COMP6991 🦀

▼ Week 1 — Basics

`6991 cargo new cname`

- Command → creates a new binary (runnable) application (directory) with a package manager
- Will create a Cargo.toml file which is the similar to JavaScript package.json
  - `cargo add some_library` is equivalent to `npm install some_library`
- Will create a src directory with an empty main.rs
- We can compile and run code in a binary application using `6991 cargo run`
  - `6991 cargo build` compiles it
  - `target/debug/cname` runs the compiled application called cname
- add `-- lib` option after new to create new library application

`6991 install cargo-expand` + `6991 cargo expand`

- First command is required to run the second one
- Will give an explanation/rundown of any macros or attributes used, and what the compiler does

Unit type → `let x: () = ()`

- It can only have the value `()`, so it is basically useless as a value, but useful as a type
- Useful when programming generically
- Functions (like main) that return `void` in C return `()` which we can omit
- Unlike `void`, `()` can be used as a variable, an argument, a result, etc.

Attributes → `#[]`

- Attributes are little tags we use to declare something to the compiler
- More on this later

Notes

- Rust does not use brackets () for if and while statements
- Enums and similar things are referred to as sum-types
- Structs and similar things are referred to as product-types

▼ Basic basics → Add function, Printing, Variables and Mutability

```
// constants need to be typed
const CONSTANT: i32 = 42;

// function takes 2 32 bit ints, and returns a 32 bit int
fn add(a: i32, b: i32) -> i32 {
    return a + b;

    // could also do
    res = a + b;
    res
}

fn printing(x: i32, y: i32, z: i32) {
    // we use {} as %flags
    println!("x = {}, y = {}, z = {}", x, y, z);
    // or we can put it in directly if we do not manipulate the value
    // i.e. {x} is fine, but {x + 1} is not
    println!("x = {x}, y = {y}, z = {z}");

    // we can number to specify which arg
    println!("x = {}, y = {}, x = {0}", x, y);
}

fn simpleVariable() {
    // this defines a constant we cannot change (immutable)
```

```
    let sth = 42;
    // unless we redeclare and overwrite it (shadowing)
    let sth = sth * 2;
    // we can also redeclare it with a different type
    let sth = 42.5;

    // note that shadowing may sometimes only temporarily override a na
    // if we use it within a scope that ends

    // mut keyword makes it mutable (changeable)
    let mut num = 42;

    // rust is relatively smart, so it infers the type (i32)
    // but, this is not dynamic typing → num = 'c' will not work
    // note that type conversions in Rust are typically explicit

    // let actually tries to match the LHS to some pattern on the structure
    // see Tuples and Struct Example

    // let actually tries to destructure the RHS if the LHS has a structure
    // that (pattern) matches the RHS
    // See tuples and structs examples
}

fn loops() {
    let mut x = 0;
    loop {
        if x > 100 {
            break;
        }
        println!("{x}");
        x += 1;
    }

    x = 0;
    while x < 101 {
        println!("{x}");
    }
```

```
    for i in 1..101 {
        println!("{i}");
    }

    // we also have python-esque loops
    let arr = [1, 2, 3, 4, 5];
    for elem in arr {
        println!("{elem}");
    }
}
```

▼ IO and Match Example → Guessing Game Example

```
// imports
use std::io;
use std::cmp::Ordering;

// rand is a third party dependency (package), so we have to cargo add
// in terminal before importing
use rand::Rng;

fn main() {
    // generates a random number between 1 and 100 inclusive
    let secret_number = rand::thread_rng().gen_range(1..=100);

    // mut means input is mutable → can change value
    let mut input = String::new();

    // reads a line from stdin into input
    io::stdin().read_line(&mut input).unwrap();
    // unwrap() allows us to assume it works, and avoid error handling

    loop {
        // reads a line from stdin into input
        io::stdin().read_line(&mut input).unwrap();
        // unwrap() allows us to assume it works, and avoid error handling
```

```
// dodgyish way of parsing string into int assuming it works
let guess = input.trim().parse::<i32>().unwrap();

// kind of like a switch statement on the result of cmp function
// which returns an Ordering enum (Less, Greater, Equal for i32)
match guess.cmp(&secret_number) {
    Ordering::Less => {
        println!("secret number is less than {guess}");
    }
    Ordering::Greater => {
        println!("secret number is greater than {guess}");
    }
    Ordering::Equal => {
        println!("You guessed it!");
        break;
    }
}
// note that match is exhaustive → have to cover all Ordering cases
// good for enforcing coverage

// match is also pattern matching, so we could have
// Ordering::Less | Ordering::Greater => { do sth }

// the above is equivalent to this, and will likely compile to the
// same machine code, so we just want the way that communicate
// idea better
if guess > secret_number {
    println!("secret number is less than {guess}");
} else if guess > secret_number {
    println!("secret number is greater than {guess}");
} else {
    println!("You guessed it!");
    break;
}

// but in Rust, we like to say what we are doing, instead of telling
// the computer, but it doesn't really matter
```

```
    }
}
```

▼ Option<T> Example

Option<T> is a special Enum in Rust that allow for an optional value

- enum can be Some(T) or None
- <T> is just generics like in Java
- We can create our own Option types if we want

```
// returns an Option with Some<String> if b is true,
// else returns an Option with None (nothing)
fn create(b: bool) -> Option<&'static str> {
    if b {
        Some("Hello There")
    } else {
        None
    }
}

fn main() {
    // method 1 -- check is_some() to see if there is a value
    let create_true = create(true);
    if create_true.is_some() {
            // prints "create(true) returned Hello There\n"
        println!("create(true) returned {}", create_true.unwrap());
    }

    // method 2 -- unwrap_or(default if None)
    println!(
        "create(false) returned {}",
        create(false).unwrap_or("<empty>")
    );

    // method 3
    match create_true {
            Some(myStr) => println!("create(true) returned {myStr}"),
```

```
            None => println!("create(false) returned <empty>");
    }

    // prints "None, Some("Hello There")\n"
    println!("{:?}, {:?}", create(false), create(true));

    // {:?} is called debug formatting, and lets us print the internal state
    // of an Option<T>
}

// some more practical example usecases of Option

// this function may return an i32 or None
fn string_to_int(myString: &str) -> Option<i32> {
        ...
}

// this function must take in an i32, and may take in a u32 or None
// and must return a String
fn int_to_string(int: i32, base: Option<u32>) -> String {
            ...
}
```

Option<T>

- Rust does not have NULL → forces us to use Option to represent optional/nullable values explicitly → if we have a function or sth that requires some T and T is not an Option, then we have to give it a value and can't give some form of NULL (bc it doesn't exist)
- There is no implicit unwrapping → will not unwrap an Option for you when you try to access it → we have to explicitly unwrap it before accessing anything
- if we try to access something in an Option, it will never try to unwrap it and crash the program in the case where it is None → we have to explicitly unwrap it to get values

- Makes Rust relatively ‘null-safe’ → prevents runtime null errors unless we explicitly unwrap and access a None option → forces us to handle None cases

This is a case where language design makes it easier to write robust programs

▼ Structs Example → Student

```
struct Student {
	zid: i32,
	name: String,
	wam: Optional<f64>,
}

fn main() {
	let mut student = Student {
		zid: String::from("Felix"),
		age: 100,
		wam: Some(100.0);
	}

	// note that we can't do the below if student is not mut
	student.age = 1;
	student.wam = None;

	// since the LHS variables has a structure that matches the struct
	// let will 'smartly' follow the structure as a pattern
	let Student { x, y, z } = student;
	// prints "Felix, 1, None\n"
	println!("{x}, {y}, {z}");

}
```

▼ Enum and Matching → automobiles

```
// we use enums for the match operator
```

```
// basic enum
enum AutomobilesBasic {
	Car,
	Motorbike,
	Train,
	Bus,
}

enum WheelDrive {
	FWD,
	FOURWD,
	AWD,
	RWD,
}

// we can also attach data to enum elements
enum Automobile {
	Car { make: String, model: String, drive: Option<WheelDrive> },
	Motorbike { make: String, model: String, horsepower: u32 },
	Train { num_carriages: u32, capacity: u64 },
	Bus { capacity: u32 },
	Irrelevant1,
	Irrelevant2,
	Irrelevant3,
}

fn printAutomobile(automobile: Automobile) {
	match automobile {
		Autombile::Car { make, model, drive } => {
			println!("Car: {make}, {model}, {:?}", drive);
		}
		Automobile::Motorbike { make, model, horsepower } => {
			println!("Motorbike: {make}, {model}, {horsepower}");
		}
		Automobile::Train { num_carriages, capacity } => {
			println!("Train: {num_carriages}, {capacity}");
		}
		Automobile::Bus { capacity } => {
```

```
                println!("Bus: {capacity}");
            }
            // _ matches anything in Rust, so it is often used for else/defa
            // in match
            _ => {
                println!("we don't care about this Automobile");
            }
        }

        // since match is exhaustive, it ensures we cover all cases,
        // which is useful when we modify or extend an enum if we didn't
        // have the _ case
}

fn stupid_matching(x: u32) {
        // in Rust, we can treat most terms as patterns
        match x {
            0 => {
                println!("0");
            }
            1 => {
                println!("1");
            }
            2 => {
                println!("2");
            }
            ...
            _ => {
                println!("why not just print the number {x}?");
            }
        }
}
```

▼ Tuples Example

```
fn createTuple(x: i32, y: u64, b: bool, c: char) -> (i32, u64, bool, char) {
    // tuples can be created with different types
    let tup1 = (x, y, b, c);
```

```
    // we can type the tuple
    let tup2: (i32, u64, bool, char) = (x, y, b, c);

    tup
}

fn printTupleThriceish(tup: (i32, u64, bool, char)) {
    println!("({}, {}, {}, {})", tup.0, tup.1, tup.2, tup.3);

    // (x, y, b, c) has a structure that matches tup, so let fills it in
    // by matching x to the i32, y to the u64 and so on
    let (x, y, b, c) = tup;
    println!("({x}, {y}, {b}, {c})");

    // .. matches anything else, but we ignore what it matches
    let (x, y, ..) = tup;
    println!("({x}, {y}");

    // _ matches any one element, but we ignore what it matches
    let (x, _, _, c) = tup;
    println!("{x}, {c}");
}
```

Notes

- Structs should be used over tuples unless what the tuple represents is extremely obvious

▼ Fixed Size Array Example

```
fn main() {
        // create an array of 3 i32s
        let arr1: [i32; 3] = [1, 2, 3];

        // create and initialise an array of 0s
        let arr2: [i32; 20] = [0; 20];

        // iterate too far and go over the array
        // prints 1, 2, 3, NONE!!!
```

```
    for i in 1..arr1.len() + 1 {
        // safe way of indexing
        match arr1.get(i) {
            Some(val) ⇒ println!("arr1[{i}] = {val}"),
            None ⇒ println!("NONE!!!");
        }
    }
}
```

▼ Result<T>

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

- Functions often return a Result<T, E> instead of an Option<T>, since this allows us to put specific error messages or types if we want
- Or, we can just use unit and return Err(()) to signify that there was some error, which makes it functionally equivalent to an option (isomorphic)

```
fn takes_result(x: Result<i32, String>) {
    match x {
        Ok(val) ⇒ {
            println!("result is {val}");
        }
        Err(error_msg) ⇒ {
            println!("ERROR: {error_msg}");
        }
    }
}
```

Rust is an expression-oriented language

- C is statement-oriented
- Statements are lines that do not evaluate to a value → `int x = 4;` is a statement
- Expressions evaluate to a value
- Rust has most things as expressions than statements → makes code more flexible → e.g. if-else blocks are statements that evaluate to one of the cases or () if none
- We can turn most things into expressions by wrapping them in braces → `let x = { let y = 42 };` will set y to `42` and x to `()`
- We turn expressions into statements by ending the expression with ';' → `let x = 42;` is a statement, but `x = 42` is an expression → note that statements sometimes evaluate to unit

▼ Expressions in Rust

```
// equivalent to a function that explicitly psets x to 42
// if flag is true, or -42 otherwise
fn foo1(flag1: bool, flag2: bool) -> Option<i32> {
    // we call them 'if expressions' and not 'if statements' in Rust
    // and they are similar to ternaries
    let x = if flag1 {
        println!("WAS TRUEEE");
        // nested like this is fine cause this inner if is also an expressi
        if flag2 {
            // early return will just return immediately and the expres
            // is not evaluated
            return None;
        } else {
            Some<42>
        }
    } else {
        println!("false...");
        Some<-42>
    };
```

```
        // since we have a let, we need to terminate the expression at the
        // but we can still have statements inside each case as long as we
        // an expression of the same type
        println!("{:?}", x);
        x
        // functions can also be treated as expressions, where the last exp

        // note that the match operator can also be treated as an expressio
}

// loops are also expressions
fn foo2(num: i32) -> i32 {
        let mut i = 0;
        let mut sum = 0;

        let x = loop {
                if i > num {
                        break sum;
                }
                sum += i;
        };
}
```

▼ Week 1 — Ownership

In Rust, memory is managed through a set of ownership rules that are checked by the compiler

- Each value in Rust has an owner, and there can only be one owner at any given time
- When the owner of a value goes out of the scope, the value will be dropped
- Note that we can also explicitly drop through ‘drop(thing)’
- Makes code memory safe, and thread safe → prevents memory issues like use after free or double free, and thread issues like a data races

