사용할 프로그래머는 이 크레이트들을 각각 디펜던시에 추가하고 스코프 내로 가져와야 할 겁니다. 물론 우린 `hello_macro` 크레이트에 `hello_macro_derive` 를 디펜던시로 추가하고, 절차적 매크로 코드를 다시 `export` 할 필요 없도록 만들 수도 있습니다. 하지만 이 방법대로는 `derive` 기능을 원치 않던 사용자들도 강제적으로 `hello_macro` 를 사용해야만 합니다.

우린 `hello_macro_derive` 크레이트가 절차적 매크로 크레이트라는 것을 나타내야 합니다. 또한 잠시 후에 볼 수 있듯이 `syn` 크레이트와 `quote` 크레이트의 기능이 필요하므로 이 둘을 디펜던시로 추가합니다. 결과적으로 `hello_macro_derive` 의 `Cargo.toml` 파일은 다음과 같습니다:

Filename: `hello_macro_derive/Cargo.toml`

```
[lib]
proc-macro = true

[dependencies]
syn = "0.11.11"
quote = "0.3.15"
```

이제 절차적 매크로를 정의해봅시다. 먼저, 여러분의 `hello_macro_crate` 크레이트 `src/lib.rs` 파일에 Listing D-3 코드를 작성합니다. 다만, 이 코드는 우리가 `impl_hello_macro` 함수를 구현하지 않는 이상 컴파일 되진 않을 겁니다.

Filename: `hello_macro_derive/src/lib.rs`

```
extern crate proc_macro;
extern crate syn;
#[macro_use]
extern crate quote;

use proc_macro::TokenStream;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a string representation of the type definition
    let s = input.to_string();

    // Parse the string representation
    let ast = syn::parse_derive_input(&s).unwrap();

    // Build the impl
    let gen = impl_hello_macro(&ast);

    // Return the generated impl
    gen.parse().unwrap()
}
```

Listing D-3: 대부분의 절차적 매크로 크레이트가 러스트 코드를 처리하는데 사용할 코드

D-3에서 함수들이 분리된것을 주목하세요; 이는 절차적 매크로를 더 편리하게 작성하기 위한 방법이므로, 여러분이 보게 될, 혹은 만들게 될 많은 크레이트도 거의 대부분 마찬가지일 겁니다. 이 때, 호출한 `impl_hello_macro` 함수에서 어떤 작업을 할 지는 여러분이 어떤 목적으로 절차적 매크로를 만드는 지에 따라 달라질 겁니다.

우린 `proc_macro`, `syn`, `quote` 3 개의 새로운 크레이트를 사용했습니다. 이 중 `proc_macro` 는 러스트에 포함되어 있으므로 `Cargo.toml` 에 디펜던시로 추가할 필요가 없으며, 러스트 코드를 문자열로 변환하는 기능을 갖습니다. 또한 `syn` 크레이트는 문자열로 변환한 러스트 코드를 연산을 수행하기 위한 자료구조

로 파싱합니다. 마지막으로 `quote` 크레이트는 `syn` 의 자료구조를 러스트 코드로 복원하는 역할을 합니다. 이 크레이트들은 어떤 종류의 러스트 코드든 우리가 다루고 싶은 것을 더 쉽게 다룰 수 있도록 도와줍니다: 모든 러스트 코드를 파싱하는 파서를 작성하는 건 결코 쉬운 일은 아닙니다.

`hello_macro_derive` 함수에 `proc_macro_derive` 와 `HelloMacro` 라는 이름을 어노테이트 하였기 때문에, `hello_macro_derive` 함수는 우리 라이브러리 사용자가 타입에 `#[derive(HelloMacro)]` 를 명시했을 때 호출됩니다. `HelloMacro` 는 우리 트레이트 이름과 매치되는데, 이름을 이런식으로 짓는 게 대부분의 절차적 매크로가 따르는 관습입니다.

이 함수는 먼저 `to_string` 을 호출하여 `TokenStream` 인 `input` 을 `String` 으로 변환합니다. 이 `String` 은 `HelloMacro` 를 derive 한 러스트 코드에 해당합니다. 즉 Listing D-2 같은 경우에 `s` 는 `# [derive(HelloMacro)]` 어노테이션을 추가한 부분의 러스트 코드인 `struct Pancakes;` 를 `String` 값으로 지닐 것입니다.

