# CS 241: Systems Programming
# Lecture 9. Basic Rust Programming

Fall 2025

Prof. Stephen Checkoway

# Mutability

Recall that variables are immutable by default

```
fn triangular_number(n: i32) -> i32 {
    let sum = 0;
    while n > 0 {
        sum += n; // <-- error cannot assign twice
        n -= 1;   // <-- error: cannot assign twice
    }
    sum
}
```

# Mutability

Add mut keyword to make variables/parameters mutable

```
fn triangular_number(mut n: i32) -> i32 {
    let mut sum = 0;
    while n > 0 {
        sum += n;
        n -= 1;
    }
    sum
}
```

# Constructing strings

We saw String::from() last time, we can also build a string piece by piece

```
let s = String::from("🌲 ⬅ is a tree");
println!("{s}");

let mut s = String::new();
s.push('🌲');
s.push_str(" ⬅ is a tree");
println!("{s}");
```

```
let mut s = String::new();
s.push('🌲');
s.push_str(" ⬅ is a tree");
```

If the mut is removed, what happens? (And why?)

A. Compile-time error

B. Run-time error

C. Compile-time warning

D. The code works correctly

E. Run-time crash

# Mutating operations require mut

Assigning a new value to a variable requires mut

Operations on data like strings and vectors that modify the data require mut

```
let mut v = Vec::new(); // Create an empty vector
v.push(5);  // Append 5 to the end of the vector
v.push(30); // Append 30 to the end of the vector

let mut s = String::new(); // Create an empty string
s.push('x'); // Append 'x' to the string
s.push('y'); // Append 'y' to the string
```

# Printing/String-construction macros

`print!()`: Prints the constructed string to stdout

`println!()`: Prints the constructed string and a new line to stdout

`format!()`: Returns the constructed string

```
let x = 10;
let y = true;
let z = "foo";

println!("x = {x}; y = {y}; z = {z}");
let s = format!("x + 5 = {}; !y = {}", x + 5, !y);
```

# Reading a string from stdin

```rust
use std::io;

fn main() {
    println!("Enter a line:");

    let mut line = String::new();
    io::stdin().read_line(&mut line).unwrap();

    print!("You entered: {line}");
}
```

# Reading a string from stdin

```
$ rustc strings.rs
$ ./strings
Enter a line:
Rust is great! 🦀🦀🦀
You entered: Rust is great! 🦀🦀🦀
```

# A closer examination

```
    let mut line = String::new();
```

Creates a new, mutable String

```
    io::stdin()
```

Returns a “handle” to `stdin`

```
        .read_line(&mut line)
```

Reads a line of text and appends it to `line`
The `&mut` is taking a reference to `line` and passing it to `read_line()`

```
        .unwrap();
```

`read_line()` can fail, if it does, `unwrap()` panics

# Causing a panic

```
$ echo -e '\xff'|./strings
Enter a line:
thread 'main' panicked at 'called `Result::unwrap()` on
an `Err` value: Error { kind: InvalidData, message:
"stream did not contain valid UTF-8" }', strings.rs:94:38
note: run with `RUST_BACKTRACE=1` environment variable to
display a backtrace
```

Strings in Rust must be valid UTF-8-encoded strings but the “string” I gave it was not because a single byte with value 255 (or 0xFF in hex) is not valid

# Panics

A panic is a controlled crash

When a run-time error is detected, the program panics, prints an error message, and exits

This prevents the program from operating in a bad state (the way that C would)

We can force a panic in several ways, including:

- panic!("Error message");
- assert!(false);
- assert_eq!(3, 5);
- Calling .unwrap() or .expect("Error message") on an Err

# A C program

```
$ ./small_primes
Enter the index of a small prime: 4
The small prime at index 4 is 11
```

**What happens if I enter a number that’s too large?**

```
$ ./small_primes
Enter the index of a small prime: 10
The small prime at index 10 is 129070014
```

???

```
$ ./small_primes
Enter the index of a small prime: -7
The small prime at index -7 is 83427584
```

???