Copy Types are types that are exempt from the Ownership rules

- All the basic scalar types (i32, u32, char, bool) are Copy types, as are tuples that only contain Copy types
  - Structs and enums consisting only of Copy types can be made copy types using the attribute `#[derive(Copy, Clone)]` above the struct/enum definition
  - Fixed-size primitive arrays where all T is a Copy type are also Copy types
- Copy types are implicitly copied and not transferred upon assignment
- Copy is also an auto-trait → Copy types with automatically have this behaviour implemented by the compiler

Note that ownership cascades for structs and enums

- When we transfer ownership of something from x to y, everything owned by x becomes owned by y, including anything owned by something within the something

▼ Ownership

```
// we need the derive to make this struct copyable
// still requires all elems/fields to be Copy types to work
#[derive(Copy, Clone)]
struct MyStruct {
    num1: i32,
    num2: i32,
}

fn simpleOwnership() {
    // all the basic types are 'Copy' type, which are implicitly copied and
    // exempt from the Ownership rules
    let x = 5;
    let y = x;

    // y is now a copy of x
```

```
    // tuples that consists only of Copy types are also Copy types
    // e.g. t2 will be a copy
    let t1 = ('a', 1, true);
    let t2 = t1;

    // but String is not a Copy type, so t3 will be killed
    let t3 = ('a', 1, String::from("Hello"));
    let t4 = t3;

    let s1 = MyStruct {
        num1: 42,
        num2: 100,
    };
    // will create a copy, since we have derive(Copy, Clone)
    let s2 = s1;

    // same stuff as structs for enums
}
// after the above brace '}', the scope ends, and both x and y are dropp

// non-Copy types must obey the Ownership rules
fn notSoSimpleOwnership() {
    let x = String::from("Hello");
    // x is now some data with a length, capacity and a pointer to the dat

    // y is a copy of x, so same length, capacity and pointer
    // y points to the same data as x
    let y = x;

    // but Rust only allows one owner so x is dead (no longer a valid vari
    // println!("{x}") will not compile
    // but the below will
    println!("{y}");

    // Rust does not implicitly make a deep copy of memory on the heap
}
// the scope ends, and x is dead, so only y is dropped, preventing
```

```
// a double-free

fn clones() {
    let x = String::from("Hello");
    // this makes a deep copy, and does not kill x
    // as y is the owner of a different lot of memory on the heap
    let y = x.clone();

    // actually, clone() is implicitly called for Copy types
    let x = 42;
    // this line is essentially let y = x.clone()
    let y = x;
}
// both x and y get dropped

fn takes_string(s: String) {
    println!("{s}");
}

fn takes_and_returns(s: String) {
    println!("{s}");
    s
}

fn functionOwnershipTransfer() {
    let s = String::from("GoodBye");
    // ownership is transferred to the s in takes_string, and our s is killed
    takes_string(s);

    // println!("{s}") will cause a compile error

    // if we really wanted to, we could transfer ownership back
    let s = String::from("Howdy");
    let s = takes_and_returns(s);
    println!("{s}");
}
// s gets dropped
```

▼ Cascading Ownership

```
struct Person {
	name: String,
	ransom: i128,
}

fn main() {
	let my_str = String::from("Felix");

	// ownership of my_str transfers to Person struct which is owned b
	let x = Person {
		name: my_str,
		ransom: -100,
	}

	// println!("{my_str}") will not compile

	// ownership of everything including my_str transfers to y
	let y = x;

	// println!("{x.name}") will not compile

	// y.name will be dropped
	let s = y.name;

	// this still works since ownership was not/cannot be transfered (C
	println!("{}", y.ransom);

	// but println!("{}", y.name); will not compile
}
```

▼ Kill copy type

```
// if we want to make the ownership rules apply to a copy type
// we just wrap it in a struct
struct Owned_i32 {
	val: i32,
```

```
}

fn main() {
	let x = Owned_i32 {
		val: 42,
	};

	// ownership transferred to y, x is killed
	let y = x;
	println!("{:?}", y);
	// prinln!("{:?}", x) will cause a compile error
}
```

▼ Week 2 — Collections & Iterators

Common collections in Rust

- Array
- Vec
- VecDeque
- String
- LinkedList
- HashMap
- BTreeMap
- HashSet
- BTreeSet
- BinaryHeap

Note: most collections live on the stack, but have a pointer to some memory

- E.g. A Vec is really just a struct stored on the stack with a ptr to memory on the Heap, a usize len and a usize capacity

▼ Vec

Vecs (vectors) provide us with dynamic arrays → resizeable

- Allows us to work with containers without knowing the exact size, and to work with containers of various sizes
- Arrays are allocated on the stack, which gives them some performance benefits over vecs, but at the cost stack memory,

which is much more limited than heap memory

- a vec has 3 parts: ptr to elements, length, and capacity
  - If length < capacity when push is called, it just uses the extra space and increments length
  - If length == capacity and push is called, it has to realloc (typically doubles the current capacity) before adding the new element

```
fn double_nums1(vec: Vec<i32>) -> Vec<i32> {
    // declares an empty vec -> could also do Vec::new()
    let mut res = vec!();

    for elem in vec {
        res.push(elem * 2);
    }

    vec
}

fn double_nums2(vec: Vec<i32>) -> Vec<i32> {
    // declares an empty vec with capacity >= to the length of vec
    let mut res = Vec::with_capacity(vec.len());
    // faster since we only need 1 allocation of memory and no reallocs
    // note that Rust may choose to allocate more than specified as a buf

    println!("res has {res.capacity()} capacity");

    for elem in vec {
        res.push(elem * 2);
    }

    vec
}

fn main() {
    // creates a mutable fixed size i32 array
```

```
    let mut arr = [0, 2, 3];
    arr[0] = 1;

    // creates a mutable Vec<i32>
    let mut my_vec = vec![0, 2, 3];
    my_vec[0] = 1;

    my_vec.push(4);
    my_vec.push(5);

    // prints "[1, 2, 3, 4, 5]
    println!("{my_vec:?}");

    // push needs to realloc to expand its size, which can be costly
    // so we can also create an empty vec with an initial size
    let mut my_vec2 = Vec::with_capacity(100);
}
```

▼ Iterator

Iterator trait

- Like in Java, most collections in Rust implement the `iter()` method that returns an `Iterator<T>`, where `T` is the type of the items in the container
- `Iterator<T>` has one primary method `next()` which returns an `Option<T>` with the next item
- Rust ownership rules mean that we can't directly modify the collection an iterator is created on while the iterator exists or hasn't finished iterating

▼ Iterator creation methods

```
into_iter(Self)
    - ownership of self is transferred to the iterator
    - collection iterator is created from is killed
    - next() returns individual Ts that we can take ownership of
```

```
fn using_into_iter() {
	let xs = vec![1, 2, 3, 4, 5];

	// ownership of xs transferred to iterator
	for item in xs.into_iter() {
		println!("{item:?}");
	}

	// println!("{xs:?"}) will not compile
}
```

iter(&Self)
- uses a shared borrow of self so no ownership transfer
- read-only iterator → cannot change the T
- next() returns shared borrows (&Ts)

```
fn using_iter() {
	let xs = vec![1, 2, 3, 4, 5];

	// shared borrow of xs given to iterator
	for item in xs.iter() {
		println!("{item:?}");
	}

	println!("{xs:?"});
}
```

iter_mut(&mut Self)
- uses an exclusive borrow of self so no ownership transfer
- Cannot have any other borrows of self in existence
- read-write → can change the T
- next() returns exclusive borrows → (&mut Ts)

```
fn using_into_iter() {
	let mut xs = vec![1, 2, 3, 4, 5];

	// ownership of xs transferred to iterator
```

```
    for item in xs.iter_mut() {
        *item += 1;
        println!("{item:?}");
    }

    // prints [2, 3, 4, 5, 6] since iter_mut changed it
    println!("{xs:?}");
}
```

▼ Example

```
fn main() {
    let my_vec = vec![1, 2, 3];

    // iterator must be mut to be able to call next()
    let mut iter = my_vec.iter();

    println!("{:?}", iter.next());  // Some(1)
    println!("{:?}", iter.next());  // Some(2)
    println!("{:?}", iter.next());  // Some(3)
    println!("{:?}", iter.next());  // None
    println!("{:?}", iter.next());  // None

    let my_vec = vec![1, 2, 3];
    let mut iter = my_vec.iter();
    loop {
        match iter.next() {
            Some(elem) ⇒ {
                println!("{elem}")
            }
            None ⇒ {
                println!("No more elements!");
                break;
            }
        }
    }

    let my_vec = vec![1, 2, 3];
```

```
    let mut iter = my_vec.iter();
    for elem in iter {
        println!("{elem}");
    }

    // iterators also provide some other handy methods
    let my_vec = vec![1, 2, 3];
    // maps it from an i32 iterator to a string iterator
    let mut iter = my_vec.into_iter().map(|x| format!("{x}"));

    // we call the '|x| sth' a closure, which is equivalent to a lambd
}

fn convert_to_string_iter(my_vec: Vec<i32>) {
    // maps it from an i32 iterator to a string iterator
    let mut iter = my_vec.into_iter().map(|x| format!("{x}"));
    println!("{:?}", iter);

    // note that '|x| sth' is called a closure, which is equivalent to la
}

fn filter_even_iter(my_vec: Vec<i32>) {
    let mut iter = my_vec.into_iter().map(|x| x % 2 == 0);
    println!("{:?}", iter);
}

fn get_avg(my_vec: Vec<i32>) -> Option<f64> {
    if my_vec.is_empty() {
        None;
    } else {
        let len = my_vec.len();
        let sum = my_vec.into_iter().sum();
        Some((sum as f64) / (len as f64))
      }
}

fn longest_run(vec1: Vec<i32>, vec2: Vec<i32>) {
    // zip merges two iterators into an iterator of tuples that pair th
```

```
        let (best, current) = vec1.iter()
            .zip(vec2.iter())
            .map(|(x, y)| x == y);
            // .map(|tup| tup.0 == tup.1) is equivalent to the above line
            // fold is similar to reduce in JavaScript and an aggregate
            .fold((0, 0), |(best, curr), elem|
                if elem {
                    let curr = curr + 1;
                    (best.max(curr), curr)
                } else {
                    (best, 0)
                }
            );
        // iter is an Iterator<bool> where an elem is true if a1[i] == a2[i]

        best
}
```

▼ String

Strings in Rust are collections of chars

- Chars in Rust represent a Unicode scalar value (code point) → ASCII, emojis, etc.
- Strings in Rust support UTF-8 encoding
- We create iterators over strings using `.chars()`

▼ HashMaps

```
fn mode(vec: Vec<i32>) -> Option<i32> {
    let mut map = HashMap::new();

    for elem in vec {
        // get the entry elem maps to
        // if it exists, increment by 1
        // otherwise insert new entry (elem, 1)
        map.entry(elem)
            .and_modify(|x| {
                *x += 1;
```

```
            })
            .or_insert(1);

        // note that we dereference x in the above
    }

    // creates iterator of tuples that are the key-value pairs
    let (num, _) = map.into_iter().max_by_key(|(_, freq)| *freq)?;

    // ? means we return None if its None

    Some(num)
}
```

▼ Week 2 — Error Handling

Idiomatically (typically) errors are handled through Sentinel values, exceptions, first-class types, and more

- We have to balance error handling against practibility → if a language enforces very defensive programming, it may be very safe, but also very tedious/impractical to implement

▼ Sentinel Values

Sentinel values are values used in an application as a flag for a termination condition

- Commonly found in C, Go and Pascal
- In C we have `NULL` and `'\0'`
  - `if (ptr == NULL) { return NULL }`
  - `if (c == '\0') { return res }`
- We also consider error codes to be sentinel values in that they signify what went wrong and why a function terminated early or didn't succeed
- Very rarely used in Rust, since we have `Option<T>` and `Result<T, E>` types, and do not have a `NULL` equivalent → see example below

```
struct Student {
	zid: u32,
	name: String,
	wam: Option<f64>,
}

// the i32 in our tuple is our sentinel value
fn get_student(bool exists) -> (Student, i32) {
	if !exists {
		// -1 signifies failed, but also how to represent no student?
		// a dummy Student struct???
		(..., -1)
	} else {
		(
			Student {
			    zid: 15,
			    name: "Felix",
			    wam: None,
			},
			0,
		)
	}

	// but, why wouldn't we just use an Option or Result instead of havi
	// a tuple with a value that has to somehow represent empty, and a
}
```

Pros:

- Easy to set up and customise error codes
- If we are certain there will never be an error, there’s no overhead cost

Cons:

- Very easy to forget to handle an error

▼ Unchecked Exceptions

Unchecked Exceptions are those that are not checked by the compiler, and happen at runtime

- Usually handled through a try-catch (try-except in python) block that adds extra control flow
- As the try-catch applies to the function call stack after it, there may be large gaps between where the exception was thrown, and where it was caught
- Common in a lot of languages → Java, C++. C#, Python, Swift, etc.
- Rust also has this try-catch exception model

▼ Python try-except example

```
def double_user_num():
    while True:
        try:
            num = int(input("Please enter a number: "))
            return num * 2
        except ValueError as err:
            print("That's not a number!")
            print("{err}")
        except RunTimeError:
            print("damn")
        except TypeError:
            print("Wrong type???")
        except:
            print("idk sth else happened")
```

Pros:

- Can help to complete program executions (not crash)
- Can help find what failed and handle it elegantly-ish

Cons:

- Requires reading documentation to see what exceptions library functions throw, and then handling them
- Overlooked exceptions will cause crashes at runtime
- Less control as exceptions can interrupt execution at any time
- Almost anything can throw an unchecked exception, so its impractical to try-catch everything

Note that Rust does have a `catch_unwind` function that can 'catch' some panics

- **Panics are not exceptions**
- Panics can either unwind or abort → catch_unwind only catches the unwinding panics
- Dodgy atm, don't use?

