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 2

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

Ownership & Borrowing Data

Ownership

- Instantiating a type and **binding** it to a variable name creates a memory resource that the Rust compiler will validate through its whole **lifetime**. The bound variable is called the resource's **owner**.

```rust
struct Foo {
    x: i32,
}

fn main() {
    // We instantiate structs and bind to variables
    // to create memory resources
    let foo = Foo { x: 42 };
    // foo is the owner
}
```

Scope-Based Resource Management

- Rust uses the end of scope as the place to deconstruct and deallocate a resource.
- The term for this deconstruction and deallocation is called a **drop**.

```
fn main() {
    let foo_a = Foo { x: 42 };
    let foo_b = Foo { x: 13 };

    println!("{}", foo_a.x);

    println!("{}", foo_b.x);
    // foo_b is dropped here
    // foo_a is dropped here
}
```

Moving Ownership

- When an owner is passed as an argument to a function, ownership is moved to the function parameter.
- After a **move** the variable in the original function can no longer be used.
- During a **move** the stack memory of the owners value is copied to the function call's parameter stack memory.

```
struct Foo {
    x: i32,
}

fn do_something(f: Foo) {
    println!("{}", f.x);
    // f is dropped here
}

fn main() {
    let foo1 = Foo { x: 42 };
    // foo1 is moved to foo2
    let foo2 = foo1;
    // println!("foo1.x={}", foo1.x); // error
    // foo1 is moved to do_something
    do_something(foo2);
    // foo2 can no longer be used
}
```

Returning Ownership

- Ownership can also be returned from a function.

```rust
struct Foo {
    x: i32,
}

fn do_something() -> Foo {
    Foo { x: 42 }
    // ownership is moved out
}

fn main() {
    let foo = do_something();
    // foo becomes the owner
    // foo is dropped because of end of function scope
}
```

Borrowing Ownership with References

- References allow us borrow access to a resource with the & operator.
- References are also dropped like other resources.

```rust
struct Foo {
    x: i32,
}

fn main() {
    let foo = Foo { x: 42 };
    let f = &foo;
    println!("{}", f.x);
    // f is dropped here
    println!("{}", foo.x);
    // foo is dropped here
}
```

Borrowing Mutable Ownership with References

- We can also borrow mutable access to a resource with the &mut operator.
- A resource owner cannot be moved or modified while mutably borrowed.

```rust
fn do_something(f: Foo) {
    println!("{}", f.x);
    // f is dropped here
}

fn main() {
    let mut foo = Foo { x: 42 };
    let f = &mut foo;

    // FAILURE: do_something(foo) would fail because
    // foo cannot be moved while mutably borrowed

    // FAILURE: foo.x = 13; would fail here because
    // foo is not modifiable while mutably borrowed

    f.x = 13;
    // f is dropped here because it's no longer used after this point

    println!("{}", foo.x);

    // this works now because all mutable references were dropped
    foo.x = 7;

    // move foo's ownership to a function
    do_something(foo);
}
```

Dereferencing

- Using &mut references, you can set the owner's value using the * operator.
- You can also get a copy of an owned value using the * operator (if the value can be copied - we will discuss copyable types in later chapters).

```rust
fn main() {
    let mut foo = 42;
    let f = &mut foo;
    let bar = *f; // get a copy of the owner's value
    *f = 13;      // set the reference's owner's value
    println!("{}", bar);
    println!("{}", foo);
}
```

Passing Around Borrowed Data

- Rust only allows there to be one mutable reference **or** multiple non-mutable references **but not both**.
- A reference must never **live longer** than its owner.

```rust
struct Foo {
    x: i32,
}
fn do_something(f: &mut Foo) {
    f.x += 1;
    // mutable reference f is dropped here
}
fn main() {
    let mut foo = Foo { x: 42 };
    do_something(&mut foo);
    // because all mutable references are dropped within
    // the function do_something, we can create another.
    do_something(&mut foo);
    // foo is dropped here
}
```

Text

Str

char

String

String Literals

- String literals are always Unicode.
- String literals type are &'static str
  - & meaning that it's referring to a place in memory, and it lacks a &mut meaning that the compiler will not allow modification
  - 'static meaning the string data will be available till the end of our program (it never drops)
  - str means that it points to a sequence of bytes that are always valid **utf-8**

```
fn main() {
    let a: &'static str = "hi 🦀";
    println!("{} {}", a, a.len());
}
```