Note: 이 내용이 작성된 시점에 `TokenStream` 은 문자열로만 변환 가능했습니다. 이 시점 이후엔 더 많은 API 가 제공될 겁니다.

이제 러스트 코드 `String` 을 우리가 해석하고, 연산을 수행할 수 있는 자료구조로 파싱해야합니다. `syn` 이 활약할 시간이 왔네요. `syn` 내 `parse_derive_input` 함수는 `String` 을 인자로 받아 러스트 코드를 파싱하여 `DeriveInput` 라는 구조체로 반환합니다. 다음 코드는 `struct Pancakes` 문자열을 파싱해서 얻은 `DeriveInput` 구조체 내용 중 문자열과 관련된 부분을 나타냅니다:

```
DeriveInput {
    // --snip--

    ident: Ident(
        "Pancakes"
    ),
    body: Struct(
        Unit
    )
}
```

이 구조체 필드는 우리가 파싱한 러스트 코드가 `Pancakes` 라는 `ident` (식별자, 즉 이름) 를 갖는 유닛 구조체라는 것을 나타냅니다. 물론 여기 나온 필드 이외에도 많은 필드가 모든 종류의 러스트 코드를 기술하기 위해 존재합니다; 자세한 내용을 원하시는 분은 `syn` 의 `DeriveInput` 문서를 참고하세요.

우린 새로운 러스트 코드를 생성할 `impl_hello_macro` 함수를 아직 정의하지 않았습니다. 하지만 이 함수를 정의하기에 앞서, `hello_macro_derive` 함수 맨 끝에서 `quote` 크레이트의 `parse` 함수를 이용해 `impl_hello_macro` 함수 출력력을 `TokenStream` 으로 변환한 것에 주목해주세요. 반환된

`TokenStream` 은 크레이트 사용자가 작성한 코드에 추가될 것이고, 이로써 사용자는 자신의 크레이트를 컴파일 할 때 우리가 제공한 추가적인 기능을 갖게 됩니다.

눈치 채셨을진 모르겠지만 여기선 `parse_derive_input` 이나 `parse` 함수를 호출하는 데 실패하면 `unwrap` 을 호출해 패닉을 일으키도록 되어 있습니다. 절차적 매크로 API 에 따르면 `proc_macro_derive` 에선 `Result` 가 아닌 `TokenStream` 을 반환해야 하기 때문에, 절차적 매크로 코드에서 에러가 발생했을 경우 패닉을 일으키는 것은 필수적입니다. 우린 예제를 간단히 하기 위해 `unwrap` 을 사용했지만, 실제 프로덕션 코드에선 `panic!` 이나 `expect` 를 사용해 정확히 무엇이 잘못됐는지 자세히 설명해주는 에러 메시지를 제공해야 합니다.

어노테이션된 러스트 코드를 `TokenStream` 에서 `String` 과 `DeriveInput` 인스턴스로 변환하는 코드를 작성했으니, 어노테이션 된 타입에 `HelloMacro` 트레이트 구현하는 코드를 만들 차례입니다.

