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, 3

장의 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 값들을 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 트레잇(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 봐온 기본 타입들은 `Display`가 기본적으로 구현되어 있는데, 이는 `1` 혹은 다른 기본 타입을 유저에게 보여주고자 하는 방법이 딱 한가지기 때문입니다. 하지만 구조체를 사용하는 경우, `println!`이 출력을 형식화하는 방법은 덜 명확한데 이는 표시 방법의 가능성이 더 많기 때문입니다: 여러분은 쉼표를 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 트레잇입니다.

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 보

는 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);
}
```

Listing5-11: `Debug` 트레잇을 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 디버깅 하는 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 트레잇을 제공하여 우리의 커스텀 타입에 대해 유용한 동작을 추가할 수 있도록 해줍니다. 이 트레잇들과 그 동작들은 부록 C에서 그 목록을 찾을 수 있습니다. 10장에서는 이 트레잇들을 커스터마이징된 동작을 수행하도록 구현하는 방법 뿐만 아니라 우리만의 트레잇을 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.