南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Advanced Programming

Prof. Shiqi Yu (于仕琪)

yusq@sustech.edu.cn

http://faculty.sustech.edu.cn/yusq/

Rust

Part 1

Most contents are from ***Tour of Rust*** https://tourofrust.com/

The Basics of Rust

Variables

- The **let** keyword.
- Data type

```rust
fn main() {
    // rust infers the type of x
    let x = 13;
    println!("{}", x);

    // rust can also be explicit about the type
    let x: f64 = 3.14159;
    println!("{}", x);

    // rust can also declare and initialize later, but this is rarely done
    let x;
    x = 0;
    println!("{}", x);
}
```

Variables

- **mutable** - the compiler will allow the variable to be written to and read from.
- **immutable** - the compiler will only allow the variable to be read from.

```rust
fn main() {
    let mut x = 42;
    println!("{}", x);
    x = 13;
    println!("{}", x);
}
```

Variable Shadowing

- **variable shadowing** allows you to reuse a variable name in the same scope to bind new values or even change types, while maintaining memory safety guarantees.

```rust
let x = 5;
let x = x * 2;
let x = "hello"; // New String type shadows previous integer

let y = 10;
{
   let y = y * 2; // Inner scope shadowing
}

// Shadowing (type change allowed)
let data = 42;
let data = data.to_string(); // i32 → String

// Mutation (type must match)
let mut value = 42;
value = 100; // Valid
value = "text"; // Compile error
```

Basic Types

- booleans - **bool** for representing true/false
- unsigned integers - **u8 u16 u32 u64 u128** for representing nonnegative whole numbers
- signed integers - **i8 i16 i32 i64 i128** for representing whole numbers
- pointer sized integers - **usize isize** for representing indexes and sizes of things in memory
- floating point - **f32 f64**
- characters - **char** for representing a single Unicode character
- tuple - **(value, value, ...)** for passing fixed sequences of values on the stack
- arrays - **[value, value, ...]** a collection of similar elements with fixed length known at compile time
- slices - a collection of similar elements with length known at runtime
- **str**(string slice) - text with a length known at runtime

```rust
fn main() {
    let x = 12; // by default this is i32
    let a = 12u8;
    let b = 4.3; // by default this is f64
    let c = 4.3f32;
    let d = 'r'; // unicode character
    let ferris = '🦀'; // also a unicode character
    let bv = true;
    let t = (13, false);
    let sentence = "hello world!";
    println!(
        "{} {} {} {} {} {} {} {} {} {}",
        x, a, b, c, d, ferris, bv, t.0, t.1, sentence
    );
}
```

Basic Type Conversion

- Rust requires explicitness when it comes to numeric types.

```
fn main() {
    let a = 13u8;
    let b = 7u32;
    let c = a as u32 + b;
    println!("{}", c);

    let t = true;
    println!("{}", t as u8);
}
```

Constants

- Unlike variables, constants must always have explicit types.
- Constant names are always in SCREAMING_SNAKE_CASE.

```rust
const PI: f32 = 3.14159;

fn main() {
    const EULER: f32 = 2.71828;
    println!(
        "PI is approximately {} and Euler's number is approximately {}",
        PI, EULER
    );
}
```

Arrays

- An *array* is a **fixed length collection** of data elements all of the same type.

```
fn main() {
    let nums: [i32; 3] = [1, 2, 3];
    println!("{:?}", nums);
    println!("{}", nums[1]);
}
```

Functions

- If you just want to return an expression, you can drop the return keyword and the semicolon at the end, as we did in the *subtract* function.
- Function names are always in snake_case.

```
fn add(x: i32, y: i32) -> i32 {
    return x + y;
}

fn subtract(x: i32, y: i32) -> i32 {
    x - y
}

fn main() {
    println!("42 + 13 = {}", add(42, 13));
    println!("42 - 13 = {}", subtract(42, 13));
}
```

Functions: Multiple Return Values

- Functions can return multiple values by returning a **tuple** of values.
- Tuple elements can be referenced by their index number.

```
fn swap(x: i32, y: i32) -> (i32, i32) {
    return (y, x);
}

fn main() {
    // return a tuple of return values
    let result = swap(123, 321);
    println!("{} {}", result.0, result.1);

    // destructure the tuple into two variables names
    let (a, b) = swap(result.0, result.1);
    println!("{} {}", a, b);
}
```

