`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 패

턴과 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 분류기와 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 `UsSate` 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 있습니

다! 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 칩시다. 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`를 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)`랑 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`와 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 샅샅이 다루어야 합니다. 특히 `Option<T>`의 경우, 즉 러스트가 우리로 하여금 `None` 케이스를 명시적으로 다루는 일을 잊는 것을 방지하는 경우에는, Null일지도 모를 값을 가지고 있음을 가정하여, 앞서 논의했던 수십억 달러짜리 실수를 하는 일을 방지해줍니다.

변경자(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 그 전에 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.