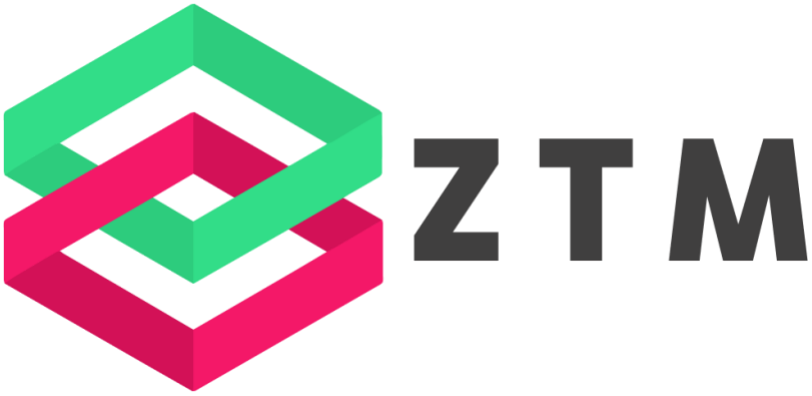

RUST CHEAT SHEET

JAYSON LENNON

V1.01

HEEELLLOOOOO!

I'm Andrei Neagoie, Founder and Lead Instructor of the Zero To Mastery Academy.

After working as a Senior Software Developer over the years, I now dedicate 100% of my time to teaching others valuable software development skills, help them break into the tech industry, and advance their careers.

In only a few years, **over 750,000 students** around the world have taken Zero To Mastery courses and many of them are now working at top tier companies like Apple, Google, Amazon, Tesla, IBM, Facebook, and Shopify, just to name a few.

This cheat sheet, created by our Rust instructor (Jayson Lennon) provides you with the key Rust concepts that you need to know and remember.

If you want to not only learn Rust but also get the exact steps to build your own projects and get hired as a Frontend or Backend Developer, then check out our Career Paths.

Happy Coding!
Andrei

Founder & Lead Instructor, Zero To Mastery
Andrei Neagoie

P.S. I also recently wrote a book called Principles For Programmers. You can download the first five chapters for free here.

Rust Cheat Sheet

Contents

Control Flow

if,
if let,
match,
while,
while let,
for,
loop

Structures

impl Blocks,
Matching on Structures,
Destructuring Assignment

Enumerations

Match on Enums

Tuples

Arrays

Slices

Slice Patterns

Option

Result / Error Handling

Question Mark Operator

Iterator

Ownership & Borrowing

Lifetimes

Traits

Associated Types,
Trait Objects,
Default,
From/Into

Generics

Operator Overloading

Index

Concurrent Programming

Threads,
Channels,
Mutex,
Async

Modules

Inline Modules,
Modules as Files

Testing

Test Module,
Doctests

Standard Library Macros

Standard Library Derive Macros

Declarative Macros

Valid Positions,
Fragment Specifiers,
Repetitions,
Example Macros,
Macro Notes

Rustup

`Rustup` is used to install and manage Rust toolchains. Toolchains are complete installations of Rust compiler and tools.

Click for more details

Command	Description
`rustup show`	Show currently installed & active toolchains
`rustup update`	Update all toolchains
`rustup default TOOLCHAIN`	Set the default toolchain
`rustup component list`	List available components
`rustup component add NAME`	Add a component (like Clippy or offline docs)
`rustup target list`	List available compilation targets
`rustup target add NAME`	Add a compilation target

Cargo

`Cargo` is a tool used to build and run Rust projects.

Click for more details

Command	Description
`cargo init`	Create a new binary project
`cargo init --lib`	Create a new library project
`cargo check`	Check code for errors
`cargo clippy`	Run code linter (use `rustup component add clippy` to install)
`cargo doc`	Generate documentation
`cargo run`	Run the project
`cargo --bin NAME`	Run a specific project binary
`cargo build`	Build everything in debug mode
`cargo build --bin NAME`	Build a specific binary in debug mode
`cargo build --release`	Bulld everything in release mode
`cargo build --target NAME`	Build for a specific target

Command	Description
`cargo --explain CODE`	Detailed information regarding an compiler error code
`cargo test`	Run all tests
`cargo test TEST_NAME`	Run a specific test
`cargo test --doc`	Run doctests only
`cargo test --examples`	Run tests for example code only
`cargo bench`	Run benchmarks

Documentation Comments

Rust has support for doc comments using the `rustdoc` tool. This tool can be invoked using `cargo doc` and it will generate HTML documentation for your crate. In addition to generating documentation, the tool will also test your example code.