Functions: Returning Nothing

- If no return type is specified for a function, it returns an empty tuple.

```
fn make_nothing() -> () {
   return ();
}
// the return type is implied as ()
fn make_nothing2() {
   // this function will return () if nothing is specified to return
}
fn main() {
   let a = make_nothing();
   let b = make_nothing2();
   // Printing a debug string for a and b
   // Because it's hard to print nothingness
   println!("The value of a: {:?}", a);
   println!("The value of b: {:?}", b);
}
```

Basic Control Flow

if / else

- Conditions don't have parentheses! Did we ever really need them? Our logic now looks nice and clean.
- All your usual relational and logical operators still work: ==, !=, <, >, <=, >=, !, ||, &&.

```rust
fn main() {
    let x = 42;
    if x < 42 {
        println!("less than 42");
    } else if x == 42 {
        println!("is 42");
    } else {
        println!("greater than 42");
    }
}
```

loop

- infinite loop
- break will escape a loop when you are ready.

```
fn main() {
    let mut x = 0;
    loop {
        x += 1;
        if x == 42 {
            break;
        }
    }
    println!("{}", x);
}
```

while

- while lets you easily add a condition to a loop.
- If the condition evaluates to false, the loop will exit.

```
fn main() {
    let mut x = 0;
    while x != 42 {
        x += 1;
    }
    println!("x is {}", x);
}
```

for

- Rust's for loop is a powerful upgrade. It iterates over values from any expression that evaluates into an iterator.
- An iterator is an object that you can ask the question "What's the next item you have?" until there are no more items.

```rust
fn main() {
    for x in 0..5 {
        println!("{}", x);
    }

    for x in 0..=5 {
        println!("{}", x);
    }
}
```

match

- match is exhaustive so all cases must be handled.
- Similar to switch-case in C/C++.

```
fn main() {
    let x = 42;
     match x {
        0 => {
           println!("found zero");
        }
        // we can match against multiple values
        1 | 2 => {
           println!("found 1 or 2!");
        }
        // we can match against ranges
        3..=9 => {
           println!("found a number 3 to 9 inclusively");
        }
        // we can bind the matched number to a variable
        matched_num @ 10..=100 => {
           println!("found {} number between 10 to 100!", matched_num);
        }
        // this is the default match that must exist if not all cases are handled
        _ => {
           println!("found something else!");
        }
    }
}
```

Returning Values From loop

- loop can break to return a value.
- The returned value follows the keyword break.

```
fn main() {
    let mut x = 0;
    let v = loop {
        x += 1;
        if x == 13 {
            break "found the 13";
        }
    };
    println!("from loop: {}", v);
}
```

Returning Values From Block Expressions

- if, match, functions, and scope blocks all have a unique way of returning values in Rust.
- If the last statement in an if, match, function, or scope block is an expression without a ;, Rust will return it as a value from the block. This is a great way to create concise logic that returns a value that can be put into a new variable.
- Notice that it also allows an if statement to operate like a concise ternary expression.

```rust
fn example() -> i32 {
    let x = 42;
    // Rust's ternary expression
    let v = if x < 42 { -1 } else { 1 };
    println!("from if: {}", v);

    let food = "hamburger";
    let result = match food {
        "hotdog" => "is hotdog",
        // notice the braces are optional when its just a single return expression
        _ => "is not hotdog",
    };
    println!("identifying food: {}", result);

    let v = {
        // This scope block lets us get a result without polluting function scope
        let a = 1;
        let b = 2;
        a + b
    };
    println!("from block: {}", v);

    // The idiomatic way to return a value in rust from a function at the end
    v + 4
}
fn main() {
    println!("from function: {}", example());
}
```

Basic Data Structure Types

Structures

- A struct is a collection of fields, **not functions**.
- A *field* is simply a data value associated with a data structure. Its value can be of a primitive type or a data structure.

```
struct SeaCreature {
    // String is a struct
    animal_type: String,
    name: String,
    arms: i32,
    legs: i32,
    weapon: String,
}
```

Calling Methods (more details next week)

- Unlike **functions**, **methods** are functions associated with a specific data type.
- **static methods** – methods that belong to a type itself are called using the :: operator.
- **instance methods** – methods that belong to an instance of a type are called using the . operator.

```
fn main() {
    // Using a static method to create an instance of String
    let s = String::from("Hello world!");
    // Using a method on the instance
    println!("{} is {} characters long.", s, s.len());
}
```

Memory

- **data memory** - For data that is fixed in size and **static** (i.e. always available through life of program), such as the text in your program (e.g. "Hello World!").
- **stack memory** - For data that is declared as variables within a function.
- **heap memory** - For data that is created while the application is running. Data in this region may be added, moved, removed, resized, etc.