Filename: hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> quote::Tokens {
    let name = &ast.ident;
    quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}", stringify!(#name));
            }
        }
    }
}
```

`ast.ident` 를 이용해 어노테이션 된 타입의 타입명(식별자)을 담고 있는 `Ident` 구조체 인스턴스를 가져왔습니다. Listing D-2 코드의 경우, `name` 값은 `Ident("Pancakes")` 가 됩니다.

`quote!` 매크로는 우리가 반환하고 싶은 러스트 코드를 작성하면 `quote::Tokens` 로 변환해 줍니다. 또한 이 이외에도 `#name` 을 작성하면 `quote!` 는 해당 부분을 `name` 변수의 값으로 대체하는 등, 굉장히 멋진 템플릿 기능을 제공합니다. You can even do some repetition similar to the way regular macros work. 자세한 내용은 [quote 크레이트 문서](#) 를 참고하세요.

우리 목표는 절차적 매크로를 이용해 사용자가 어노테이션을 추가한 타입 (`#name` 으로 가져올 수 있는)에 `HelloMacro` 트레이트 구현체를 생성하도록 하고, 구현된 트레이트는 `hello_macro` 라는 함수 하나를 가지며, 그 함수는 `Hello, Macro! My name is` 와 그 뒤에 어노테이션 된 타입의 이름을 출력하는 기능을 갖도록 하는 것입니다.

여기서 사용한 `stringify!` 매크로는 러스트 안에 포함되어있습니다. 이 매크로는 `1 + 2` 같은 러스트 표현식을 받아서 컴파일 타임에 해당 표현식을 `"1 + 2"` 처럼 스트링 리터럴로 변환합니다. `format!` 이나 `println!` 처럼 표현식을 평가하고 결과를 `String` 으로 변환하는 것과는 다릅니다. `stringify!` 를

사용하는 이유는 `#name` 입력이 그대로 출력돼야 하는 표현식일 수도 있기 때문입니다. 또한 `stringify!`는 컴파일 타임에 `#name` 을 스트링 리터럴로 변환하여, 할당을 절약하는 효과를 가져오기도 합니다.

이 시점에서 `cargo build` 는 `hello_macro` 와 `hello_macro_derive` 양쪽 모두에서 문제 없이 돌아갑니다. 그럼 이제 이 크레이트들과 Listing D-2 코드를 연결해 절차적 매크로가 실제 작동하는 모습을 살펴봅시다. 여러분의 `projects` 디렉토리에 `cargo new --bin pancakes` 를 실행해 새 바이너리 프로젝트를 생성한 뒤, `pancakes` 의 `Cargo.toml` 에 `hello_macro` 와 `hello_macro_derive` 를 디펜던시로 추가하세요. 만약 여러분이 <https://crates.io/> 에 이 크레이트를 배포하셨다면 상관 없겠지만, 그렇지 않다면 이때 다음과 같이 디펜던시에 `path` 를 명시해야 합니다.

```
[dependencies]
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Listing D-2 코드를 `src/main.rs` 에 작성하고, `cargo run` 을 실행해보세요: `Hello, Macro! My name is Pancakes!` 가 출력될 겁니다. `pancakes` 크레이트에서 따로 구현할 필요 없이, 절차적 매크로로부터 만들어진 `HelloMacro` 트레이트 구현체만으로 말이죠; 트레이트 구현체는 `#[derive(HelloMacro)]` 로 인해 추가됩니다.

향후의 매크로

러스트는 앞으로 선언적, 절차적 매크로를 발전시킬 겁니다. `macro` 키워드를 이용해 더 나은 선언적 매크로 시스템을 사용하고, `derive` 보다 더 강력한 여러 작업을 위해 더 많은 종류의 절차적 매크로를 추가할 겁니다. 이 기능들은 이 내용이 작성 된 시점에선 아직 개발중이니, 최근 정보에 대해서는 러스트 온라인 문서를 참조하시기 바랍니다.

부록 E: 본 책의 번역본 목록

영어 원본 이외의 번역본 대다수는 아직 진행중입니다; 각 번역에 참여할 의사가 있는 분은 [번역 라벨 목록](#)을 살펴보시고, 만약 여기에 등록되지 않은 번역본이 있다면 저희에게 알려주세요!

- [Português \(BR\)](#)
- [Português \(PT\)](#)
- [Tiếng việt](#)
- [简体中文, alternate](#)
- [Українська](#)
- [Español](#)
- [Italiano](#)
- [Русский](#)
- [한국어](#)
- [日本語](#)
- [Français](#)
- [Polski](#)
- [עברית](#)
- [Cebuano](#)
- [Tagalog](#)

부록 F: 새로운 기능

이번 부록에선 본 책의 주요 내용이 작성되고 난 이후에 러스트 stable 에 추가된 몇 가지 기능을 다룹니다.

더 짧은 필드 초기화

자료 구조(구조체, 열거형, union)에서 필드명을 갖는 필드를 초기화할 때 `fieldname:fieldname` 을 `fieldname` 으로 줄여서 쓸 수 있습니다. 이 기능은 초기화 구문을 간결하게 만들어 코드 중복을 줄여줍니다.

