

ECE326

PROGRAMMING LANGUAGES

Lecture 14 : Control Flow in Rust

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Control Flow

- No parenthesis
 - Conditions in control flow does not need parenthesis
 - Unlike in C/C++
- Expressions
 - Almost everything in Rust is an expression
 - Including loops and blocks
 - Their return value is the last expression in each block
 - do not add a semicolon, which makes it a statement
 - All branches must return the same type
 - Otherwise the compiler complains

If/Else

```
let n = 5;
if n < 0 {
    print!("{}", is negative", n);
} else if n > 0 {
    print!("{}", is positive", n);
} else {
    print!("{}", is zero", n);
}
```

← normal usage

```
let big_n = if n < 10 && n > -10 {
    println!(", and is a small number, increase ten-fold");
    10 * n // no semicolon here
} else {
    println!(", and is a big number, halve the number");
    n / 2
}; // semicolon here (let is a statement)

println!("{}", -> {}", n, big_n);
```

use as an expression



Loop

- An infinite loop
 - Requires break statement to exit

```
loop {  
    count += 1;  
    if count == 3 {  
        println!("three");  
        // Skip the rest of this iteration  
        continue;  
    }  
    println!("{}", count);  
    if count == 5 {  
        println!("OK, that's enough");  
        // Exit this loop  
        break;  
    }  
}
```

Retry

- Example
 - Loop until “successful”
 - Loop can also be an expression

```
let mut counter = 0;

let result = loop {
    counter += 1;

    if counter == 10 {
        break counter * 2; // return value of loop
    }

    println!("I only exit on 10");
};
```

While Loop

- Loop while condition is true

```
let mut n = 1;

while n < 101 {
    if n % 15 == 0 {
        println!("fizzbuzz");
    } else if n % 3 == 0 {
        println!("fizz");
    } else if n % 5 == 0 {
        println!("buzz");
    } else {
        println!("{}", n);
    }

    n += 1;
}
```

For Loop

- Uses iterator to loop through elements of a collection
- Range
 - Similar to Python, creates an iterator for integers

- `a..b`

- Loops from `a` to `b-1`

```
for n in 1..101
```

- `a..=b`

- Loops from `a` to `b`

```
for n in 1..=100
```

Vector

- Similar to `std::vector`
 - Element type can be inferred by first element inserted
- `vec!`
 - Macro to initialize the vector

```
let mut v = vec![1, 2, 3]; // type of v is Vec<i32>
v.push(5);                // add more elements to v
v.push(6);
```

```
for i in &v {              // read-only loop
    println!("{}", i);
}
for i in &mut v {          // read-update loop
    *i += 10;
}
```


Match

- The switch statement of Rust
 - Except much more powerful and more frequently used
- Has ability to pattern match
 - Known as “destructuring”
- Can specify conditions
 - Similar to a set of if...else if statements
- Compiler requires all possible values to be handled
 - E.g. it is an error to miss one of the enum values

Match

- Syntax

```
match expression {  
    pattern => expression, // this is called an "arm"  
    pattern => expression,  
    ...  
}
```

- Example

```
let boolean = true  
let binary = match boolean {  
    false => 0,  
    true => 1,  
};  
println!("{}", boolean, binary);
```

Match

- Match on an integer

```
let number = 13;
```

```
println!("Tell me about {}", number);
match number {
    // Match a single value
    1 => println!("One!"),
    // Match several values
    2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
    // Match an inclusive range
    13...19 => println!("A teen"),
    // Handle the rest of cases
    _ => println!("Ain't special"),
}
```

... is inclusive, .. is exclusive (e.g. 13..19 matches 13 to 18)

_ is the catch-all case (same as default in C/C++)

Destructuring

- Allows matching of value(s) inside tuples

```
let pair = (0, -2);  
match pair {  
  (0, y) => println!("First is `0` and `y` is `{:?}`", y),  
  (x, 0) => println!("`x` is `{:?}` and last is `0`", x),  
  _ => println!("It doesn't matter what they are"),  
}
```