Click for more details

```
/// Documentation comments use triple slashes.
///
/// They are parsed in markdown format, so things
/// like headers, tables, task lists, and links to other types
/// can be included in the documentation.
///
/// Example code can also be included in doc comments with
/// three backticks (`). All example code in documentation is
/// tested with `cargo test` (this only applies to library crates).
fn is_local_phone_number(num: &str) -> bool {
    use regex::Regex;
    let re = Regex::new(r"[0-9]{3}-[0-9]{4}").unwrap();
    re.is_match(num)
}
```

Operators

Mathematical

Operator	Description
`+`	add
`-`	subtract

Operator	Description
`*`	multiply
`/`	divide
`%`	remainder / modulo
`+=`	add and assign
`-=`	subtract and assign
`*=`	multiply and assign
`/=`	divide and assign
`%=`	remainder / modulo and assign

Comparison

Operator	Description
`==`	equal
`!=`	not equal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal

Logical

Operator	Description
`&&`	and
`||`	or
`!`	not

Bitwise

Operator	Description
`&`	`and`
`|`	`or`

Operator	Description
`^`	`xor`
`<<`	left shift
`>>`	right shift
`&=`	`and` and assign
`|=`	`or` and assign
`^=`	`xor` and assign
`<<=`	left shift and assign
`>>=`	right shift and assign

Primitive Data Types

Signed Integers

Type	Default	Range
i8	0	-128..127
i16	0	-32768..32767
i32	0	-2147483648..2147483647
i64	0	-9223372036854775808..9223372036854775807
i128	0	min: -170141183460469231731687303715884105728
i128	0	max: 170141183460469231731687303715884105727
isize	0	<pointer size on target architecture>

Unsigned Integers

Type	Default	Range
u8	0	0..255
u16	0	0..65535
u32	0	0..4294967295
u64	0	0..18446744073709551615
u128	0	0..340282366920938463463374607431768211455

Type	Default	Range
usize	0	<pointer size on target architecture>

Floating Point Numbers

Type	Default	Notes
f32	0	32-bit floating point
f64	0	64-bit floating point

Strings / Characters

Type	Notes
char	Unicode scalar value. Create with single quotes `''`
String	UTF-8-encoded string
&str	Slice into `String` / Slice into a static `str`. Create with double quotes `""` or `r#""#` for a raw mode (no escape sequences, can use double quotes)
OsString	Platform-native string
OsStr	Borrowed `OsString`
CString	C-compatible nul-terminated string
CStr	Borrowed `CString`

Other

Type	Notes
bool	`true` or `false`
unit	`()` No value / Meaningless value
fn	Function pointer
tuple	Finite length sequence
array	Fixed-sized array
slice	Dynamically-sized view into a contiguous sequence

Declarations

Variables

```
// `let` will create a new variable binding
let foo = 1;
// bindings are immutable by default
foo = 2;              // ERROR: cannot assign; `foo` not mutable

let mut bar = 1;      // create mutable binding
bar = 2;              // OK to mutate

let baz = 'a';        // use single quotes to create a character
let baz = "ok";       // use double quotes to create a string
// variables can be shadowed, so these lines have been valid
let baz = 42;         // `baz` is now an integer; 'a' and "ok" no longer accessible

// Rust infers types, but you can use annotations as well
let foo: i32 = 50;    // set `foo` to i32
let foo: u8 = 100;    // set `foo` to u8
// let foo: u8 = 256; // ERROR: 256 too large to fit in u8

let bar = 14.5_f32;   // underscore can be used to set numeric type...
let bar = 99_u8;
let bar = 1_234_567;  // ...and also to make it easier to read long numbers

let baz;              // variables can start uninitialized, but they must be set before us
age
// let foo = baz;     // ERROR: possibly uninitialized.
baz = 0;              // `baz` is now initialized
// baz = 1;           // ERROR: didn't declare baz as mutable

// naming convention:
let use_snake_case_for_variables = ();
```