Multi-line String Literals

- Rust strings are multiline by default.
- Use a \ at the end of a line if you don't want a line break.

```
fn main() {
    let haiku: &'static str = "
        I write, erase, rewrite
        Erase again, and then
        A poppy blooms.
        - Tachibana Hokushi";
    println!("{}", haiku);


    println!("hello \
    world") // notice that the spacing before w is ignored
}
```

What is utf-8

- What is Unicode?
- **utf-8** was introduced with a variable byte length of 1-4 bytes greatly increasing the range of possible characters.
- A downside of variable sized characters is that character lookup can no longer be done quickly (**O(1)** constant time) with a simple indexing (e.g. `my_text[3]` to get the 4th character).

String Slice

- A string slice is a reference to a sequence of bytes in memory that must always be valid utf-8.
- A string slice (a sub-slice) of a str slice, must also be valid utf-8.

```
fn main() {
    let a = "hi 🦀";
    println!("{}", a.len());
    let first_word = &a[0..2];
    let second_word = &a[3..7];
    // let half_crab = &a[3..5]; FAILS
    // Rust does not accept slices of invalid unicode characters
    println!("{} {}", first_word, second_word);
}
```

Chars

- With so much difficulty in working with Unicode, Rust offers a way to retrieve a sequence of utf-8 bytes as a vector of characters of type `char`.
- A `char` is always 4 bytes long (allowing for efficient lookup of individual characters).

```rust
fn main() {
    // collect the characters as a vector of char
    let chars = "hi 🦀".chars().collect::<Vec<char>>();
    println!("{}", chars.len()); // should be 4
    // since chars are 4 bytes we can convert to u32
    println!("{}", chars[3] as u32);
}
```

String

- A **String** is a struct that owns a sequence of utf-8 bytes in heap memory.
- Because its memory is on the heap, it can be extended, modified, etc. in ways string literals cannot.

```
fn main() {
    let mut helloworld = String::from("hello");
    helloworld.push_str(" world");
    helloworld = helloworld + "!";
    println!("{}", helloworld);
}
```

Text As Function Parameters

- String literals and strings are generally passed around as a string slice to functions. This offers a lot of flexibility for most scenarios where you don't actually have to pass ownership.

```
fn say_it_loud(msg:&str){
    println!("{}!!!",msg.to_string().to_uppercase());
}

fn main() {
    // say_it_loud can borrow &'static str as a &str
    say_it_loud("hello");
    // say_it_loud can also borrow String as a &str
    say_it_loud(&String::from("goodbye"));
}
```

Converting Strings

- Many types can be converted to a string using to_string.
- The generic function parse can be used to convert strings or string literals into a typed value. This function returns a Result because it could fail.

```
fn main() -> Result<(), std::num::ParseIntError> {
    let a = 42;
    let a_string = a.to_string();
    let b = a_string.parse::<i32>()?;
    println!("{} {}", a, b);
    Ok(())
}
```

Object Oriented Programming

Rust Is Not OOP

- Rust lacks inheritance of data and behavior in any meaningful way.
- Structs cannot inherit fields from a parent struct.
- Structs cannot inherit functions from a parent struct.
- That said, Rust implements many programming language features, so that you might not mind this lacking.

Encapsulation With Methods

- Rust supports the concept of an *object* that is a struct associated with some functions (also known as *methods*).
- The first parameter of any method must be a reference to the instance associated with the method call (e.g. instanceOfObj.foo()). Rust uses:
  - &self - Immutable reference to the instance.
  - &mut self - Mutable reference to the instance.

```rust
struct SeaCreature {
    noise: String,
}
impl SeaCreature {
    fn get_sound(&self) -> &str {
        &self.noise
    }
}
fn main() {
    let creature = SeaCreature {
        noise: String::from("blub"),
    };
    println!("{}", creature.get_sound());
}
```

Abstraction With Selective Exposure

- Rust can hide the inner workings of objects.
- By default fields and methods are accessible only to the module they belong to.
- The pub keyword exposes struct fields and methods outside of the module.

```
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}
```

Project Organization and Structure

Modules

- Every Rust program or library is a *crate*.
- Every crate is made of a hierarchy of *modules*.
- Every crate has a root module.
- A module can hold global variables, functions, structs, traits or even other modules!

Program or Library

- A program has a root module in a file called main.rs.
- A library has a root module in a file called lib.rs.