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

fn main() {
    let name = String::from("Peter");
    let age = 27;

    // 전체 구문
    let peter = Person { name: name, age: age };

    let name = String::from("Portia");
    let age = 27;

    // 단축된 필드 초기화 구문
    let portia = Person { name, age };

    println!("{}:{} {}", portia);
}
```

loop 에서 반환하기

`loop` 는 특정 스레드가 작업을 완료했는지 알아보는 등, 어떤 연산이 실패할 수도 있다는 것을 알고 있을 때, 해당 연산을 재시도 하는 데 사용 가능합니다. 이 연산 결과를 다른 코드로 넘겨 주어야 한다면, `break` 를 이용해 반복을 멈추고 결과를 반환할 수 있습니다:

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    assert_eq!(result, 20);
}
```

중복된 `use` 선언 합치기

모듈을 여러 하위 모듈이 이루고 있어 구조가 복잡한 모듈에서 몇 개의 모듈만 가져와야 할 때, 선언 속 중복되는 부분을 합칠 수 있다면 코드를 깔끔하게 만들 수 있을겁니다.

`use` 선언은 간결한 임포트 및 글룹에서 합칠 수 있습니다. 다음은 `bar` 과 `Foo`, 그리고 `baz` 와 `Bar` 내 모든 항목을 가져오는 예시입니다:

```
use foo::{
    bar::{self, Foo},
    baz: {*, quux::Bar},
};
```

포괄적인 범위 표현

앞서 범위 문법(`..` 와 `...` 를 말합니다)를 사용할 때, 표현식에선 상한을 포함하지 않는 `..` 를 사용하고, 패턴에선 상한을 포함하는 `...` 를 사용했습니다. 하지만 이제 `...=` 하나로 표현식과 패턴 모두에서 사용할 수 있습니다.

```
fn main() {
    for i in 0 ..= 10 {
        match i {
            0 ..= 5 => println!("{}: low", i),
            6 ..= 10 => println!("{}: high", i),
            _ => println!("{}: out of range", i),
        }
    }
}
```

match 내에선 `...` 를 사용해도 문제는 없지만 표현식에선 사용할 수 없으니 `..=` 를 권장합니다.

128 비트 정수

128 비트 정수가 러스트 1.26.0 에 추가됐습니다:

- `u128`: 부호가 없으며 $[0, 2^{128} - 1]$ 범위를 갖는 128 비트 정수
- `i128`: 부호가 있으며 $[-(2^{127}), 2^{127} - 1]$ 범위를 갖는 128 비트 정수

이들은 LLVM 을 통해 효율적으로 구현됐기 때문에, 128 비트 정수를 지원하지 않는 플랫폼에서도 다른 정수 타입들과 마찬가지 방식으로 사용 가능합니다.

이 기능은 암호화 알고리즘 등, 아주 큰 정수를 효율적으로 다뤄야 하는 알고리즘에서 유용할 겁니다.

부록 G: 러스트가 만들어지는 과정과 "Rust Nightly"

다음 내용은 러스트가 만들어지는 과정과 해당 과정이 러스트 개발자인 여러분에게 어떤 영향이 미치는지를 다룹니다. 앞서 이 책의 예제들은 러스트 stable 1.21.0 버전을 기준으로 만들었다고 했지만 모든 예제는 더 높은 버전에서도 작동할 겁니다. 어떻게 이런 일이 가능한지 알아보도록 합시다.

막힘 없이 안정된 발전 (Stability Without Stagnation)

러스트는 하나의 언어로서 여러분 코드를 여러 방면으로 관리하는 동시에 여러분이 안심하고 위에 무언가 지을 수 있는 단단한 기반을 마련해야 합니다. 언어가 자주 변경된다면 이 목표는 달성하기 힘들어지겠죠. 하지만 새로운 기능을 시험해보지 못한다면 심각한 문제가 있더라도 릴리즈 이전에 알아채지 못할 겁니다. 물론, 릴리즈 이후엔 고칠 방법이 없습니다.

이러한 문제의 해결법이 바로 러스트의 유도 원칙이기도 한 "막힘 없이 안정된 발전(Stability Without Stagnation)"입니다: 모든 업데이트에서 발생하는 문제를 최소화하고, 새로운 기능은 착실히 추가하되, 버그를 줄이고 컴파일 속도를 높여서 사용자들이 마음 편히 업데이트할 수 있게 하는 것이 목표입니다.

