4_Control flow_en

May 2, 2023

1 Control flow

The most common constructs that let you control the flow of execution of Rust code are **if** expressions and **loop**s.

1.1 *if* Expressions

[ ]:
```
fn main() {
    let number = 3;

    if number < 5 { // conditions are sometimes called "arms" and must be␣
 ↪boolean
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
main();
```

[ ]:
```
fn main() {
    let number = 3;

    if number { // the condition must be bool, otherwise we get an error !
        println!("number was three");
    }
}
main();
```

Unlike languages such as Ruby and JavaScript, Rust will not automatically try to convert non-Boolean types to a Boolean.

1.2 Multiple conditions with *else if*

[ ]:
```
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");
    }
}
main();
```

Note that even though 6 is divisible by 2, we don't see the output number is divisible by 2, nor do we see the number is not divisible by 4, 3, or 2 text from the else block. That's because Rust only executes the block for the first true condition, and once it finds one, it doesn't even check the rest.

1.3 Using *if* in a *let* statement

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

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

[ ]:
```
fn main() {
    let condition = true;

    let number = if condition {
        5   // result is evaluated to an integer
    } else {
        "six" // type mismatch
    };

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

Rust needs to know at compile time what type the `number` variable is, definitively, so it can verify at compile time that its type is valid everywhere we use number. Rust wouldn't be able to do that if the type of number was only determined at runtime; the compiler would be more complex and would make fewer guarantees about the code if it had to keep track of multiple hypothetical types for any variable.

2 Loops

2.1 Repeating Code with *loop*

[ ]:
```
fn main() {
    loop {  // type ^C to stop the loop
        println!("again!");
    }
}
main();

// WARNING, do not run this cell
// if you do, you need to restart the Rust kernel (menu Kernel/restart)
```

One of the uses of a loop is to retry an operation you know can fail, such as checking if a thread completed its job. However, you might need to pass the result of that operation to the rest of your code. If you add the value you want to return after the break expression you use to stop the loop, it will be returned out of the loop so you can use the value, as shown here:

[ ]:
```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {}", result);
}
main();
```

2.2 Conditional Loops with *while*

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

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

        number = number - 1;
    }

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

```
main();
```

[ ]:
```
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;
    }
}
main();
```

This approach is error prone; we could cause the program to panic if the index length is incorrect. It's also slow, because the compiler adds runtime code to perform the conditional check on every element on every iteration through the loop. As a more concise alternative, you can use a for loop and execute some code for each item in a collection.

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

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

[ ]:
```
fn main() {
    for number in (1..4).rev() { // note the sequence definition using the ..␣
 ↪operator
        println!("{}!", number);
    }
    println!("LIFTOFF!!!");
}
main();
```

3 Exercises

Exercise 1: Implement a fizz buzz game that counts up to 20.

For each number, print * `Fizz` if the number is divisible by 3 * `Buzz` if the number is divisible by 5 * `Fizz Buzz` if the number is divisible by 15 * The number itself, otherwise

[ ]:
```
fn main() {
    // TO DO Implement the fizz buzz game
```

```
}
main();
```

Solution:

[ ]:
```
fn main() {
    for number in 1..21 {
        if number % 3 == 0 {
            if number % 5 == 0 {
                println!("Fizz Buzz");
            } else {
                println!("Fizz");
            }
        } else if number % 5 == 0 {
            println!("Buzz");
        }
        else {
            println!("{}", number);
        }
    }
}
main();
```

Exercise 2 : Fibonacci * Implement a recursive version of a function that compute the n-th term of the Fibonacci series. * Implement an iterative version. * Compute a table of the 50 first terms of the series.

[ ]:
```
fn main() {

}
main();
```

Solution:

[ ]:
```
fn fibonacci_recursive(n: u32) -> u32 {
    if n == 0 {
        0
    } else if n == 1 {
        1
    } else {
        fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
    }
}

fn fibonacci_iterative(n: u32) -> u32 {
    if n == 0 {
        return 0;
    }
```

```
    let mut n0 = 0;
    let mut n1 = 1;
    for _ in 2..(n+1) {
        let next = n0 + n1;
        n0 = n1;
        n1 = next;
    }
    n1
}

fn fibonacci_firsts() -> [u32 ; 10] {
    let mut result = [0 ; 10];
    result[1] = 1;
    for i in 2..10 {
        result[i] = result[i-2] + result[i-1];
    }
    result
}

fn main() {
    println!("fibonacci_recursive(10) = {}", fibonacci_recursive(10));
    println!("fibonacci_iterative(10) = {}", fibonacci_iterative(10));
    println!("fibonacci_firsts = {:?}", fibonacci_firsts());
}

main();
```