Constants

```
// `const` will create a new constant value
const PEACE: char = '☮';    // type annotations are required
const MY_CONST: i32 = 4;    // naming conventions is SCREAMING_SNAKE_CASE

// const UNINIT_CONST: usize;  // ERROR: must have initial value for constants

// use `once_cell` crate if you need lazy initialization of a constant
use once_cell::sync::OnceCell;
const HOME_DIR: OnceCell<String> = OnceCell::new();
// use .set to set the value (can only be done once)
HOME_DIR.set(std::env::var("HOME").expect("HOME not set"));
```

```
// use .get to retrieve the value
HOME_DIR.get().unwrap();
```

Type Aliases

Type aliases allow long types to be represented in a more compact format.

```
// use `type` to create a new type alias
type Foo = Bar;
type Miles = u64;
type Centimeters = u64;
type Callbacks = HashMap<String, Box<dyn Fn(i32, i32) -> i32>>;

struct Contact {
    name: String,
    phone: String,
}
type ContactName = String;
// type aliases can contain other type aliases
type ContactIndex = HashMap<ContactName, Contact>;

// type aliases can be used anywhere a type can be used
fn add_contact(index: &mut ContactIndex, contact: Contact) {
    index.insert(contact.name.to_owned(), contact);
}

// type aliases can also contain lifetimes ...
type BorrowedItems<'a> = Vec<&'a str>;
// ... and also contain generic types
type GenericThings<T> = Vec<Thing<T>>;
```

New Types

"New Types" are simply existing types wrapped up in a new type. This can be used to implement traits for types that are defined outside of your crate and can be used for stricter compile-time type checking.

```
// This block uses type aliases instead of New Types:
{
    type Centimeters = f64;
    type Kilograms = f64;
    type Celsius = f64;

    fn add_distance(a: Centimeters, b: Centimeters) -> Centimeters {
        a + b
    }
```

```
    fn add_weight(a: Kilograms, b: Kilograms) -> Kilograms {
        a + b
    }
    fn add_temperature(a: Celsius, b: Celsius) -> Celsius {
        a + b
    }

    let length = 20.0;
    let weight = 90.0;
    let temp = 27.0;

    // Since type aliases are the same as their underlying type,
    // it's possible to accidentally use the wrong data as seen here:
    let distance = add_distance(weight, 10.0);
    let total_weight = add_weight(temp, 20.0);
    let new_temp = add_temperature(length, 5.0);
}

// This block uses new types instead of type aliases:
{
    // create 3 tuple structs as new types, each wrapping f64
    struct Centimeters(f64);
    struct Kilograms(f64);
    struct Celsius(f64);

    fn add_distance(a: Centimeters, b: Centimeters) -> Centimeters {
        // access the field using .0
        Centimeters(a.0 + b.0)
    }
    fn add_weight(a: Kilograms, b: Kilograms) -> Kilograms {
        Kilograms(a.0 + b.0)
    }
    fn add_temperature(a: Celsius, b: Celsius) -> Celsius {
        Celsius(a.0 + b.0)
    }

    // the type must be specified
    let length = Centimeters(20.0);
    let weight = Kilograms(90.0);
    let temp = Celsius(27.0);

    let distance = add_distance(length, Centimeters(10.0));
    let total_weight = add_weight(weight, Kilograms(20.0));
    let new_temp = add_temperature(temp, Celsius(5.0));

    // using the wrong type is now a compiler error:
    // let distance = add_distance(weight, Centimeters(10.0));
    // let total_weight = add_weight(temp, Kilograms(20.0));
    // let new_temp = add_temperature(length, Celsius(5.0));
}
```

Functions

Functions are fundamental to programming in Rust. Signatures require type annotations for all input parameters and all output types. Functions evaluate their bodies as an expression, so data can be returned without using the `return` keyword.

Click for more details

```
// use the `fn` keyword to create a function
fn func_name() { /* body */ }

// type annotations required for all parameters
fn print(msg: &str) {
    println!("{msg}");
}

// use -> to return values
fn sum(a: i32, b: i32) -> i32 {
    a + b   // `return` keyword optional
}
sum(1, 2);  // call a function

// `main` is the entry point to all Rust programs
fn main() {}

// functions can be nested
fn outer() -> u32 {
    fn inner() -> u32 { 42 }
    inner()         // call nested function & return the result
}

// use `pub` to make a function public
pub fn foo() {}

// naming convention:
fn snake_case_for_functions() {}
```

Closures

Closures are similar to functions but offer additional capabilities. They capture (or "close over") their environment which allows them to capture variables without needing to explicitly supply them via parameters.