릴리즈 채널 열차가 달려갑니다

러스트 개발은 열차 스케줄 (*train schedule*)로 운영됩니다. 모든 개발은 러스트 저장소의 `master` 브랜치에서 완료됩니다. 릴리즈 방식은 Cisco IOS를 비롯한 여러 소프트웨어 프로젝트에서 사용하는 "train model"을 사용합니다. 다음은 러스트의 3 가지 릴리즈 채널입니다:

- Nightly
- Beta
- Stable

러스트 개발자는 대부분 stable 채널을 주로 사용하며, 새 기능을 사용해보려는 사람들은 nightly나 beta 채널을 사용하기도 합니다.

개발 및 릴리즈 과정은 다음 예시처럼 돌아갑니다: 러스트 개발 팀이 러스트 1.5 버전 릴리즈 작업을 하고 있다고 가정해봅시다. (1.5 버전은 2015년 12월에 릴리즈되었지만, 현실적인 버전 가정을 위해 이와 같이 설정했습니다) 러스트에 새 기능이 추가됐습니다. 다시 말해, `master` 브랜치에 새 커밋이 올라갔습니다. 매일 밤, 러스트 nightly 버전이 릴리즈됩니다. 매일 밤 생성되는 이 릴리즈는 러스트 릴리즈 인프라가 자동으로 생성합니다. 시간이 지남에 따라 러스트 릴리즈는 다음과 같은 모습이 됩니다:

```
nightly: * - - * - - *
```

beta 브랜치는 6 주마다 nightly 에 사용되는 **master** 브랜치로부터 떨어져 나와 생성됩니다. 이제 릴리즈는 두 종류가 됐네요:

```
nightly: * - - * - - *  
|  
beta:      *
```

beta 릴리즈는 사용하는 사람이 그다지 많지 않지만, 러스트는 CI 시스템을 이용해 가능한 한 문제점을 찾으려고 노력합니다. 이 동안에도 nightly 는 매일 밤 릴리즈됩니다.

```
nightly: * - - * - - * - - * - - *  
|  
beta:      *
```

문제점이 발견됐다고 가정해봅시다. 오류가 stable 릴리즈로 넘어가기 전에 beta 릴리즈 테스트에서 잡아냈다는 게 불행 중 다행이네요. 오류 수정 내용을 **master** 브랜치에 반영하면 자연스레 nightly 버전이 고쳐집니다. 이후 해당 내용이 **beta** 브랜치에 백포트되고(backport, 상위 버전의 기능을 하위 버전에 반영하는 것을 말함) 나면 새로운 beta 릴리즈가 제공됩니다:

```
nightly: * - - * - - * - - * - - *  
|  
beta:      * - - - - - - - - *
```

첫 베타 버전이 만들어지고 6 주가 지나면, **stable** 브랜치가 **beta** 브랜치로부터 만들어져 stable 릴리즈가 생성됩니다.

```
nightly: * - - * - - * - - * - - * - - *  
|  
beta:      * - - - - - - - - *  
|  
stable:    *
```

마침내 러스트 1.5 버전을 완성했습니다! 하지만, 그동안 6주가 지나버렸기 때문에 다음 버전이 될 1.6 버전에 대응할 새로운 beta 가 필요합니다. 따라서 **beta** 브랜치는 **stable** 버전이 만들어진 후에도 계속 **nightly** 브랜치로부터 떨어져 나옵니다:

```
nightly: * - - * - - * - - * - - * - * - *
          |           |
beta:      * - - - - - - - - *
          |
stable:    *
```

이 과정을 "train model" 이라고 부르는 이유는 6주마다 이루어지는 릴리즈가 "열차가 역을 지나는" 것과 유사하기 때문입니다. 그리고 열차는 다음 stable 릴리즈 역에 도착할 때까지 beta 채널 위를 달리겠죠.

러스트는 6주마다 일정하게 릴리즈됩니다. 릴리즈 날짜를 하나 알고 있다면 6주를 더해 다음 릴리즈 일을 알 아낼 수도 있죠. 일정한 주기로 릴리즈 하여 얻는 장점은 언제가 될지 모를 릴리즈일을 하염없이 기다릴 필요가 없단 점입니다. 어떤 기능이 특정 릴리즈에 누락되더라도 곧 있으면 다음 릴리즈가 생성될 테니 걱정할 필요 없죠. 이 방식은 아직 다듬을 필요가 있는 기능을 개발 중인 개발자들의 릴리즈 기한을 맞춰야 한다는 압박감을 줄여주기도 합니다.