Referencing Other Modules and Crates

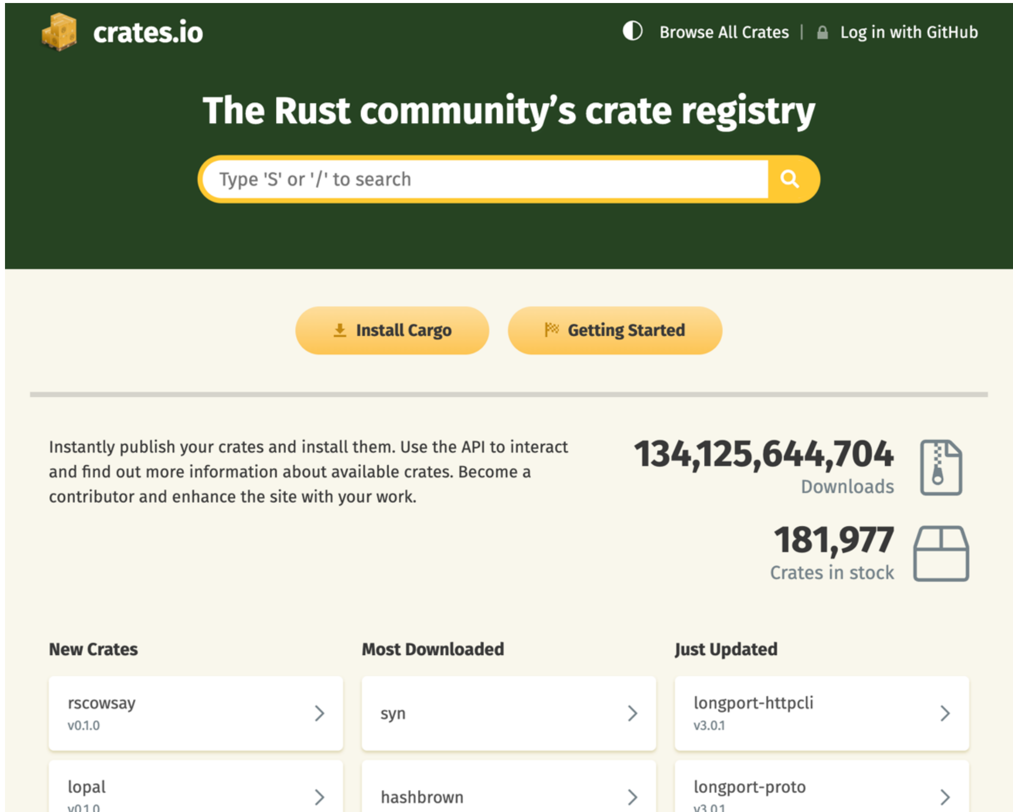
- Items in modules can be referenced with their full module path std::f64::consts::PI.
- **std** is the crate of the **standard library** of Rust which is full of useful data structures and functions for interacting with your operating system.

```rust
use std::f64::consts::PI;

fn main() {
    println!("Welcome to the playground!");
    println!("I would love a slice of {}!", PI);
}
```

```rust
use std::f64::consts::{PI,TAU}
```

https://crates.io/



Creating Modules

- There are two ways in Rust to declare a module. For example, a module foo can be represented as:
  - a file named foo.rs
  - a directory named foo with a file mod.rs inside

Module Hierarchy

- A module can depend on another one. In order to establish a relationship between a module and its sub-module, you must write in the parent module:

```
mod foo;
```

- The declaration above will look for a file named foo.rs or foo/mod.rs and will insert its contents inside a module named foo under this scope.

Inline Module

- A sub-module can be directly inlined within a module's code.
- One very common use for inline modules is creating unit tests. We create an inline module that only exists when Rust is used for testing!

```
// This macro removes this inline module when Rust
// is not in test mode.
#[cfg(test)]
mod tests {
    // Notice that we don't immediately get access to the
    // parent module. We must be explicit.
    use super::*;

    ... tests go here ...
}
```

Internal Module Referencing

- Rust has several keywords you can use in your `use` path to quickly get ahold of the module you want:
  - crate - the root module of your crate
  - super - the parent module of your current module
  - self - the current module

Prelude

- You might be wondering how we have access to Vec or Box everywhere without a use to import them. It is because of the module prelude in the standard library.
- Know that in the Rust standard library anything that is exported in std::prelude::* is automatically available to every part of Rust. That is the case for Vec and Box but others as well (Option, Copy, etc.).