Creating Data In Memory

- When we **instantiate** a **struct** in our code our program creates the associated field data side by side in memory.

```rust
fn main() {
    // SeaCreature's data is on stack
    let ferris = SeaCreature {
        // String struct is also on stack,
        // but holds a reference to data on heap
        animal_type: String::from("crab"),
        name: String::from("Ferris"),
        arms: 2,
        legs: 4,
        weapon: String::from("claw"),
    };
    println!(
        "{} is a {}. They have {} arms, {} legs, and a {} weapon",
        ferris.name, ferris.animal_type, ferris.arms, ferris.legs, ferris.weapon
    );
}
```

Tuple-like Structs

```rust
struct Location(i32, i32);

fn main() {
    // This is still a struct on a stack
    let loc = Location(42, 32);
    println!("{}, {}", loc.0, loc.1);
}
```

Unit-like Structs

- Structs do not have to have any fields at all.

```rust
struct Marker;

fn main() {
    let _m = Marker;
}
```

Enumerations

- Enumerations allow you to create a new type that can have a value of several tagged elements using the enum keyword.
- match helps ensure exhaustive handling of all possible enum values making it a powerful tool in ensuring quality code.

```
enum Species {
    Crab,
    Octopus,
    Fish,
    Clam
}
fn main() {
    let ferris = SeaCreature {
        species: Species::Crab,
        name: String::from("Ferris"),
        arms: 2,
        legs: 4,
        weapon: String::from("claw"),
    };

    match ferris.species {
        Species::Crab => println!("{} is a crab",ferris.name),
        Species::Octopus => println!("{} is a octopus",ferris.name),
        Species::Fish => println!("{} is a fish",ferris.name),
        Species::Clam => println!("{} is a clam",ferris.name),
    }
}
```

```
struct SeaCreature {
    species: Species,
    name: String,
    arms: i32,
    legs: i32,
    weapon: String,
}
```

Generic Types

Generic Types

- Generic types allow us to partially define a `struct` or `enum`, enabling a compiler to create a fully defined version at compile-time based off our code usage.

```rust
// A partially defined struct type
struct BagOfHolding<T> {
    item: T,
}
fn main() {
    // Note: by using generic types here, we create compile-time created types.
    // Turbofish lets us be explicit.
    let i32_bag = BagOfHolding::<i32> { item: 42 };
    let bool_bag = BagOfHolding::<bool> { item: true };

    // Rust can infer types for generics too!
    let float_bag = BagOfHolding { item: 3.14 };

    // Note: never put a bag of holding in a bag of holding in real life
    let bag_in_bag = BagOfHolding {
        item: BagOfHolding { item: "boom!" },
    };

    println!(
        "{} {} {} {}",
        i32_bag.item, bool_bag.item, float_bag.item, bag_in_bag.item.item
    );
}
```

Vectors

- The struct Vec.
- The macro vec! .

```
fn main() {
    // We can be explicit with type
    let mut i32_vec = Vec::<i32>::new(); // turbofish <3
    i32_vec.push(1);
    i32_vec.push(2);
    i32_vec.push(3);

    // But look how clever Rust is about determining the type automatically
    let mut float_vec = Vec::new();
    float_vec.push(1.3);
    float_vec.push(2.3);
    float_vec.push(3.4);

    // That's a beautiful macro!
    let string_vec = vec![String::from("Hello"), String::from("World")];

    for word in string_vec.iter() {
        println!("{}", word);
    }
}
```

Generic Functions

```
// A generic function that returns the larger of two values
fn largest<T>(a: T, b: T) -> T
where
    T: PartialOrd,  // Constraint: T must be comparable
{
    if a > b { a } else { b }
}

fn main() {
    println!("{}", largest(3,6));
    println!("{}", largest(3.3,6.6));
    println!("{}", largest::<f32>(3.33,6.66));
}
```