덕분에 여러분들은 언제나 다음 러스트 빌드를 확인할 수 있고 버전을 업그레이드 하는 데에도 부담이 없습니다: 다만 가끔씩 beta 릴리즈에서 문제가 발생하기도 합니다. 모든 소프트웨어에는 버그가 존재할 수 있는 법이고, `rustc` 도 소프트웨어니까요. 하지만 beta 릴리즈에 문제가 있더라도 개발팀에 제보하면 stable 릴리즈 이전에 해당 오류를 수정받을 수 있습니다.

불안정한 기능

릴리즈 모델에서 하나 더 짚고 넘어갈 게 있습니다: 바로 '불안정한 기능'입니다. 러스트에는 "Feature flags"라는 기술이 적용됐기 때문에 릴리즈에서 어떤 기능을 활성화하거나, 비활성화 할 수 있습니다. 예를 들어, 아직 개발중인 기능이 `master` 에 추가되면 자연스레 nightly 에도 추가되지만 *feature flag*에 가려진 상태로 추가됩니다. 따라서, 아직 개발중인 기능을 사용하고 싶은 분은 nightly 릴리즈에서 적절한 flag를 소스코드에 명시하셔야 합니다.

알아두실 것은 feature flag 는 어디까지나 새로운 기능이 stable 에 정착되기 전에 연습해볼 수 있도록 하는 용도이기 때문에 beta 나 stable 릴리즈에선 feature flag 를 사용할 수 없습니다. 따라서 안정적인 환경을 원하는 사용자는 이 기능을 이용하지 않는 것을 추천드립니다.

본 책의 내용은 stable 릴리즈의 기능만 담고 있습니다. 개발 중인 기능에 관한 내용은 언제든 변경될 수 있으므로 책에 작성된 내용과 실제 stable 빌드에 추가된 내용이 전혀 다를 가능성이 있기 때문입니다. 따라서 nightly 에만 존재하는 기능은 온라인에서 관련 문서를 찾아보시기 바랍니다.

Rustup 과 Rust Nightly 의 역할

여러분은 stable 러스트를 설치하셨을 겁니다. 하지만 특정 프로젝트에선 특정 릴리즈 채널을 사용하도록 설

정하거나, 혹은 글로벌 설정을 변경하고 싶다면 어떻게 해야 할까요? 답은 Rustup입니다. rustup으로 nightly 를 설치하는 방법은 다음과 같습니다:

```
$ rustup install nightly
```

rustup 으로 여러분이 설치한 모든 툴체인(toolchains) (여러 러스트 릴리즈와 관련 컴포넌트를 포함한 것을 말합니다)를 확인할 수도 있습니다. 필자의 윈도우 컴퓨터에서 실행한 결과를 예로 가져왔습니다:

```
> rustup toolchain list  
stable-x86_64-pc-windows-msvc (default)  
beta-x86_64-pc-windows-msvc  
nightly-x86_64-pc-windows-msvc
```

보시다시피 기본 설정된 툴체인은 stable 입니다. 러스트 사용자들은 대부분 stable 을 사용하긴 하지만 최첨단 기능을 고려해야 하는 특정 프로젝트에선 nigthly 가 필요할 수도 있습니다. 이처럼 어떤 프로젝트만 다른 툴체인을 사용하고 싶을 땐 해당 프로젝트 디렉토리에서 `rustup override` 를 이용해 원하는 툴체인을 사용할 수 있습니다:

```
$ cd ~/projects/needs-nightly  
$ rustup override add nightly
```

이제 `rustup` 은 `~/projects/needs-nightly` 에서 `rustc` 나 `cargo` 가 호출될 때마다 기본값인 stable 대신 nigthly 를 사용하고 있는지 확인합니다. 덕분에 앞으로 여러분이 관리할 러스트 프로젝트가 많아지더라도 관리가 편해질 겁니다.