▼ Checked Exceptions

Checked Exceptions are those that must be expicitly handled

- Enforced by compiler → kinda only in Java
- We usually only have checked exceptions for things we can deal with → losing a database connection, file fails to open or invalid permissions, etc.
- We need to declare that a function throws a checked exception, and any functions that are declared to do so must be wrapped in a try-catch

Pros:

- Forces us to handle exceptions → cover error cases

Cons:

- Can be very messy/tedious

- May expose implementation details
- Other users of a function may be forced to handle an exception which their application may never cause
- May often clash with other features in the language

First-Class (Sum) Types

Rust's approach to error handling is to use Sum types (i.e. enums), and add extra tooling to help with it

- `Option<T>` and `Result<T, E>` are our main tools
- We can also create our own enums for error handling

Read number example

```
// custom error enum
enum GetInputError {
    InputFailed,
    InputWasNotNumber {
        user_input: String,
    },
    InputEmpty,
    ...
}

fn get_input() -> Result<i32, GetInputError> {
    let mut input = String::new();
    let result = std::io::stdin().read_line(&mut input);

    // if we weren't able to read a result return an Err
    match result {
        Ok(_) => {}
        Err(_) => {
            return Err(GetInputError::InputFailed);
        }
    }

    // if we weren't able to parse a result return Err
```

```
        let num: Result<i32, _> = input.trim().parse();
        match num {
            Ok(value) => Ok(value),
            Err(err) => Err(GetInputError::InputWasNotNumber {
                user_input: input,
            }),
        }
}

fn helper() -> i32 {
        loop {
            match get_input() {
                Ok(value) => {
                    return value
                }
                Err(err) => {
                    match err {
                        // if we can't read input, we don't want to ke
                        // trying, so we abort with panic which will g
                        // a function call trace then crash
                        GetInputError::InputFailed | GetInputError::Ir
                            panic!("Could not read input!");
                        }
                        GetInputError::InputWasNotNumber { input }
                            println!("'{input}' is not a number. Try a
                        }
                    }
                }
            }
        }
}
```

▼ Read Number example with fancy Rust stuff

```
// custom error enum
enum GetInputError {
    InputFailed,
    InputWasNotNumber {
```

```
        user_input: String,
    },
    InputEmpty,
    ...
}

// impl → defines an implementation of From for GetInputError
impl From<std::io::Error> for GetInputError {
    fn from(_error: std::io::Error) → Self {
        GetInputError::InputFailed
    }
}

// From is a trait used to convert one type into another
// in the above case, this implementation allows
// GetInputError::InputFailed errors to be created From stdio errors
// automatically when using '?' operator

// we could do the same for ParseIntError, but there's no way for us
// to get the input string, so we will use .map_err()
// impl From<ParseIntError> for GetInputError {
//     fn from(_error: ParseIntError) → Self {
//         GetInputError::InputWasNotNumber
//     }
// }

fn get_input() → Result<i32, GetInputError> {
    let mut input = String::new();
    // the question mark at the end of the line means
    // if error, return error, else continue
    std::io::stdin().read_line(&mut input)?;

    // map_err to convert to InputWasNotNumber
    let num: i32 = input.trim().parse()
        .map_err(|_| GetInputError::InputWasNotNumber{
            user_input: input
        })?;
    Ok(num)
```

```
}
// look at the '? operator explanation' for more on '?'

fn helper() -> i32 {
    loop {
        match get_input() {
            Ok(value) => {
                return value
            }
            Err(err) => {
                match err {
                    // if we can't read input, we don't want to ke
                    // trying, so we abort with panic which will g
                    // a function call trace then crash
                    GetInputError::InputFailed | GetInputError::Ir
                        panic!("Could not read input!");
                    }
                    GetInputError::InputWasNotNumber { input }
                        println!("'{input}' is not a number. Try a
                    }
                }
            }
        }
    }
}
```

▼ ? operator explanation

```
fn foo() -> Result<..., ...> {
    ...
}

fn main() {
    let x = match foo() {
        Ok(val) => val,
        Err(err) => return Err(err.into()),
    }
```

```
        // the above is equivalent to 'let x = foo()?'
        // the err.into() line is what converts the Error type and value
        // if the current error type does not match the expected type
        // by using the From conversion definition for the expected type

        // won't compile if the expected type does not have From implemented
        // for the type we are converting from
}
```

Panic!

- It is appropriate to use panic for unrecoverable errors
- In the example above, if we can't read from stdin, there's no way to fix that at runtime, so we panic, but if we can't parse, we can recover by trying again with new input
- `RUST_BACKTRACE=1 cargo run` → gives a nicer function call trace if your program crashes from a panic

Notes:

- `.map_err(|e| do_sth())` is our friend

▼ Week 3 — Borrowing

Ownership is great for ensuring memory and thread safe code, but can also be a pain to work with

- May have to constantly transfer ownership back and forth for function calls and use tuples as return types if we wanted some other return value or to transfer multiple ownerships back

Rust allows borrowing through the concept of 'shared xor mutable'

- A term can be shared, or it can be mutable but not both
  - Shared → many things can read a term at any given time → read-only
  - Mutable → only one thing can access and mutate a term

- HOWEVER, a term can be both shared-borrowed and exclusive-borrowed over its lifetime
  - Cannot exclusive borrow is something else is already borrowing → guarantees that shared borrows will always read the current version
  - Can share-borrow as much as you want as long as an exclusive borrow doesn't exist
- Borrows are very cheap to pass to functions → basically a pointer with some special Rust stuff
  - Share borrows are read-only pointers
  - Exclusive borrows can mutate data without taking ownerships
- Note that calling `into_iter()` or `iter()` on a borrowed term will not kill the original term

▼ Borrowing Explanation

```
// T is just the basic Owned type → has all access since its the owner
// &T is a shared borrow (immutable reference) → read only
// &mut T is an exclusive borrow (mutable reference) → read, write but

// Note that shared borrows are Copy types
fn shared(s: String) {
    // create a shared borrow
    let x = &s;
    // copies x to y, so both are valid
    let y = x;

    // we can also create as many share borrows as we want
    let a = &s;
    let b = &s;
    ...
    let z = &s;

    // but we cannot create exclusive while we have shared borrows
    // let excl = &mut s will not compile
```

```
        // both will print s
        println!("{}", *x);
        println!("{}", *y);

        // Rust actually dereferences for us in most cases, so we could jus
        println!("{}", x);
        println!("{}", y);
}

// but exclussive borrows are not Copy types
fn exclusive(s: String) {
        // create an exclusive borrow
        let x = &mut s;
        // moves x to y, so x is no longer valid
        let y = x;

        // we can mutate s through y
        y.push_str(" HOWDY!");

        // we cannot create any shared or exclusvie borrows while an excl
        // borrow exists → i.e. the below will not compile
        // let a = &mut s;
        // let b = &mut s;

        // println!("{}", x) will not compile
        println!("{}", y);
}

// we cannot change borrow types
fn change(s: &String) {
        // let excl_borrow = &mut *s will not compile
}
```

▼ Shared Borrowing Example → string num chars

```
// we are just reading a string's bytes, we can just use a shared borrow
```

```
fn string_num_chars(s: &String) → usize {
    // note that we can't do s.len() since that returns the number of by
    // and non-ascii chars may take more than 1 byte

    // s.push_str("mutating!") will not compile

    // the chars() method takes in &self
    s.chars().count()
}

fn main() {
    let s = String::from("ab");

    // we need to specify the borrow type if we are lending it
    let res = string_num_chars(&s);

    // s not killed
    println!("{s}");
    // will work since string_num_chars terminated and borrow was re
    s.push_str("cdef");
}
```

▼ Exclusive Borrowing Example → emphasise string

```
fn emphasise(s: &mut String) {
    // check documentation when looking at methods
    // make_ascii_uppercase() takes a &mut Self, which we have, while
    // to_uppercase() and some others will not
    s.make_ascii_uppercase();

    // technically, s.make_ascii_uppercase uses a reborrow → emphas
    // lends its borrow then gets it back

    // let x = &s and let y = &mut s will not compile since we already ha
    // an exclusive borrow in existence

    // push_str also takes an &mut_self
    s.push_str("!!!");
```

```
}

fn main() {
    // only mutable terms can be exclusive borrowed
    let mut s = "howdy";

    emphasise(&mut s);

    // prints "HOWDY!!!"
    println!("{s}");
    s.push_str("???");
}
```

▼ Borrowing with Structs

```
struct Person {
    name: String,
    height: u32,
}

fn main() {
    let p1 = Person { name: "Felix", height: 178 };
    let p2 = Person { name: "Ribbit", height: 5 };

    let bp1 = &p1;

    // fine cause height: u32 is Copy type
    p2.height = bp1.height;

    // this will not compile since we can't transfer ownership through a
    // p2.name = bp1.name;

    // this will not work either, since Person requires a String not an &S
    // p2.name = &bp1.name;
}
```

Dangling References are when we return a pointer to something that has been dropped as the scope that something exists in has closed

▼ Dangling Reference Example

```
struct ListNode {

}

fn create_ll() -> &ListNode {
	let list = LinkedList { };

	&list
}
// after this, list is dropped from the stack since the scope ends, and sc
// we have a pointer to something that has been dropped.
// we call this a dangling reference

// fortunately, Rust doesn't allow this and the above won't actually com

// we do, however, have &`static which is a pointer to something that wi
// exist for the entire program's lifetime -> a memory leak that never ge
// cleaned

// good for global variables

fn skull() -> &`static ListNode {
	let list = ListNode { };

	// creates a Box (ptr to allocated memory on heap)
	// for the list and returns a static pointer to it
	Box::leak(Box::new(list))
}
```

Slices are a borrow reference for a contiguous subsection of a Collection/container

- Borrow so no ownership -> can be shared of exclusive borrow

- Unifies how we want to have a shared borrow for arrays, vecs, etc. since they are all quite similar if we want a read-only reference → borrow them to create a slice and then use that
- Slices are not sized → practically impossible or useless to use by themselves since the size of the data must be known to be able to place it on the stack/heap → won't compile
- Need to use references to slices → compiler will create a 'fat pointer' that has the reference to the data of the slice and its length
- Fat pointers have the size of 2 pointers
- Rust compiler can usually change a borrow of anything that can be viewed/converted to a slice into a slice
  - Arrays
  - Vecs
  - Other slices
- **NOTE**: slices can be created as exclusive references where we can modify the data, BUT we cannot expand or shrink the data

String slices → `&str`

- The `String` equivalent of a slice is `&str` → pointer to a String and a length
- Rust compiler can usually change `&String` to `&str` for you, and an `&[char]` into an `&[u8]`
- String literals have type `&str`

▼ Slices → explanation

```
// this will not compile since the compiler doesn't know the length/size
// fn bad_take_slice(s: [i32[) {
//      ...
// }

fn good_take_slice(s: &[i32]) {
        println!("{:?}", s);
```

```
}

fn create_slice() {
    let arr = [1, 2, 3, 4, 5];

    // create a mutable slice from arr that takes indexes [1, 4)
    let s = &mut arr[1..4];
    // s has a reference to [2, 3, 4] and a length of 3

    // through s, we can change arr, but we cannot expand or shrink it
    s[0] = 5;

    // prints [1, 5, 3, 4, 5], since the above line changed it
    println!("{:?}", arr);
}

fn main() {
    create_slice();

    // the below will all print the same thing
    let my_arr = [1, 2, 3, 4, 5];
    good_take_slice(&my_arr);

    let my_vec = vec![1, 2, 3, 4, 5];
    good_take_slice(&my_vec);

    good_take_slice(&my_arr[..]);
}
```

▼ String slices → first word

```
fn first_word(s: &str) -> &str {
    let mut end_index = 0;

    for (index, item) in string.as_bytes().into_iter().enumerate() {
        // note that we need to dereference item since byte iter gives
        // but we want to compare u8 -> b' ' means compare to byte f
        if *item == b' ' {
```

```
                // return a slice from the start to the end_index
                return &s[..end_index];
            }

            end_index += 1;
        }

        // return a slice that is the entire string
        &s[..]
}

fn main() {
        // prints "Hello" → string literals are &str type
        println!("{}", first_word("Hello matey"));

        let s1 = String::from("Hello matey");

        // prints "Hello"
        println!("{}", &s1[..]);

        // we pass in an &String instead of &str and it still works
        println!("{}", first_word(&s1));

        let s2
}
```

▼ Week 3 — Lifetimes

The lifetime or a term refers to how long it exists for in code

- Scopes are defined by braces → similar to a stack frame
- We denote a lifetime `a, `b, `c
- `b: `a → we read this as `b outlives `a
- `static means it lives forever → global variables
- We cannot assign lifetimes ourselves in Rust → let z = &`b a does not work → we cannot explicitly only borrow a for `a

- Crummy tip → don’t worry about lifetime naming until the compiler says theres a problem