# Normally, panicking is the last resort

We should strive to have panic-free code

Panics indicate something has gone wrong and we couldn’t recover

Instead, it’s better to indicate that an error occurred

In Python/Java, we’d throw an exception

In C, we’d return -1

**In Rust, we return a `Result<T, E>`**

# Result<T, E>

`Result<T, E>` is a parameterized type called an enum

It represents either

- Success in which case it holds a value of type T; or
- Error in which case it holds a value of type E

Examples

- `Ok(val)`
- `Ok("some success string")`
- `Err("some error message")`
- `Err(err)` — where err is some type of error like `ParseIntError`

# Many functions return a Result

```rust
fn complicated_function() -> Result<i32, String> {
    // Do some stuff
    if some_error_condition {
        return Err(String::from("Some error"));
    }
    // More stuff
    if other_error_condition {
        return Err(String::from("Some other error"));
    }
    Ok(return_value)
}
```

```rust
fn safe_div(x: i32, y: i32) -> Result<i32, String> {
    if y == 0 {
        return Err(format!("Cannot divide {x} by 0"));
    }
    todo!("What goes on this line?")
}
```

What should we replace the todo!() with to return x divided by y?

A. `return x / y;`

B. `x / y`

C. `return Ok(x / y)`

D. `Ok(x / y);`

E. `Ok(x / y)`

# Unwrapping a Result

Results have .unwrap() and .expect("msg") methods

- If the result is an Ok(val), it returns val
- If the result is an Err(err), it panics

```rust
let r: Result<i32, &str> = Ok(5);
let e: Result<i32, &str> = Err("oh no!");
println!("{}", r.unwrap());
println!("{}", e.unwrap());
```

```
Output:
5
thread 'main' panicked at 'called `Result::unwrap()` on
an `Err` value: "oh no!"', strings.rs:35:22
```

# Printing a Result

# Printing a Result

```
let r: Result<i32, &str> = Ok(5);
println!("{r}");
```

# Printing a Result

```rust
let r: Result<i32, &str> = Ok(5);
println!("{r}");
```

```
error[E0277]: `Result<i32, &str>` doesn't implement
`std::fmt::Display`
  --> strings.rs:33:15
   |
33 |     println!("{r}");
   |               ^^^ `Result<i32, &str>` cannot be
formatted with the default formatter
```

# Printing a Result

```rust
let r: Result<i32, &str> = Ok(5);
println!("{r:?}");
```

`{var}` means print var’s Display representation  
`{var:?}` means print var’s Debug representation

Basic types like `i32`, `bool`, `String` have `Display` representations

Most, more complicated types do not, by default

**Most types have a Debug representation**

# Printing a Result

```rust
let r: Result<i32, &str> = Ok(5);
let e: Result<i32, &str> = Err("oh noes!");
println!("{r:?}");
println!("{e:?}");
```

This prints:

```
Ok(5)
Err("oh noes!")
```

# Result is everywhere!

Most Rust functions that can fail return a Result

Rust will warn you if you call a function that returns a Result and you don’t do anything with it because it might have been an error you should not ignore

```
fn can_fail() -> Result<i32, String> {
    Err(String::from("Some error message"))
}

fn main() {
    can_fail();
}
```

Rust playground link

# For now

For now, we can generally .unwrap() our results

In the future we’ll want to handle them

We can use .is_ok() and .is_err() methods to determine which case it is but we’ll have a better option than

```
let r = can_fail();
if r.is_ok() {
    let val = r.unwrap();
    // …
}
```

# Converting from a string to another type

# Converting from a string to another type

```
let s = "42";
let t = "true";
```

# Converting from a string to another type

```
let s = "42";
let t = "true";

let i: i32 = s.parse().expect("Expected an i32");
let b: bool = t.parse().expect("Expected a bool");
println!("{i} {b}”);
```

# Converting from a string to another type

```
let s = "42";
let t = "true";

let i: i32 = s.parse().expect("Expected an i32");
let b: bool = t.parse().expect("Expected a bool");
println!("{i} {b}”);

let z: i32 = t.parse().expect("Expected an i32");
println!("{z}");
```

# Converting from a string to another type

```
let s = "42";
let t = "true";

let i: i32 = s.parse().expect("Expected an i32");
let b: bool = t.parse().expect("Expected a bool");
println!("{i} {b}”);

let z: i32 = t.parse().expect("Expected an i32");
println!("{z}");
```

```
$ ./strings
42 true
thread 'main' panicked at 'Expected an i32: ParseIntError
{ kind: InvalidDigit }', strings.rs:27:28
```

# How did parse() know what type to use?

```rust
let s = "42";
let t = "true";

let i: i32 = s.parse().expect("Expected an i32");
let b: bool = t.parse().expect("Expected a bool");
```

```rust
pub fn parse<F>(&self) -> Result<F, <F as FromStr>::Err>
```

parse() is parameterized by the type of result it is returning

Type inference lets us omit the type if the type of the result is known which it is above

Otherwise, you have to use `"42".parse::<i32>()`