RFC 과정과 러스트 RFC 팀

이런 새로운 기능에 관한 정보는 어디서 볼 수 있을까요? 러스트 개발 모델은 *RFC (Request For Comments)* 과정을 따릅니다. 따라서 러스트를 더 발전시키고 싶다면, 제안서, 즉 RFC 를 작성하면 됩니다.

RFC는 러스트를 발전시키고 싶은 사람이라면 누구든 작성할 수 있습니다. 여러분이 작성한 RFC는 해당 주제에 연관된 러스트 팀에서 제안 내용을 읽고, 토론과 리뷰를 진행한 뒤 직접 의견을 남기며, 최종적으로는 해당 기능을 받아들일지 말지 합의합니다. (러스트에는 언어 디자인, 컴파일러 구현, 인프라, 문서화 등 다양한 세부 분야에 대응하는 팀이 존재하며, 전체 팀 목록은 [러스트 웹사이트](#) 에서 확인하실 수 있습니다.)

제안이 받아들여지면 누군가 구현할 수 있도록 러스트 저장소에 이슈가 등록됩니다. 이때, 기능을 구현한 사람과 기능을 제안한 사람이 다를 수도 있습니다. 어찌 됐건 구현되고 나면 "불안정한 기능" 절에서 다른 대로 feature flag 에 가려진 채 `master` 브랜치에 올라갑니다.

시간이 지나 nightly 릴리즈 사용자가 해당 기능을 사용할 수 있게 되면 러스트 팀은 해당 기능이 nightly 에

서 유용하게 쓰였는지 의논하고, stable 에 포함할 것인지 결정합니다. 의논 결과가 긍정적이라면 feature flag 의 그림자로부터 나와 예비 stable 로 취급됩니다. 다음 stable 릴리즈 역에서 러스트 릴리즈 열차에 탑승하는 거죠.

부록 H - 번역 용어 정리

이 절은 한국어 번역본에만 포함되어 있는 절로서, 원문에 대한 번역 용어를 정리한 곳입니다.

- abort: 그만두기
- allocating: 할당
- annotation: 어노테이션, 주석
- arm: (match 문에서의 arm) 갈래
- atomic: 아토믹
- attribute: 속성
- assert: 단언하다, 단정
- assertion: 어서션, 단언
- assign: 대입하다
- associated function: 연관함수
- associated type: 연관 타입
- automatic referencing and dereferencing: 자동 참조 및 역참조
- backtrace: 백트레이스
- binary: 바이너리
- binary target: 바이너리 타겟
- bind: 묶다, 바인드
- boilerplate code: 보일러플레이트 코드
- borrowing: 빌림
- borrow checker: 빌림 검사기
- bounded parametric polymorphism: 범주내 매개변수형 다형성
- box: 박스
- buffer: 버퍼
- buffer overread: 버퍼 오버리드
- CamelCase: 낙타 표기법
- cargo: 카고
- channel: 채널
- clone: 클론
- collection: 컬렉션
- crate: 크레이트
- copy: 복사
- concatenation: 접합
- concrete type: 구체 타입
- concern: 핵심기능
- connection accept: 연결 수락
- construct function: 생성 함수

- constructor: 생성자
- configuration: 환경 설정
- control flow: 제어문
- crate: 크레이트
- concurrency: 동시성
- concurrent programming: 동시성 프로그래밍
- dangling pointer: 땅글링 포인터
- dangling reference: 땅글링 참조자
- data race: 데이터 레이스
- deadlock: 데드록, 교착 상태
- deep copy: 깊은 복사
- dependency: 디펜던시, 의존성
- deref coercion: 역참조 강제
- dereference operator: 역참조 연산자
- derivable: 추론 가능한
- derivable traits: 파생 가능한 트레이트
- derived trait: 파생 트레이트
- destructuring: 해체, 디스트럭처링
- destructor: 소멸자
- diverging function: 발산 함수
- doc comments: 문서 주석
- documentation comments: 문서화 주석
- double free: 중복 해제
- **drop**: **drop** 한다, 버리다
- duck typing: 오리 타이핑
- dynamically sized type: 동적인 크기의 타입
- empty type: 빈 타입
- enumerate: 나열
- enumerations: 열거형
- equality: 동치
- exception: 예외
- expression: 표현식
- external crate: 외부 크레이트
- feature: 특성
- fearless concurrency: 겁없는 동시성
- format string: 형식 문자열
- fully qualified syntax: 완전 정규화 문법
- function: 함수
- GC: 가비지 콜렉터