▼ Scopes and Lifetime explanation

```
// start scope
{
	// start t's lifetime
	let t = T{ ... };
	// start tb's lifetime
	let tb = &t;

	// borrow expires first, so tb's lifetime ends
	// t's lifetime ends
}
// end scope

// start outer scope
{
	// start t's lifetime
	let t = T { ... };

	// start inner scope
	{
		// start tb's lifetime
		let tb = &t;

	}
	// end inner scope and tb's lifetime
}
// end outer scope and t's lifetime


// the Rust compiler is also smart enough to truncate lifetimes so that th
// are as short as possible if needed to compile

// we refer to such inferred lifetimes as non-lexical lifetimes
```

```
{
	let t = T { ... };

	{
		let tb = &t;

		// compiler truncates tb and t's lifetime here, since they are n
		// used after this, and we want to transfer ownership to foo
		foo(t);
	}
	// end inner scope
}

// to reiterate the dangling reference example, Rust won't compile
// if it detects a dangling reference
fn dangling() {
	let mut r: &i32;

	{
		x = 5;
		r = &x;     // this line won't compile since r's lifetime goes
	}               // beyond x's → will say x "does not live long enough"
}
```

▼ Global data

```
static x: String = String::from("hello");

// borrow is valid for the program's entire lifetime
static y: &`static String = &x;

// but globals in Rust are generally avoided → difficult to use and
// dangeorous
```

Rust’s compiler also needs to know the lifetimes when working with borrowed values

- Rust has a feature called named lifetimes → `&a`T`
- If we are working with multiple borrows, and have to return either one, the compiler needs to know the lifetime of the returned result
    - we must and should only name the lifetimes of any inputs that are returned

▼ Functions working with lifetimes

```
BAD → COMPILE ERRORS ABOUT LIFETIME

fn longest1(x: &str, y: &str) → &str {
	if x.len() > y.len() {
		x
	} else {
		y
	}
}

fn foo1() {
	let long_lifetime = String::from("hello");

	{
		let short_lifetime = String::from("Goodbye, everybody, my tim

		// what is the lifetime of longer?
		// well, in our case, its within this inner scope, but how can th
		// compiler be sure?
		let longer = longest1(&long_lifetime, &short_lifetime);
	}
}

GOOD → NAMED LIFETIMES
	- longest2 and longest3 are idiomatic

// longest has a named lifetime param `a
// Rust compiler will choose `a to be the min lifetime between x and y
fn longest2<`a>(x: &`a str, y: &`a str) → &`a str {
```

```
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

// alternatively, we can also manually name and set lifetimes
// using where to set the conditions → `x and `y outlive (>=) `z
fn longest3<`x, `y, `z>(x: &`a str, y: &`y str) → &`z str
where
    `x: `z,
    `y: `z,
{
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn foo2() {
    let long_lifetime = String::from("hello");

    {
        let longer;
        {
            let short_lifetime = String::from("Goodbye, everybody, m

            // the compiler can now be sure of the lifetime of longer
            longer = longest2(&long_lifetime, &short_lifetime);

            // this is happy, since longer exists
            println!("{}", longer);
        }

        // this is not happy, as short_lifetime does not live long enoug
        // even though longer does
```

```
            // println!("{}", longer);
        }
}
```

Structs and Enums can contain borrows

- The struct/enum will have a lifetime equal to the shortest lifetime of the borrowed terms it has → requires named lifetimes in the struct definition

▼ Structs and Enums with Lifetimes

```
// struct StructWithBorrow {
//      x: &String,                 this line will not compile
// }

struct StructWithBorrow<`a> {
        x: &`a String,
}

// same for Enums
enum EnumWithBorrow<`a> {
        Owned(String),
        Borrowed(&`a String),
}

// the <`_> is optional in newer Rust versions, but nice to have for clarit
fn foo(s: &String) → StructWithBorrow<`_> {
        StructWithBorrow {
                x: s,
        }
}

fn main() {
        let s = String::from("Howdy!");
```

```
    let sb = foo(&s);
}
```

▼ Week 3 — Smart Pointers

Rust offers a range of 'Smart Pointer' types

- `Box<T>`
- `Rc<T>`
- `RefCell<T>`

A `Box<T>` is really just a unique pointer to some data of type T on the Heap

- Owned type → if you own `Box<T>`, you own the `T` inside → maintains that there can only be one owner of a Box
- Lets you put data of type `T` on the Heap and maintain access to it
- Uses `malloc` and `free` and all the usual pointer memory management stuff, which can make its performance poor if we put a lot of things in Boxes
- Will `free` when it goes out of scope (gets dropped)
- Allows us to create recursive data structures like linked lists and trees

▼ Box Example → linked list

```
struct ListNode {
    val: i32,
    next: Option<Box<ListNode>>,
}

fn print_node(curr: &Box<ListNode>) {
    print!("{}", curr.value);

    match &curr.next {
        Some(next) ⇒ {
            print!(" → ");
            print_node(&next);
        }
```

```
            None => {}
        }
}

fn main() {
        // create 3 nodes
        let tail = Box::new(ListNode { val: 3, next: None });
        let mid = Box::new(ListNode { val: 2, next: Some(tail) });
        let head = Box::new(ListNode { val: 1, next: Some(mid) });

        // prints 1 -> 2 -> 3
        print_node(head);
}
```

An `Rc<T>` (reference counted) is a pointer to some data of type `T` on the Heap, but it also tracks how many references there are to that `T`, functioning as an access point to `T`

- the `.clone()` method for `Rc<T>` doesn't create a copy on the heap, but actually increments the number of references to that `T`
- If there are no references, the Rc gets dropped and cleans up the memory, otherwise it persists → allows for it to exist across scopes by adding and dropping 'referees'
- Similar to Java instance objects → get freed if there is nothing referencing it anymore → but Java uses garbage collection
- Effectively enables share borrowing without a lifetime on `T`
- Limited in that there is a small overhead in incrementing and decrementing the counter when creating or dropping references
- **NOTE**: access to `T` through an `Rc<T>` is strictly read only → otherwise we'd have memory and thread safety issues

▼ Rc<T> Simple Example

```
use std::rc::Rc;
```

```
fn main() {
	let longer;

	{
		let my_string = String::from("Howdy!");
		let rc: Rc<String> = Rc::new(my_string);

		// rc increments counter
		longer = rc.clone();

		// both will print "2, Howdy!" as there are 2 referees
		println!("{}, {}", Rc::strong_count(&rc), rc);
		println!("{}, {}", Rc::strong_count(&longer), longer);
	}
	// scope ends so reference_counted is dropped, but there's still or
	// reference (longer), so my_string persists on the Heap

	// prints "1, Howdy!"
	println!("{}, {}", Rc::strong_count(&longer), longer);
}
// now there are no more
```

▼ Rc<T> Cycle Explanation

```
struct Struct1 {
	rc: Option<Rc<Struct2>>,
}

struct Struct2 {
	rc: Option<<Rc<Struct1>>,
}
```

we clearly have a cycle between Struct1 and Struct2, so what if we try to drop a Struct1 instance? Then we need to decrement the counter for Rc<Struct1> and drop a Struct2. Since we drop a Struct2, we need to d the counter for Rc<Struct2> and drop a Struct1. And so on and so forth we only have 1 reference of each Rc, which is the other Rc and get stuc a cycle → the Rcs will never get freed

`RefCell<T>`

- Sort of allows borrow-checking rules to be deferred to runtime → will panic! and crash your code if you do have any borrow errors at runtime →
- Only use if the compiler won't let you do something that you are absolutely sure will work
- Allows interior mutability → mutating `T` when you shouldn't be able to

▼ RefCell<T> Simple Example

```
use std::cell::RefCell;

struct Data {
    val: i32,
}

fn bad_will_panic() {
        // wrap data in RefCell
        let data = RefCell::new(Data { val: 10 });

        {
                // create an immutable borrow
                let immut_ref = data.borrow();
                println!("{}", immut_ref.val);

                // since a shared borrow exists, this will cause a panic!
                let mut mut_ref = data.borrow_mut();
                mut_ref.val += 1;
        }

        println!("{}", data.borrow().val);
}

fn good() {
        // wrap data in RefCell
```

```
    let data = RefCell::new(Data { val: 10 });

    {
        {
            // create an immutable borrow
            let immut_ref = data.borrow();
            println!("{}", immut_ref.val);
        }

        let mut mut_ref = data.borrow_mut();
        mut_ref.val += 1;
    }

    // prints 11
    println!("{}", data.borrow().val);
}
```

▼ Week 4 — Modularity

Cargo is Rust's build system and package manager

- Cargo is made up of crates → each crate is a compilable project and can be a library or binary or both
  - Binary is executable
  - Library is a package

Within a library, modules and functions can have different levels of visibility/access modifiers

- By default, the visibility is private, meaning it cannot be accessed outside the module (only in the module and sub-modules)
- Public (`pub`) features can be used anywhere
- `pub(crate)` features can be used anywhere within the crate → other modules inside the same crate including sub-crates and sub-modules
- `pub(path)` features can be used by any of the crates/modules in the specified path(s)

▼ Library + Struct classes (r_class.rs) Example

```
Wk4-Mod
    |__ Cargo.toml
    |__ Cargo.lock
    |__ target
    |__ src
        |__ lib.rs
        |__ main.rs
        |__ one_file_module.rs
        |__ folder_module
            |__ mod.rs
            |__ r_class.rs
            |__ b.rs
            |__ another_folder_module
                |__ ...
                |__ ...
```

▼ Cargo.toml

```
[package]
name = "Wk4-Mod"
version = "0.1.0"
edition = "2024"

[dependencies]
some_other_cargo = { path = "../some_other_cargo" }
```

▼ src/lib.rs

```
use some_other_cargo::do_other_thing;

// we define any modules we will create in other files within the car
// and the Rust compiler will try to find these files
pub mod one_file_module;

pub mod folder_module;
```

```
// or we can define modules inline in the same file
// noting that this one doesn't have pub and is default private
// (cannot be used in other non-sub cargos)
mod inline_module {
	pub fn double_num(x: i32) -> i32 {
		2 * x
	}

	pub fn weak_double(x: i32) -> i32 {
		// super goes up one level in the module, kind of like ./.. in
		super::add_two(x, x)
	}
}

pub fn add_two(x: i32, y: i32) -> i32 {
	Wk4_Mod::inline_module::double_num(x / 2) + y
}

pub fn use_other_fn() {
	println!("{}", do_other_thing());
}
```

▼ src/one_file_module.rs → access/visibility modifiers

```
fn private_function() {
	println!("PRIVATE!!!");
}

pub(self) fn also_private() {
	println!("ALSO PRIVATE!!!");
}

pub(super) fn semi_private() {
	println!("THIS MODULE AND THE SUPER ONE ONLY");
}

pub(crate) fn crate_visible() {
```

```
        println!("ONLY IN THIS CRATE");
}

pub fn exhibitionism(s: &str) {
        println!("FOR THE WHOLE WORLD TO WITNESS MY GLORY A
        println!("{s} TO THEIR KNEES BEFORE THE MIGHT OF THE GL
}

// imaginary must be within the same crate, and imaginary and any
// have access to this
pub(in crate::imaginary) favouritism() {
        println!("ONLY FOR THIS MODULE AND 'imaginary'");
}

mod imaginary {
        mod imaginary2 {
                fn something() {
                        crate::favouritism();
                }
        }

        fn something() {
                crate::favouritism();
        }
}
```

▼ src/folder_module/mod.rs

All folder modules must have a mod.rs file

- Put whatever you want → structs, enums, functions, constants, static data, etc.
- Idiomatically, we re-export stuff in the folder module
  - Allows us to not have to go very deep into the module-tree to use things when we export a module
- Kind of like a nice optional interface for a module

```
pub mod a;
pub mod b;

pub use a::function_from_a;
pub use b::function_from_b;
```

▼ src/folder_module/r_class.rs → structs and struct methods → Rust “classes”

Rust “classes”

- Rust does not have an explicit class system like Java does, but we can kind of create a class through structs and by implementing methods for structs
- We still do NOT have inheritance, however

```
// structs and their fields have visibility modifiers too
// fields are also private by default
pub struct Foo {
	x: i32,
	pub y: i32
}

// implement method(s) for the Foo struct
impl Foo {
	// define a constructor if we have private fields
	// or want to do some initialisation work or place constraints
	pub fn new(val: i32) -> Option<Self> {
		if val > 42 {
			return None;
		}

		Some(Self {
			x: 0,
			y: val,
		})
	}
```

```
        // we can overload methods and constructors
        // but typically we prefer builder pattern over telescoping
        pub fn new(val1: i32, val2: i32) -> Option<Self> {
                Some(Self {
                        x: val1,
                        y: val2,
                })
        }

        pub fn new(val1: i32, val2: i32, val3: i32) -> Option<Self> {
                Some(Self {
                        x: val1 % val3,
                        y: val2 - 5 * val3,
                })
        }

        pub fn print_x(&self) {
                println!("Foo has x = {}", self.x);
        }

        pub fn print_y(&self) {
                println!("Foo has y = {}", self.y);
        }
}

pub fn function_from_a() {
        println!("AAAAAAAAAAAAAAAAAA");
}
```

▼ src/folder_module/b.rs

```
pub fn function_from_b() {
        println!("CCCCCCCCCCCCCCCCCCCC... thought it was gon
}
```

▼ Week 4 — Testing & Documentation

Unit Tests in Rust

- We write tests by declaring functions with a `#[test]` header
- We run all tests in a cargo through the command `cargo test --lib`
- It's fine to just put unit tests at the bottom of the file/module they test → very common to define an inline submodule for testing and place them there

Integration Tests in Rust

- We usually make another folder module and place test files in there → declared as a normal `.rs` file → will act as a separate crate → ensures your library works externally
- Will need to import (`use::`) modules/functions/features