- Guards (the if conditions)

```
match pair {  
  (x, y) if x == y => println!("These are twins"),  
  (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),  
  (x, _) if x % 2 == 1 => println!("The first one is odd"),  
  _ => println!("No correlation..."),  
}
```

Enum in Rust

- Much more powerful than enums in C/C++
- Similar to enum class, values must be scoped
- Unlike C/C++, can *never* be assigned an integer value

```
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,  
    } // no semicolon here so we return what match returns  
}
```

Variant Type

- Also known as tagged union in C/C++
 - Algebraic data type in functional programming languages
- Data can be placed directly into each variant of enum
 - Each variant can have its own set of data

```
enum IpAddr {  
    V4(u8, u8, u8, u8),  
    V6(String),  
}
```

```
let home = IpAddr::V4(127, 0, 0, 1);  
let loopback = IpAddr::V6(String::from("::1"));
```

Destructuring

- Access data inside an enum

```
enum Color {  
    RGB(u32, u32, u32),  
    HSV(u32, u32, u32),  
    CMYK(u32, u32, u32, u32),  
}
```

```
match color {  
    // binds name r, g, b to each piece of data inside variant RGB  
    Color::RGB(r, g, b) =>  
        println!("Red: {}, green: {}, and blue: {}!", r, g, b),  
    Color::HSV(h, s, v) =>  
        println!("Hue: {}, saturation: {}, value: {}!", h, s, v),  
    Color::CMYK(c, m, y, k) =>  
        println!("Cyan: {}, magenta: {}, yellow: {}, key (black): {}!",  
            c, m, y, k),  
}
```

Option<T>

- Commonly used enum
- Replaces use of null pointer
 - provides safety for “pointers” that can be null

```
enum Option<T> { // already defined in standard library
    Some(T),      // as a generic type
    None,
}
```

```
let x = Some(5);
let y = match x {
    None => None,
    Some(i) => Some(i + 1),
};
```


If let

- Useful for matching one specific enum
 - Looks much cleaner than match

```
// syntax 1: using match
match optional {
  Some(i) => {
    println!("This is a really long string and `{:?}`", i);
  },
  _ => {}, // required because match is exhaustive
};

// syntax 2: using if let
if let Some(i) = letter {
  println!("Matched {:?}", i);
} else {
  // Destructure failed. Change to the failure case.
  println!("Was not a Some!");
}
```

Result<T, E>

- Commonly used for error handling

```
enum Result<T, E> { // already defined in standard library
    Ok(T),           // as a generic type
    Err(E),
}

let f = File::open("hello.txt");
let f = match f {
    Ok(file) => file,
    Err(error) => {
        // this is how you write multiline string in Rust
        panic!("There was a problem opening the file: \
{:?}", error)
    },
};
```

Error Handling

- `unwrap()` / `expect(msg)`
 - Attempt to access data inside `Ok`, panic if `Err` is found

```
let f = File::open("hello.txt").unwrap();  
let f = File::open("hello.txt").expect("Failed to open hello.txt");
```

- `? operator`
 - Returns the same error inside `Err()` immediately
 - Requires return type of function to also be `Result<T, E>`

```
fn read_username_from_file() -> Result<String, io::Error> {  
    let mut f = File::open("hello.txt"?;    // returns on error  
    let mut s = String::new();  
    f.read_to_string(&mut s)?;              // returns on error  
    Ok(s)  
}
```

Example

```
pub fn parse(tokens: Vec<String>)
    -> Result<Vec<Table>, &'static str> {
    let mut tables: Vec<Table> = vec![];
    let mut it = tokens.iter();
    loop {
        let table = match parse_table(&mut it, &tables)? {
            Some(table) => table,
            None => break,
        };
        tables.push(table);
    }
    if tables.len() == 0 {
        Err("schema file is empty")
    } else {
        Ok(tables)
    }
}
```