- generic: 제네릭
- getter: 게터
- glob: 글롭
- global variable: 전역 변수
- grapheme cluster: 문자소 클러스터
- growable: 확장 가능한
- halting problem: 정지 문제
- handle: 핸들
- hasher: 해시어
- heap: 힙
- identifier: 식별자
- immutable: 불변
- input lifetime: 입력 라이프타임
- indirection: 간접
- inequality: 부동
- instance: 인스턴스
- interior mutability: 내부 가변성
- integration test: 통합 테스트
- invariant: 불변성
- irrefutable pattern: 반증 불가 패턴
- iteration: 반복
- iterator: 반복자
- lazy evaluation: 자연 평가
- license: 라이센스
- license identifier value: 라이센스 식별자 값
- library: 라이브러리
- lifetime: 라이프타임
- lifetime bound: 라이프타임 바운드
- lifetime elision rules: 라이프타임 생략 규칙
- literal: 리터럴, 상수
- macro: 매크로
- mangling: 맹글링
- memoization: 메모이제이션
- memory leak: 메모리 릭, 메모리 누수
- message passing: 메세지 패싱
- meta data: 메타 데이터
- method: 메소드
- method syntax: 메소드 문법
- mock: 목

- mock object: 목 객체
- module: 모듈
- monomorphization: 단형성화
- move: 이동
- mutable: 가변
- mutable static variable: 가변 정적 변수
- mutex: 뮤텍스
- mutual exclusion: 상호 배제
- named variables: 명명 변수
- namespace: 이름공간
- naming conflict: 이름 간의 충돌
- never type: 부정 타입
- newtype pattern: 뉴타입 패턴
- nightly: 나이틀리
- nomicon: 노미콘
- operator overloading: 연산자 오버로딩
- orphan rule: 고아 규칙
- output lifetime: 입력 라이프타임
- ownership: 소유권
- panic: 패닉
- parallel programming: 병렬 프로그래밍
- pig Latin: 피그 라틴
- placeholder: 변경자
- pointer: 포인터
- polymorphism: 다형성
- prelude: 프렐루드
- privacy rules: 비공개 규칙
- private: 비공개
- public: 공개
- public item: 공개 항목
- publish: 배포
- race condition: 경쟁 조건
- raw pointer: 로우 포인터
- receiver: 수신자
- recover: 복구
- recoverable: 복구 가능한
- recursive type: 재귀적 타입
- reference: 참조자
- reference counting: 참조 카운팅

- reference cycle: 참조 순환
- refutability: 반증 가능성
- refutable pattern: 반증 가능 패턴
- regression: 변경점
- registry: 레지스트리, 등기소
- release: 릴리즈
- release profiles: 릴리즈 프로필
- return: 반환
- rust: 러스트
- scope: 스코프
- seed: 시드
- segmentation fault: 세그먼테이션 폴트
- Semantic Versioning rules: 유의적 버전 규칙
- semantics: 의미론
- separator: 구분자
- shallow copy: 얕은 복사
- shorthand: 약칭 (구문)
- signature: 시그니처
- slice: 슬라이스
- smart pointer: 스마트 포인터
- stack: 스택
- statement: 구문
- static dispatch: 정적 디스패치
- static lifetime: 정적 라이프타임
- string literal: 스트링 리터럴
- subtyping: 서브타이핑
- supertrait: 슈퍼트레잇
- syntax: 구문, 문법
- syntax sugar: 문법적 설탕
- test double: 테스트 더블
- test harness: 테스트 도구
- trait: 트레이트
- trait bound: 트레이트 바운드
- trait object: 트레이트 객체
- transmitter: 송신자
- type alias: 타입 별칭
- type annotation: 타입 명시
- unit test: 단위 테스트
- unrecoverable: 복구 불가능한

- Unrolling: 언롤링
- unsized type: 크기 없는 타입
- unwinding: 되감기
- variable: 변수
- visibility: 가시성
- workspace: 작업공간
- weak reference: 약한 참조