▼ Basic Example

```
pub fn add_two(x: i32, y: i32) → i32 {
	x + y
}

fn panics(x: i32) → i32 {
	if x != 42 {
		panic!("NOOOOOO");
	}

	x
}

#[cfg(test)]	// specify to only build this module when testing
mod tests {
	use super::*;

	#[test]	// declare that the function is a test
	fn test_1() {
		assert_eq!(add_two(2, 3), 5);
	}
```

```
        // note that assertions will panic!
        #[test]
        fn test_2() {
            // panic!
            assert_ne!(add_two(0, 0), 0);
        }

        // if we expect a panic!, we need to add another attribute
        #[test]
        #[should_panic]
        fn test_3() {
            assert_eq!(panics(43), 43);
        }

        #[test]
        #[should_panic]
        // this test also passes since the assert_eq panics!
        fn test_3() {
            assert_eq!(panics(42), 43);
        }
}
```

Documentation in Rust

- We write documentation using `///` above a function as a header comment
- We typically start with a function description, then declare any panic cases within a `# Panics` section, and finally give an example within the `# Examples` section
- ^ Actually just markdown
- We can create links to other things in our documentation through `[`Other`]`
- Within our documentation, we can write documentation code by enclosing it in ``` ``` within the `# Examples` section
    - If we write tests in here, they can be run
    - The compiler will check that the code here actually compiles

    - We can hide lines that we don't want displayed by using `/// #` within the code block
- We can document the module itself (instead of a specific function or sth) by using `//!`
- `6991 cargo doc` generates our documentation in a file found at `target/doc/crate_name/index.html`
- By default, documentation for private features is hidden from the index.html
- `#![warn(missing_docs)]` attribute will warn about missing documentation for any public features in the module

▼ Documentation Example

```
//! This is a crate. I cba naming it and everything, so here we go

/// # A wrapper struct for an i32
/// HOWDY!
///
/// # Warning
/// requires an i32 to construct...
///
/// ```
/// let s = Foo {
///     x: 42,
/// };
/// println!("{}", s.x);
/// ```
pub struct Foo {
        pub x: i32,
}

/// # A public function that takes 2 numbers and returns the sum
///
/// Check out this other thing called [`Foo`] ← click on the link
///
/// # Panics
```

```
/// This function panics if either x or y is negative
///
/// # Example
/// ```
/// # use crate_name::module_name::add_two;
/// let r1 = add_two(40, 2);
/// assert_eq(r1, 42);
/// assert_ne(add_two(r1, 42), 88);
/// ```
pub fn add_two(x: i32, y: i32) -> i32 {
	if x < 0 | y < 0 {
		panic!("Cannot use negative nums because I said so!");
	}

	x + y
}
```

Commands

- `6991 cargo test` runs all tests in the cargo
- `6991 cargo test --lib` runs all unit tests
- `6991 cargo test --test my_tests` runs integration tests in a crate called my_tests
- `6991 cargo test --doc` runs documentation tests

▼ Week 5 — Polymorphism (generics)

Rust has object-oriented capabilities despite not being a specially object-oriented language like Java

- Main lacking point is no inheritance over types
- Most OOP patterns can be emulated in Rust

Like Java, Rust provides a way to use generics

- Rust takes a monomorphisation approach to this → the compiler actually creates different copies of the generic code for each concrete type that

interacts with it

- Monomorphisation allows for fast code at the cost of compile time and program size → compiler can use optimisations specific to each concrete type
  - 0-cost abstraction → using generics or making a function for each type has no difference in terms of cost/performance
- Requires us to know the exact concrete types that will be used with a generic feature at compile time

Note: Generally better to only declare type bounds where needed, otherwise it can get very messy very quickly

▼ Simple Generic Function

```
fn smallest_i32(x: i32, y: i32) → i32 {
	if x < y {
		x
	} else {
		y
	}
}

fn smallest_f64(x: f64, y: f64) → f64 {
	if x < y {
		x
	} else {
		y
	}
}

fn smallest_char(x: char, y: char) → char {
	if x < y {
		x
	} else {
		y
	}
```

```
}

// the above three (and possibly more for other types) could be replace
// by the below. Rust's monomorphisation approach actually means
// that the below compiles into machine code for each of the above
// functions if we use the below one for i32, f64 and char

// smallest accepts two arguments of the same type, and the type impl
// the trait PartialOrd (<, >, ==, etc.), and returns the same type
fn smallest<T: PartialOrd>(x: T, y: T) -> T {
    if x < y {
        x
    } else {
        y
    }
}

// alternative syntax
// fn smallest<T>(x: T, y: T) -> T
// where
//     T: PartialOrd
// { ... }

// Java equivalent would be
// public <T implements PartialOrd> T smallest(x: T, y: T)

// also note that our own custom types that implement PartialOrd
// can be used, but more on this in Traits

fn main() {
    // dbg! macro evaluates and prints and returns the expression
    // prints 42
    dbg!(smallest::<i32>(100, 42));

    // usually Rust compiler can infer the type for us
    dbg!(smallest(88, 89));
}
```

▼ Super Genericised

```
// smallest works for any amount of Ts, and takes in a container that
// implements IntoIterator for Items of type T (look at rust doc).
// If we specified a specific container like Vec, then sth like an array
// wouldn't work
fn smallest<T, I>(xs: I) -> Option<T>
where
	T: PartialOrd,
	I: IntoIterator<Item = T>,
{
	let mut iter = xs.into_iter();
	let mut min = xs.next()?;
	for item in xs.iter() {
		if item < min {
			min = item;
		}
	}
	Some(min)
}
```

▼ Week 5 — Traits

Traits allow us to do everything a Java interface does and more

- Also allows something called ad-hoc polymorphism
- Traits can be generic, and may sometimes require you to declare some types within the braces of an `impl`
- There are a lot of common traits in `std`
  - ToString, Display, Debug
  - Default, Copy, Clone
  - From, Into, TryFrom, TryInto, FromStr (used for `.parse()`)
  - Error → gives a bunch of default functions to present when an error occurs
  - Iterator, IntoIterator, FromIterator (`.collect()`)

- It is worth noting that we often have multiple implementations of a Trait for an item, a shared borrow of an item, and for an exclusive borrow of an item
    - IntoIterator is implemented for `Vec<T, A>`, `&Vec<T, A>`, and `&mut Vec<T, A>`

Since we don't have inheritance, we must use traits if we want to unify some things

- Could have an empty marker trait
- Could have a trait with methods both defined and undefined → abstract class type idea
- Can't have default methods related to any fields or data in the type unless we use getters/setters

NOTE: for the methods defined by a trait to be used, the trait itself must be in scope (i.e. imported or visible)

▼ Implementing a trait

```
struct MyStruct1 {
	x: i32,
	y: char,
	z: String,
}

// implement the Clone trait for MyStruct
impl Clone for MyStruct1 {
	// clone from std takes in a shared borrow of the calling item, and
	// returns an item of the same type
	fn clone(&self) -> Self {
		MyStruct1 {
			x: self.x,
			y: self.y,
			z: self.z.clone(),
		}
```

```
        }
}

// note that this could be achieved through
#[derive(Clone)]
struct MyStruct2 {
        x: i32,
        y: char,
        z: String,
}
// since Clone provides a derive macro → requires all fields to impleme
// Clone

// Traits also use generics
struct GenericStruct<T> {
        x: T,
        y: T,
}

// implement Add for GenericStruct with type T, where T implements A
// such a way that the output is T
impl<T> Add for GenericStruct<T>
where
        T: Add<Output = T>
{
        type Output = Self;

        // this is what gets called when we use the '+' operator
        fn add(self, other: Self) -> Self::Output {
                Self {
                        x: self.x + other.x,
                        y: self.y + other.y,
                }
        }
}
```

▼ Making a custom trait

```
pub trait Animal {
	fn name(&self) -> String;
	fn age(&self) -> u32;
	fn speak(&self) -> String;

	fn default_method() {
		println!("default done");
	}

	fn greet<A>(&self, animal: &A) -> String
	where
		A: Animal;
}

pub struct Crab {
	name: String,
	age: u32,
	snippy: bool,
}

pub struct WhaleShark {
	name: String,
	age: u32,
	kill_streak: i128,
}

impl Animal for Crab {
	fn name(&self) -> String {
		self.name.clone()
	}

	fn age(&self) -> u32 {
		self.age
	}

	fn speak(&self) -> String {
		if self.snippy {
			String::from("*snip* *snip*")
```

```
            } else {
                String::from("...geopolitical state of the world...")
            }
        }

        fn greet<A>(&self, animal: &A) -> String {
            if self.snippy {
                format!("*snip* *snip*, {}, *snip* *snip*", animal.name())
            } else {
                format!("Howdy, {}! Good weather today! Great for some
            }
        }
}

impl Animal for WhaleShark {
        fn name(&self) -> String {
            self.name.clone()
        }

        fn age(&self) -> u32 {
            self.age
        }

        fn speak(&self) -> String {
            String::from("NEVER FORGET 19XX RAHHHHHHHHHHH")
        }

        fn greet<A>(&self, animal: &A) -> String {
            String::from("d4079249d325ba7cff1554aa1c2209840d40b19
        }
}

pub fn main() {
        let c = Crab {
            name: String::from("Crabulon"),
            age: 1,
            snippy: true,
        };
```

```
    let ws = WhaleShark {
        name: String::from("Willy"),
        age: 888,
        kill_count: std::i128::MAX - 37,       // boutta evolve into kyogre
    };

    println!("{}", c.speak());
    println!("{}", ws.speak());
    println!("{}", c.greet(&ws));
    println!("{}", ws.greet(&c));
}
```

We can specify any type requirements on the struct or on the 'impl'

- if specified on the struct, then it can only be created with the specified types
- If specified on the `impl`, then the methods defined in the `impl` can only be used with structs of matching types

Dynamic dispatch of traits allows us to have true generic behaviour without monomorphisation

- With monomorphisation, if we said a `Vec<T>` where `T: SomeTrait`, all `T`s in the `Vec` must be of the same concrete type
- the `dyn` keyword allows us to use trait objects (dynamic dispatch), but we cannot have collections of trait objects → must be collections of pointers to trait objects
- Using dyn has a small runtime cost, but faster compile time since copies for each concrete type do not have to be made

Object Safety Rules

- Any traits with a function using generic parameters cannot be dynamically dispatched

    - Make the function use a trait object instead of generics
    - Move the function into a separate trait and make the types implement both traits
- Any traits with a function that returns Self cannot be dynamically dispatched
- Any traits that have a `Sized` bound on `Self` → no `trait SomeTrait: Sized`

▼ Trait Objects → builds on Making a Custom Trait example

```
// the greet method uses generics, so we can't create trait objects of A
pub trait Animal1 {
	fn name(&self) -> String;
	fn age(&self) -> u32;
	fn speak(&self) -> String;

	fn greet<A>(&self, animal: &A) -> String
	where
		A: Animal;
}

// greet method now uses a trait object as well, and so we can create
// trait objects
pub trait Animal2 {
	fn name(&self) -> String;
	fn age(&self) -> u32;
	fn speak(&self) -> String;

	fn greet<A>(&self, animal: &dyn Animal) -> String;
}

// this signature will not compile since compiler needs to know
// the exact size of elements in a Collection
// fn speaking_animals(animals: Vec<dyn Animal1>)

// nor will this
fn speaking_animals(animals: Vec<dyn Animal2>)
```

```
// this will compile since borrows all have the same size
// fn speaking_animals: Vec<&dyn Animal2>)

// this will also compile since it is a Vec of Boxes, and
// all boxes have the same size
fn speaking_animals(animals: Vec<Box<dyn Animal2>>) {
	// note that elements in animals can only call methods from Anima
	for a in animals {
		println!("{}", a.speak());
	}
}

fn main() {
	let animals: Vec<Box<dyn Animal2>> = vec![
		Box::new(Crab {
			name: String::from("Crab1"),
			age: 1,
			snippy = false,
		}),
		Box::new(WhaleShark {
			name: String::from("ws1"),
			age: 2,
			kill_streak = 88,
		}),
		Box::new(Crab {
			name: String::from("Crab2"),
			age: 3,
			snippy = true,
		}),
	];

	speaking_animals(animals);
}
```

▼ Trait Objects → separating traits and Sized short example

```
pub trait Animal3 {
	fn name(&self) -> String;
	fn age(&self) -> u32;
	fn speak(&self) -> String;
}

// new trait that can only be implemented by types that have implement
pub trait AnimalGreet: Animal3 {
	fn greet<A>(&self, animal: &A) -> String
	where
		A: Animal;
}

// uses Sized constraint to make greet implemented for Traits, but not t
// allowing trait objects to be created for Animal4
pub trait Animal4 {
	fn name(&self) -> String;
	fn age(&self) -> u32;
	fn speak(&self) -> String;
	// the greet function is only implemented on types that are Sized
	// i.e. not on trait objects
	fn greet<A>(&self, animal: &A) -> String
	where
		A: Animal,
		Self: Sized,
}
```

▼ Fibonacci Iterator Example

```
struct Fibonacci {
	curr: u32,
	next: u32
}

fn fibonacci() -> Fibonacci {
	Fibonacci {
```

```
            curr: 1,
            next: 1,
        }
}

impl Iterator for Fibonacci {
        type Item = u32;

        fn next(&mut self) -> Option<Self::Item> {
            let curr = self.curr;
            let tmp = curr + self.next;
            self.curr = self.next;
            self.next = tmp;

            Some(curr)
        }
}

fn main() {
        for i in fibonacci().take(10) {
            println!("{}", i);
        }
}
```

▼ Custom Vec Iterator

```
// a VecIter must live at least as long as the Vec
struct VecIter<`a, T> {
        vec: &`a Vec<T>,
        curr_pos: usize,
}

// note that the lifetimes are inferred for us
fn my_vec_iter<T>(vec: &Vec<T>) -> VecIter<T> {
        VecIter {
            vec: vec,
            curr_pos: 0,
        }
```

```
}

impl<a`, T> Iterator for VecIter<a`, T> {
	// vec is a &`a Vec<T>, so the return must be a n &`a T
	type Item = &`a T;

	fn next(&mut self) -> Option<Self::Item> {
		let item = self.vec.get(self.curr_pos)?;
		self.curr_pos += 1;
		Some(item)
	}
}
```

▼ Week 7 — Metaprogramming

Metaprogramming is about writing code that produces new code we can run

- In Rust. this is mainly present through Macros! → largely used for convenience, polymorphism and syntax extension
- polymorphism → macros can replicate monomorphisation behaviour very simply → not used much since Rust already has decent support
- Macros are advantageous in that none of the overhead required for a function call is needed to use a macro
- Macros effectively work by substituting things into code

Declarative macros are those that are relatively simple and try to match values into code

- Aka macros by example
- Easy to read and write, and semantically simple, but limited in capability
- In Rust, we declare declarative macros using `macro_rules! macro_name { ...rules...}` where the `...rules...` are `() => ` structures
  - Each rule takes in a predefined set of tokens
  - Compiler will try to exact match input tokens to the defined rules to determine which one it should execute

- Rust has a bunch of macro metavariables that we can use in our rules, the most common being
  - `item` → an item
  - `block` → a block expression `{ ... expr ... }`
  - `expr` → an expression
  - `stmt` → a statement without the trailing semicolon (except for item statements, they still need the semicolon)
- Less common ones:
  - `ty` → a type
  - `ident` → an identifier or keyword or ra identifier
  - `$my_label:meta_var` → declares that a rule should expect a token that is a `meta_var` metavariable, and that `meta_var` should be substituted wherever `$my_label` is used in the macro

▼ C/C++ Declarative Macro Examples

```
#include <stdio.h>
#include <stdlib.h>

// constants in C are (object-style) macros behind the scenes
#define MY_CONSTANT 42
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define V1_TIMES_5(x) x * 5
#define V2_TIMES_5(x) (x) * 5
#define V1_ADD_5(x) (x) + 5
#define V2_ADD_5(x) ((x) + 5))

int print_and_return(int a) {
	printf("%d\n", a);
	return a;
}

int main(void) {
	// print 42 and 3.14
```

```
    printf("%d\n", MAX(10, MY_CONSTANT));
    printf("%d\n", MAX(3.14, 2.78));

    // prints 4, 2, 4, 2, 4
    printf("%d\n", MAX(print_and_return(4), print_and_return(2)));
    // Note that the Rust equivalent may print 4, 2, 4 since we can mak
    // evaluate each expr once and store the result for use

    // prints 12
    printf("%d\n", V1_TIMES_5(2 + 2));
    // prints 20
    printf("%d\n", V2_TIMES_5(2 + 2));

    // prints 24
    printf("%d\n", V1_ADD_5(2 + 2) * 4);
    // prints 36
    printf("%d\n", V2_ADD_5(2 + 2) * 4);
}

// compiled main → notice how we literally just substitute the inputs
// into the defined code
int compiled_main(void) {
    printf("%d\n", ((10) > (MY_CONSTANT) ? (10) : (MY_CONSTANT)))
    printf("%d\n", ((3.14) > (2.78) ? (3.14) : (2.78)));

    // the function call is substituted and executed every time
    printf("%d\n", ((print_and_return(4)) > (print_and_return(2)) ? (prin

    // * operator takes precedence so we get 2 + (2 * 5) = 12
    printf("%d\n", 2 + 2 * 5);
    // brackets, so (2 + 2) * 5 = 20
    printf("%d\n", (2 + 2) * 5);

    // * operator takes precedence, so we get (2 + 2) + (5 * 4) = 24
    printf("%d\n", (2 + 2) + 5 * 4);
    // brackets make it ((2 + 2) + 5) * 4 = 36
    printf("%d\n", ((2 + 2) + 5) * 4);
}
```

▼ Rust Declarative Macro Examples

```
// declare a new declarative macro called my_vec
macro_rules! my_vec {
    // needs exact syntactical token match, so this will never be called
    // unless you do my_vec![item, foo, bar]
    (item, foo, bar) => {
        Vec::new()
    };

    // equiv to vec![]
    () => {
        Vec::new()
    };

    // rule takes in an expr token, which we will use the placeholder
    // $some_item for
    ($some_item:expr) => {
        {
            let mut vec = Vec::new();
            vec.push($some_item);
            vec
        }
    };

    // we can also have repetitions
    // * = any amount, + = at least one, and an optional separator toke
    // also have ?, which means 0 or 1 occurrences of something

    // we have one expr, or multiple comma separated exprs
    ($($some_item:expr),+) => {
        {
            let mut vec = Vec::new();
            // this bit repeats for each of the expr tokens input
            $(
                vec.push($some_item);
            )+
```

```
                vec
            }
        }
        // ^^ equivalent to vec![x1, x2, x3, ...]

        // and similar for * repetition operator where exprs are semi-colon
        // but this time the tokens may or may not have a trailing semi-colo
        ($($some_item:expr);* $(;)?) => {
            {
                let mut vec = Vec::new();
                // this bit repeats for each of the expr tokens input
                $(
                    vec.push($some_item);
                )*

                vec
            }
        }
        // ^^ this rule is good if we have trailing commas:
        // my_vec![
        //     42,
        //     41,
        // ]
}
```

▼ Rust Small for-loop Declarative Macro Example

```
macro_rules! up_to {
    ($var:ident, $n:expr, $body:block) => {
        for $var in 0..$n {
            $body
        }
    };
}

macro_rules! c_style_for {
    (for ($init:stmt; $cond:expr; $step:stmt) $body:block) => {
        $init
```

```
            while ($cond) {
                $body
                $step
            }
        };
}

fn main() {
        // for foo in 0..10 { println!("{foo}"); }
        up_to![foo, 10, {
            println!("{foo}");
        }];

        // compiler matches this exactly to the tokens
        // for (
        // $init: let mut i = 0
        // ;
        // $cond: i < 10
        // ;
        // $step: i += 1
        // )
        // $body: { println!("{i}"); }
        c_style_for![
            for (let mut i = 0; i < 10; i += 1) {
                println!("{i}");
            }
        ];

        // note that we can actually use any brackets {, (, [ for macros
        println!("hello");
        println!{"I am"};
        println!["Felix"];
}

// the c_style_for! in main expands to
let mut i = 0;
while (i < 10) {
```

```
        {
            println!("{i}");
        }
        i += 1;
    }
```

Procedural Macros are types of procedures that behave more like functions

- They take code as input, manipulate it, and return the output code
- 3 types:
  - custom derive → `#[derive(traits)]`
  - attribute-like → `#[attribute]`
  - function-like →
- Can be very difficult to read and write, but are very very powerful

Note: `6991 cargo expand fName`

- Will expand out any macros used in a function called `fName` within the current crate
- Function has to be in `main.rs`
- This command working doesn't necessarily mean your code compiles

▼ Week 7 — Functions

Function pointers `fn`

- Functions are first class citizens → can be treated as terms (values)
- We pass around function pointers, which have types similar to their signature → `fn(T, U, V, ...) → R`

Closures (anonymous functions) are effectively a function pointer in an environment/capture

- Environment/capture refers to the external data from the surrounding scope that is used by the closure
  - If a closure captures data from its surrounding scope, it cannot be used as a function pointer
- If a closure has an environment, it **CANNOT** be used as a function pointer
  - Compiler actually has to make a struct for borrows of the captured values (including lifetimes)
  - Then compiler will implement `Fn` for the created struct
  - But the struct contains shared borrows so we can't mutate any values in it either
- The concrete type of a closure is indeterminable → only the compiler can name the type internally → we don't really care as long as we can use it

`F...(T, U, V, ...) -> R`

- Compiler will actually create a `struct` from the captured values, then implement `Fn`, `FnOnce` and/or `FnMut` on the `struct` based on how the closure interacts with the environment
- Since we don't know what the compiler actually names the `struct`, we can't determine the type of the closure, but do know that it implements a `F...` trait, so we can call it
- We can use `F...(T, U, V, ...) -> R` instead of `fn(T, U, V, ...) -> R` for type bounds to make something able to take in both functions and closures with captures
- Note that `fn` can be treated as `Fn`, and `Fn` can be treated as `FnMut`, and `FnMut` can be treated as `FnOnce`, and they chain up
- `FnOnce`
  - Ownership is transferred to compiled `struct` → can only be called once and allows all closures → the captured values and closure can never be used again

    - Use as a type bound when we don't need to call the closure multiple times
    - Implemented for all closures
- `FnMut`
    - Uses exclusive `&mut` borrow in the compiled `struct` → can be called indefinitely, but never simultaneously, and captured terms will persist
    - Use as a type bound when we need to call the closure multiple times but never simultaneously
    - Implemented if closure uses an `&mut` of a term in the environment
    - Only `FnMut` can modify the environment
- `Fn`
    - Uses shared `&` borrow in the compiled `struct` → can be called indefinitely and simultaneously, and captured terms will persist
    - Use as a type bound when we need to call the closure multiple times simultaneously
    - Implemented if closure uses `&` of a term in the environment
- Note that accepting `FnOnce` accepts all `F...` and `fn`, accepting `FnMut` accepts all `F...` and `fn` except `FnOnce`, accepting `Fn` accepts `Fn` and `fn`, accepting `fn` only accepts `fn`
    - Flexible code will try to accept as many things as possible → use `FnOnce` over `FnMut` and `FnMut` over `Fn` and so on where possible

▼ Simple Function Pointer and Fn Example

```
// f must implement Fn such that it takes 2 i32s and returns an f32
fn foo(f: F) -> f32
	where
		F: Fn(i32, i32) -> f32
{
	f(2, 2)
}
```

```
fn main() {
	let factor = 0.5;
	// note that thingthaniel is not a function pointer, but will be sometl
	// that implements Fn
	let thingthaniel = |x: i32, y: i32| -> f32 { (x + y) as f32 * factor };
	let thingy: foo(thingthaniel);
	// prints OMG 2, NO WAY!!!
	println("OMG {}, NO WAY!!!", thingy());
	let thingo: fn() = main;

	// infinite recursion since this is just main()
	thingo();
}

// the compiler would've interpreted thingthaniel as
struct ThingthanielClosure<`a> {
	borrow_factor: &`a f32,
}

impl Fn(i32, i32) -> f32 for ThingthanielClosure<`_> {
	fn call(&self, x: i32, y: i32) -> f32 {
		(x + y) as f32 * self.borrow_factor
	}
}
```

▼ Iterator Map Extension Example

```
use std::f32::consts::PI;

struct MyMap<I, CurrType, NewType> {
	iter: I,
	func: fn(CurrType) -> NewType
}

impl<I, CurrType, NewType> Iterator for MyMap<I, CurrType, NewType
where
	I: Iterator<Item = CurrType>, // note that we need this where claus
{				// even if we had it in the struct
```

```
        type Item = NewType;

        fn next(&mut self) -> Option<Self::Item> {
            let curr_item = self.iter.next()?;
            // parentheses are needed or else the compiler will look for a
            // called func instead of the func field of MyMap
            Some((self.func)(curr_item))
        }
}

// constructor
fn my_map<I, CurrType, NewType>(
        iter: I,
        func: fn(CurrType) -> NewType
) -> MyMap<I, CurrType, NewType>
where
        // I must be an iterator of items with CurrType type
        I: Iterator<Item = CurrType>,
{
        MyMap {
            iter,
            func,
        }
}

fn main() {
        let vec = vec![1, 2, 3, 4, 5];

        // let factor = 2.5;
        // this will not compile since the closure captures factor
        // let mapped = my_map(vec.into_iter(), |x| x * PI * factor);

        // this is fine since the closure just uses a constant and the supplie
        let mapped = my_map(vec.into_iter(), |x| x * PI);
}
```

▼ Iterator Map Allowing Environment Closures

```
// NOTE: Using FnMut would've been better for flexibility, but for the sa
// of the example, Fn was used

use std::f32::consts::PI;

struct MyMap<I, F> {
	iter: I,
	func: F,
}

impl<I, F, CurrType, NewType> Iterator for MyMap<I, F, CurrType, New
where
	I: Iterator<Item = CurrType>,
	F: Fn(CurrType) -> NewType,
{
	type Item = NewType;

	fn next(&mut self) -> Option<Self::Item> {
		let curr_item = self.iter.next()?;
		Some((self.func)(curr_item))
	}
}

// we could've kept the CurrType param, but this works too
fn my_map<I, F, NewType>(
	iter: I,
	F: Fn(<I as Iterator>::Item) -> NewType,
) -> MyMap<I, F>
where
	I: Iterator,
{
	MyMap {
		iter,
		func,
	}
}

fn main() {
```

```
    let vec = vec![1, 2, 3, 4, 5];

    let factor = 2.5;
    // this will now compile, since compiler implements Fn for it
    let mapped = my_map(vec.clone().into_iter(), |x| x * PI * factor);

    // this is fine since the closure just uses a constant and the supplie
    let mapped = my_map(vec.into_iter(), |x| x * PI);
}
```

▼ `Fn...` Example

```
fn check_if_fnonce(f: impl FnOnce()) {};
fn check_if_fnmut(f: impl FnMut()) {};
fn check_if_fn(f: impl Fn()) {};

fn main() {
    let s1 = String::from("Howdy!");
    let s2 = String::from("My name is");
    let s3 = String::from("Felix");

    let x = || {
        let a1 = &s1;
        let a2 = &mut s2;
        let a3 = s3;
    }

    // this will compile
    check_if_fnonce(x);

    // this would compile if we didn't have a3
    // check_if_fnmut(x);

    // this would compile if we didn't have a2 and a3
    // check_if_fn(x);
}
```

▼ `FnOnce` Usecase Example

```
fn time_closure<F>(f: F) -> Duration
	where
		F: FnOnce()
{
	let start = Instant::now();
	f();
	Instant::now().duration_since(start)
}

fn main() {
	// we only ever call a closure that we want to time once
	// so it is suitable to use FnOnce
	let (duration, retval) = time_closure(|| {
		println!("Is this the real life?");
		println!("Is this just fantasy?");
		println!("Caught in a landslide, no escape from reality");
		// ...
		println!("Nothing really matters...");
		println!("Nothing really matters... to meeeeeee!");

		return 42;
	});

	// prints whatever duration is, 42
	println!("{:?}, {:?}", duration, retval);
}
```

▼ Week 8 — Concurrency Locks

A single threaded program goes through code one execution at a time

- Slow cause it can only really do one thing at a time

Concurrency uses multiple threads to execute multiple independent tasks ‘simultaneously’

- Context switching → we switch between the different tasks to run them concurrently → we do not actually do them all at the same exact time →

that is parallelism

- Blocking calls are calls that will stop execution to wait for something
  - Calls that wait for input, or fetch data or some response from an external source, etc. are likely blocking calls
  - Without concurrency, all computation will have to stop and wait for that something
  - With concurrency, we will switch to different tasks and keep working, switching back occasionally to check if its done
- Concurrency requires us to define different paths of execution (threads) to run concurrently

Note: In Rust, we can still sometimes intentionally create a data race by wrapping what we would normally mutex lock and unlock in an `unsafe {}` block

- Not everything can be chucked into `unsafe {}` and work
- allows us to use mutable static (global) variables, and read and write to them

▼ Simple Thread Example + `thread::spawn()` explanation

```
use std::thread;
// this program will print "howdy doody!" every second, and read and r
// a line of input at the 'same' time

// note that main itself is a thread
fn main() {
	thread::spawn(|| {
		loop {
			let mut line = String::new();
			std::io::stdin().read_line(&mut line);

			println!("Read line: {line}");
		}
	});
```

```
        loop {
            println!("howdy doody!");
            thread::sleep(Duration::from_secs(1));
        }
}

pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
        F: FnOnce() -> T,
        F: Send + `static,
        T: Send + `static,

// the `static on F and T means that everything in the FnOnce() (includir
// struct) must outlive the thread

// the Send means that F and T must be able to be transferred across tl
// implemented iff Sync is

// Sync is implemented iff Send is, and means that a type is safe to sha
// references between threads.
// types that have interior mutability in a non thread-safe form
// (e.g. Cel, Rc) are not Sync -> !Sync and cannot be used with threads

// Send and Share are unsafe auto marker traits -> only compiler can
// implement, and will do so automatically if it sees fit
```

▼ Rust Mutex Deadlock Example

```
fn main() {
        let mut1 = Mutex::new(());
        let mut2 = Mutex::new(());

        std::thread::scope(|my_scope| {
            // suppose T1 obtains mut1 lock and T2 obtains mut2 lock,
            // then T1 is waiting on mut2 to be unlocked, and T1 is waiting
            // mut1 to be unlocked, so they are deadlocked waiting for ea
```

```
            // T1
            my_scope.spawn(|| {
                for _ in 0..10000 {
                    let _1 = mut1.lock();
                    let _2 = mut2.lock();
                }
            });

            // T2
            my_scope.spawn(|| {
                for _ in 0..10000 {
                    let _2 = mut2.lock();
                    let _1 = mut1.lock();
                }
            });

            // fix by having locks obtained in the same order throughout t
            // entire program
        });

        std::thread::scope(|my_scope| {
            for _ in 0..10000 {
                let _1 = mut1.lock();
                let _2 = mut2.lock();
            }
        });
}
```

▼ C Mutex Example

```
#define N_THREADS 50
#define N_INCREMENTS 100000

int shared_global_num = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread(void *data) {
    for (int i = 0; i < N_ICREMENTS; i++) {
```

```
            // without the mutex lock (or if we forgot it), we would have a
            pthread_mutex_lock(&mutex);
            shared_global_num += 1;
            pthread_mutex_unlock(&mutex);
            // if we forget to unlock, the resource is always locked and nc
            // threads that are waiting for it to unlock will finish → deadloc
        }

        return NULL;
}

int main() {
        pthread_t *threads = malloc(sizeof(pthread_t) * N_THREADS);

        for (int i = 0; i < N_THREADS; i++) {
            pthread_create(&threads[i], NULL, thread, NULL);
        }

        for (int i = 0; i < N_THREADS; i++) {
            pthread_join(threads[i], NULL);
        }

        printf("TOTAL: %d\nEXPECTED: %d\n", shared_global_num, N_TH
}
```

▼ Java Mutex Example

```
class FixRace {
        static final int N_THREADS = 50;
        static final int N_INCREMENTS = 100000;

        static int shared_num = 0;
        // Java objects all have mutexes and can be synchronised
        static final Object my_lock = new Object();

        static void thread() {
            for (int i = 0; i < N_INCREMENTS; i++) {
```

```
                // synchronized obtains the lock on my_lock at {, and
                // unlocks at } → makes sure we never forget or misplace
                synchronized (my_lock) {
                    shared_num += 1;
                }
            }
        }
}
```

▼ Broken Rust Example

```
const N_THREADS: usize = 50;
const N_INCREMENTS: usize = 100000;

use std::array;

// this won't work, since Rust doesn't let you mutate a static
// and Rust doesn't like mutable statics → Rust doesn't like globals
// static SHARED_NUM: u64 = 0;

// so we try pass in an exclusive borrow
fn my_thread(shared_num: &mut u64) {
    for _ in 0..N_INCREMENTS {
        shared_num += 1;
    }
}

pub fn main() {
    // since the global doesn't work, we try defining it in main
    let mut shared_num: u64 = 0;

    let mut threads: [_; N_THREADS] = array::from_fn(|_| None);

    for i in 0..N_THREADS {
        // but the line below won't work, since the previous iteration
        // already took out an exclusive borrow, and that exclusive bo
        // outlives the thread (`static) so we can't create another thre
        threads[i] = Some(std::thread::spawn(|| my_thread(&mut shar
```

```
            // we also can't use Cell or Rc, since they are !Sync types →
            // see 'Simple Thread Example'
        }

        for i in 0..N_THREADS {
            threads[i].take().unwrap().join();
        }
}
```

▼ Working Rust Example 1 + `Mutex` and `Arc` Explanation

```
const N_THREADS: usize = 50;
const N_INCREMENTS: usize = 100000;

use std::array;

// won't work, lifetime problem, look below
// fn my_thread(shared_num: &Mutex<u64>) {

fn my_thread(shared_num: Arc<Mutex<u64>>) {
        for _ in 0..N_INCREMENTS {
            *shared_num.lock().unwrap() += 1;
        }
}

fn main() {
        // to avoid lifetime problem explained below, we will not use this
        // let shared_num: Mutex<u64> = Mutex::new(0);

        let shared_num: Arc<Mutex<u64>> = Arc::new(Mutex::new(0));
        let mut threads: [_; N_THREADS] = array::from_fn(|_| None);

        for i in 0..N_THREADS {
            // this still won't work, since the compiler is now worried that
            // thread will outlive the shared_num mutex
            // threads[i] = Some(std::thread::spawn(|| my_thread(&shared
```

```
            // so we use the Arc type, which effectively has no lifetime, ar
            // pass by ownership of a 'clone'
            let cloned = shared_num.clone();
            threads[i] = Some(std::thread::spawn(|| my_thread(cloned));
        }

        for i in 0..N_THREADS {
            threads[i].take().unwrap().join();
        }
}
```

Mutex<T>.lock(&self) returns a LockResult<MutexGuard<`_, T>, which
Result<MutexGuard<`_, T>, PoisonError<MutexGuard<`_, T>
- MutexGuard implements the Deref and DerefMut traits → we can acc
  locked T through the methods of these traits
- When a MutexGuard gets dropped (goes out of scope), it automatical
- implements Send and Sync if T does

Arc (Atomically Referenece Counted) is a thread-safe version of Rc
- exists as long as there is a reference to it → no lifetime same as Rc

▼ Working Rust Example 2 → Scoped Threads

```
const N_THREADS: usize = 50;
const N_INCREMENTS: usize = 100000;

use std::{array, cell::Cell};
use std::sync::{atomic::{AtomicU64, Ordering}, Mutex, Arc};

fn my_thread(shared_num: &Mutex<u64>) {
    for _ in 0..N_INCREMENTS {
        *shared_num.lock().unwrap() += 1;
    }
}

fn main() {
    let shared_num: Mutex<u64> = Mutex::new(0);
    let mut threads: [_; N_THREADS] = array::from_fn(|_| None);
```

```
    // creates a scope for spawning scoped threads
    std::thread::scope(|my_scope| {
        for i in 0..N_THREADS {
            // create a scoped thread
            my_scope.spawn(|| my_thread(&my_number);
        }

        // end of scope, automatically join all threads spawned
        // in the scope
    });

    println!("{}", *shared_number.lock().unwrap());
}

Scoped threads can borrow non `static data → scope guarantees that a
threads will be joined at the end of the scope → guarantees that the
thread will not outlive the borrowed data
- the spawn method requires `scope instead of `static → only requires t
  data to live as long as the scope itself

pub fn spawn<F, T>(&`scope self, f: F) → ScopedJoinHandle<`scope, T
where
    F: FnOnce() → T,
    F: Send + `scope,
    T: Send + `scope,
- any captured environment variables have to outlive `scope
```

▼ Working Rust Example 3 → Atomics

```
const N_THREADS: usize = 50;
const N_INCREMENTS: usize = 100000;

use std::{array, cell::Cell};
use std::sync::{atomic::{AtomicU64, Ordering}, Mutex, Arc};

static SHARED_NUM: AtomicU64 = AtomicU64::new(0);
```

```
fn my_thread() {
	for _ in 0..N_INCREMENTS {
		SHARED_NUM.fetch_add(1, Ordering::SeqCst);
	}
}

fn main() {
	let mut threads: [_; N_THREADS] = array::from_fn(|_| None);

	for i in 0..N_THREADS {
		threads[i] = Some(std::thread::spawn(my_thread())
	}

	for i in 0..N_THREADS {
		threads[i].take().unwrap().join();
	}

	println!("{}", SHARED_NUM.load(Ordering::SeqCst));
}

// atomics are really fast compared to mutex and scoped threads,
// but can only be used with atomic types
```

`move ||` closures

- Rust won't compile if we spawn a thread that may outlive the spawning function but borrows something owned by the spawning function
- `move ||` will convert any variables captured by reference (`&` or `&mut`) into references capture by value (ownership is transferred)
- Can still implement `Fn` or `FnMut` since compiler implements based on how the variables are used, not what is captured
- Often when we spawn threads, we want to use a `move ||` closure

▼ Move Closure Basic Example

```
fn main() {
	let my_string = String::from("HOWDY");

	// won't compile, because the spawned thread may outlive main,
	// which is where my_string is declared → we capture a borrow of
	// my_string, so the compiled struct for the closure will use a borro
	// so lifetimes matter
	// std::thread::spawn(|| {

	// this will compile, since we move my_string into the compiled clo
	// struct instead of a borrow
	std::thread::;spawn(move || {
		let capture_string = &my_string;
		println!("{capture_string}");
	}).join().unwrap();

	// this will not compile, since ownership was transferred
	// println!("{my_string}");
}

With just ||, the compiled struct would be something like
struct Compiled1<`a> {
	my_string: &`a String,
}

With move ||, the compiled struct would be something like
struct Compiled2 {
	my_string: String,
}

and we get this

impl FnOnce for Compiled2 {
	type Output = ();

	extern "rust-call" fn call_once(self, args: Args) → Self::Output {
		let capture_string = &self.my_string;
		println!("{capture_string}");
```

```
        }
    }
```

Mutex Poisoning → recall that `Mutex<T>.lock()` effectively returns a `Result<MutexGuard<'_, T>, PoisonError<MutexGuard<'_, T>>`

- If another user of some `mutex` panicked while holding the `mutex` (having the lock), the `mutex` will set itself to be poisoned and calling `mutex.lock()` in any threads will return a `PoisonError`
- `mutex.is_poisoned()` will return a bool signifying its poison status, and `mutex.clear_poison()` will set the poison status to false
    - Used to need to use `PoisonError.into_inner()` to retrieve the `MutexGuard` for use

`RwLock` is similar to `Mutex`, except it allows any number of readers, but only 1 writer at any given time

- `Mutex` is kind of like a `RwLock` that assumed you are always trying to be a writer when you lock it
- Can still be poisoned like `mutex`
- Can result in deadlocks → if we have a lot more reads than writes, the reads will keep getting access while the write has to wait and keeps getting pushed down the line
    - This happening will largely depends on the OS's implementation
    - Some OSs may make the reads wait too if there is a write waiting for a lock
    - A thread may try to get a read lock multiple times before unlocking when it uses a function call or recursion → split stuff into wrappers that get the lock then call a function with the main body and just use that main body function for stuff

More on `Send` and `Sync`

- A type is `Send` if it can safely have its ownership transferred from one thread to another

- A type is `Sync` if it is safe for multiple threads to concurrently access a reference to the same value
- `Mutex<T>` does not implement `Send` and `Sync` unless `T` does
  - The `Mutex` usually needs to be sent across threads so that the thread can obtain a lock on it
  - The `Mutex` usually needs to be concurrently accessed by reference so that threads can try to obtain a lock on it and wait if the lock is obtained elsewhere
  - if `T` was not `Send` and `Sync` the `mutex` itself would not be safe to send and access synchronously

▼ Week 8 — Channels and Rayon Parallelism

Channels can be thought of a FIFO structure (queue) that sends items from `Sender` to `Receiver`

- Only one `Receiver` allowed, but we can have multiple `Senders`
- We have `channel<T>() → (Sender<T>, Receiver<T>)`
  - Returns a tuple with a `Sender` and a `Receiver`
  - `Receiver.recv()` receives and consumes the first item in the channel, or waits until there is an item in the channel if the channel is empty
  - `Sender.send(item: T)` sends some item
  - `Sender.clone()` creates a clone to allow for multiple senders, and for senders to be transferred to threads through ownership
- We also have `sync_channel<T>(bound:usize) → (SyncSender<T>, Receiver<T>)`
  - Bounds the number of items that we can have in the FIFO structure
  - If the FIFO structure is full, `send()` will wait until there is more space → until a `SyncReceiver` consumes enough for there to be space again
  - If `bound == 0`, calling `send()` will wait until there is a `recv()`, and `recv()` will wait until there is a `send()`

▼ Simple Channel Example

```
fn main() {
    // create a channel for Strings between sender and receiver
    let (sender, receiver) = mspc::channel::<String>();

    // creating a channel only creates one sender and one receiver,
    // but Sender implements Clone
    {
        let sender = sender.clone();
        // need to clone since we use a move closure

        std::thread::spawn(move || {
            thread::sleep(Duration::from_secs(3));
            // send returns a Result<(), SendError<T>>
            // we get a SendError if receiver goes out of scope
            sender.send(String::from("Here you go 1!")).unwrap();
        });
    }

    {
        let sender = sender.clone();
        std::thread::spawn(move || {
            thread::sleep(Duration::from_secs(5));
            // send returns a Result<(), SendError<T>>
            // we get a SendError if receiver goes out of scope
            sender.send(String::from("Here you go 2!")).unwrap();
        });
    }

    // recv() will wait until there is something in the channel for it to
    // receive → in our case, it will wait 3 seconds
    let message = receiver.recv().unwrap();
    println!("Main received: {}", message);

    // receiving will consume (dequeue) the T, so the channel will be e
    // and recv() will wait 2 seconds for the next T
    // prints "Here you go 2!"      let message = receiver.recv().unwrap
    let message = receiver.recv().unwrap();
```

```
        println!("Main received: {}", message);
}
```

Note that the `crossbeam` crate is a popular crate for concurrency programming

- `crossbeam` has a multi-consumer multi-producer version of a channel

`rayon` is a data-parallelism crate that makes it easy to convert sequential computations into parallel ones

- Typically one core does one thing by default, and the computer may switch cores while completing it → most we can get is concurrency
- But we can explicitly split the work into chunks and have different cores do a chunk of the work → all the cores are working on the overall task at the same time → parallelism
- `std::Iterator` goes sequentially, `rayon::ParallelIterator` goes
    - `ParallelIterator` can do most things `Iterator` does, but it does it in parallel as well so its faster

▼ Simple rayon ParallelIterator Example

```
use rayon::iter::{IntoParallelIterator, ParallelIterator};

fn main() {
        // create a par iterator
        let sum: f64 = (1..10000000000u64).into_par_iter()
                .map(|n| n as f64)
                .sum();
        // with a typical iterator, one core goes from 1 to 10 billion
        // with par_iter, all cores take a portion of 1 to 10 billion and work
        // on that portion → e.g. if we had 16 cores, each one works on
        // a different 1/16 of the 1 to 10 billion values at the same time
}
```

▼ Week 9 — Unsafe Rust

Everything we have done so far is ‘safe’ Rust → we haven’t really used the `unsafe` keyword ourselves

- We have memory safety by default and no undefined behaviour
  - Things that compile without the `unsafe` keyword will be memory safe, and will never produce undefined behaviour → the compiler will know what will generally happen if its within its control
- We have opportunities for quality diagnostics and optimisation
- Low level of complexity relative to unsafe rust
- But sometimes safe Rust isn’t flexible enough

Compilers make assumptions to make optimisations

- If we have undefined behaviour, any and all behaviour becomes ‘correct’ behaviour according to the compiler

Unsafe Rust

- Sometimes ‘unsafe’ code is correct, but the Rust compiler won’t allow it
- Allows low-level control and raw access to the system
- Allows us to further optimise the small super low-level things
- But unsafe Rust threatens the benefits of safe Rust → memory safety, no undefined behaviour, etc.
- Unsafe Rust actually only lets us do 5 things safe Rust disallows
  - Dereferencing a raw pointer
  - Reading or writing a mutable or external static variable → mutable globals
  - Accessing a field of a union in cases other than to assign to it
  - Calling an unsafe (`unsafe fn`) function
  - Implementing an unsafe trait

NOTE Rust programs must still never cause undefined behaviour

- If something is declared `unsafe`, it gets the above powers, but should still definitely be safe to use → `unsafe` only means that undefined behaviour is on the programmer's fault
- It is the programmers responsibility when writing `unsafe` code to ensure that any safe code interacting with the `unsafe` code cannot trigger undefined behaviour
    - If unsafe code can be misused by safe code to exhibit undefined behaviour it is called unsound code, otherwise it is sound

A Rust program can then be viewed as a network between components of safe and unsafe Rust code, where the unsafe Rust code is just kind of treated as a blackbox

- The unsafe Rust code is treated as a safe abstraction of the unsafe code

Rust has (unstable) sanitisers to catch out errors/issues at a cost to performance

- We kinda only really use it for testing
- Unstable so has to run on nightly
- Miri is a Rust interpreter that will catch out undefined behaviour
    - `6991 cargo +nightly miri run`
    - `rustup +nightly component add miri`

▼ 'Proper' Ways of Interacting with Raw Pointers

```
fn foo<T>(ptr: *mut T, b: T) {
    // this may cause undefined behaviour
    // let c = *ptr;
    // *ptr = b;

    // *ptr will try to read and drop (in the reassignment) the data it
    // points to, which will cause undefined behaviour if ptr points to
    // uninitialised data
```

```
    // this is the proper 'safe unsafe' way
    let c = unsafe { ptr.read() };
    unsafe { ptr.write(b) };
}
```

▼ MyVec Example → **NOTE: VERY DODGY, PROB WON'T PASS MIRI TESTS**

```
// whenever we have some type of unsafe abstraction, we typically wra
// in its own module, so that we can toggle access modifiers to
// control what users have access to

mod my_vec {
    use std::alloc::{Layout, self, realloc};

    pub struct MyVec {
        // raw pointer → *const signals we just reading, *mut signals
        // modifying T → note we can always do ((&val as *const T) a
        ptr: *mut T,
        len: usize,
        cap: usize,
    }

    impl<T> MyVec<T> {
        const INITIAL_CAP: usize = 8;

        pub fn new() → Self {
            Self {
                ptr: std::ptr::null_mut(),
                size: 0,
                cap: 0,
            }
        }

        pub fn push(&mut self, value: T) {
            // realloc if out of space
            if self.len == self.cap {
                self.expand_cap();
```

```
        }

        // SAFETY: If len >= to cap, we would've expanded cap a
        // so len will always be a valid index within ptr
        let tmp_ptr = unsafe { self.pointer_to_elem(self.len) };

        // SAFETY: The pointer returned by self.pointer_to_elem i
        // valid pointer within our allocation, so we're safe to writ
        // on this address
        unsafe { tmp_ptr.write(value) };
        self.len += 1;
    }

    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            // we checked that the vec is non-empty, and self.si
            // represents the last valid member of the vec due to
            // -= 1 operation
            self.len -= 1;
            let tmp_ptr = unsafe { self.pointer_to_elem(self.len) }

            // SAFETY: same as above, and it is fine to take own
            // since we remove it from the vec as well
            let value = nsafe { tmp_ptr.read() };

            Some(value);
        }
    }

    pub fn get(&self, index: usize) -> Option<&T> {
        if index >= self.size {
            None
        } else {
            // SAFETY: We checked that the index provided repr
            // valid element in the vec
            let tmp_ptr = unsafe { self.pointer_to_elem(self.len) }
```

```
            // SAFETY: same as above, and it is safe to borrow s
            // is tired to the lifetime of &self
            let value = unsafe { &*tmp_ptr };
            // we can deref the raw pointer, since we know it is
            // element, and are not reassigining it (nothing gets

            Some(&value);
        }
    }

    // doubles cap of MyVec, or allocates INITIAL_CAP if unalloca
    fn expand_cap(&mut self) {
        if self.cap == 0 {
            let layout = Self::layout_for(INITIAL_CAP);

            // when we write unsafe code, we have to declare (
            // what is expected of users such that this would no
            // undefined behaviour despite being unsafe/unsour

            // SAFETY: Layout has non-zero size
            self.ptr = unsafe { alloc::alloc(layout) } as _;
            // note the `as _` tells the compiler to cast to whatev
            // is expected
            self.cap = INITIAL_CAP;
        } else {
            let new_cap = self.cap.checked_mul(2).expect("cap
            let old_layout = Self::layout_for(self.cap);
            let new_layout = Self::layout_for(new_cap);

            // SAFETY: we allocated self.ptr ourselves with a lay
            // equal to old_layout, and the new_layout is > 0 byt
            // size, and we assume it doesn't overflow
            // → don't actually assume in production code
            self.ptr = unsafe { alloc::realloc(self.ptr as _, old_lay
        }
    }
```

```
            fn layout_for(n: usize) -> Layout {
                // Layout helps us determine how many bytes, and what
                // are needed (some types need alignments in memory a
                // assume its fine for sake of example
                Layout::array::<T>(n).unwrap();
            }

            /// # SAFETY
            ///
            /// `index` must be a valid index within our current ptr
            ///
            /// since this is an unsafe function, it must be used within
            /// an unsafe {} block
            fn unsafe pointer_to_elem(&self, index: usize) -> *mut T {
                unsafe { self.ptr.add(index) };
            }
        }

        impl<T> Drop for MyVec<T> {
            fn drop(&mut self) {
                // drop each element
                for index in 0..self.len {
                    unsafe { std::ptr::;drop_in_place(self.pointer_to_elem
                }

                let layout = Self::layout_for(self.cap);

                // SAFETY: We allocated this pointer ourselves using the
                // provided
                unsafe { alloc::dealloc(self.ptr, layout) };
            }
        }
    }
```

▼ Week 9 — Foreign Function Interface

Foreign Function Interface (FFI) refers to how we communicate between languages

- e.g. bridging Rust to C
- Existing interfaces/methods: → they're slow and not always available
  - Spawn a process from a program in the other language as a child process
  - Using a file system to communicate → pass `json` or `xml` files between programs to communicate data
  - Using a raw socket to communicate over a network → or even using `HTTP`
- **NOTE**: Interacting with another language is always unsafe

▼ Using `C` FFI `libcurl` → **ALSO SUPER DODGY**

Rust program using some simple functionality of `C` `libcurl` through the header file (interface)

- https://github.com/curl/curl/blob/master/include/curl/curl.h
- We are trying to emulate the sample here https://curl.se/libcurl/c/simple.html

```
type CURL = std::ffi::c_void;

#[repr(C)].    // lay the CURLoption type out in the same way as a C
enum CURLoption {
        CURLOPT_URL = 10002,
}

#[repr(C)]
enum CURLcode {
        ... imagine lines 516 to 646 of the .h file ...
}

// declare that we are using external C code from a library called curl
#[link(name = "curl")]
extern "C" {
        // import signatures of functions we will use in Rust syntax
```

```
        // CURL *curl_easy_init();
        fn curl_easy_init() -> *mut CURL;

        // CURLcode curl_easy_setopt(CURL *handle, CURLoption option,
        fn curl_easy_setopt(handle: *mut CURL, option: CURLoption, parar

        // CURLcode curl_easy_perform(CURL *easy_handle);
        fn curl_easy_perform(easy_handle: *mut CURL) -> CURLcode;

        // void curl_easy_cleanup(CURL *handle);
        fn curl_easy_cleanup(handle: *mut CURL);
}
// Rust bindgen can automatically generate the FFI bindings to C ^^^
// https://github.com/rust-lang/rust-bindgen

fn main() {
        unsafe {
                let curl = curl_easy_init();

                // if it expects a C string, we need to give it a C string
                let string = CString::new("https://insou.dev/").unwrap();
                curl_easy_setopt(curl, CURLoption::CURLOPT_URL, string.as_

                curl_esy_perform(curl);
                curl_easy_cleanup(curl);
        }
}
```

```
6991 rslogo sth.lg output.svg 200 200
```

- Runs the sample soln for ass1