Type	Notes
Fn	Closure can be called any number of times

Type	Notes
FnMut	Closure can mutate values
FnOnce	Closure can only be called one time

Click for more details

```
// use pipes to create closures
let hello = || println!("hi");
// parameters to closures go between the pipes
let msg = |msg| println!("{msg}");
// closures are called just like a function
msg("hello");

// type annotations can be provided...
let sum = |a: i32, b: i32| -> i32 { a + b };
// ...but they are optional
let sum = |a, b| a + b;
let four = sum(2, 2);
assert_eq!(four, 4);

// closures can be passed to functions using the `dyn` keyword
fn take_closure(clos: &dyn Fn()) {
    clos();
}
let hello = || println!("hi");
take_closure(&hello);

// use the `move` keyword to move values into the closure
let hi = String::from("hi");
let hello = move || println!("{hi}");
// `hi` can no longer be used because it was moved into `hello`
```

Control Flow

Control flow allows code to branch to different sections, or to repeat an action multiple times. Rust provides multiple control flow mechanisms to use for different situations.

if

`if` checks if a condition evalutes to `true` and if so, will execute a specific branch of code.

```
if some_bool { /* body */ }
```

```
if one && another {
    // when true
} else {
    // when false
}

if a || (b && c) {
    // when one of the above
} else if d {
    // when d
} else {
    // none are true
}

// `if` is an expression, so it can be assigned to a variable
let (min, max, num) = (0, 10, 12);
let num = if num > max {
    max
} else if num < min {
    min
} else {
    num
};
assert_eq!(num, 10);
```

if let

`if let` will destructure data only if it matches the provided pattern. It is commonly used to operate on data within an `Option` or `Result`.

Click for more details

```
let something = Some(1);
if let Some(inner) = something {
    // use `inner` data
    assert_eq!(inner, 1);
}

enum Foo {
    Bar,
    Baz
}
let bar = Foo::Bar;
if let Foo::Baz = bar {
    // when bar == Foo::Baz
} else {
    // anything else
}

// `if let` is an expression, so it can be assigned to a variable
```

```
let maybe_num = Some(1);
let definitely_num = if let Some(num) = maybe_num { num } else { 10 };
assert_eq!(definitely_num, 1);
```

match

`Match` provides *exhaustive* pattern matching. This allows the compiler to ensure that every possible case is handled and therefore reduces runtime errors.

Click for more details

```
let num = 0;
match num {
    // ... on a single value
    0 => println!("zero"),
    // ... on multiple values
    1 | 2 | 3 => println!("1, 2, or 3"),
    // ... on a range
    4..=9 => println!("4 through 9"),
    // ... with a guard
    n if n >= 10 && n <= 20 => println!("{n} is between 10 and 20"),
    // ... using a binding
    n @ 21..=30 => println!("{n} is between 21 and 30"),
    // ... anything else
    _ => println!("number is ignored"),
}

// `match` is an expression, so it will evaluate and can be assigned
let num = 0;
let msg = match num {
    0 => "zero",
    1 => "one",
    _ => "other",
};
assert_eq!(msg, "zero");
```

while

`While` will continually execute code as long as a condition is true.

```
// must be mutable so it can be modified in the loop
let mut i = 0;

// as long as `i` is less than 10, execute the body
while i < 10 {
    if i == 5 {
        break;     // completely stop execution of the loop
```

```
    }
    if i == 8 {
        continue;  // stop execution of this iteration, restart from `while`
    }

    // don't forget to adjust `i`, otherwise the loop will never terminate
    i += 1;
}

// `while` loops can be labeled for clarity and must start with single quote (')
let mut r = 0;
let mut c = 0;
// label named 'row
'row: while r < 10 {
    // label named 'col
    'col: while c < 10 {
        if c == 3 {
            break 'row;     // break from 'row, terminating the entire loop
        }
        if c == 4 {
            continue 'row;  // stop current 'col iteration and continue from 'row
        }
        if c == 5 {
            continue 'col;  // stop current 'col iteration and continue from 'col
        }
        c += 1;
    }
    r += 1;
}
```

while let

`while let` will continue looping as long as a pattern match is successful. The `let` portion of `while let` is similar to `if let`: it can be used to destructure data for utilization in the loop.

```
let mut maybe = Some(10);
// if `maybe` is a `Some`, bind the inner data to `value` and execute the loop
while let Some(value) = maybe {
    println!("{maybe:?}");
    if value == 1 {
        // loop will exit on next iteration
        // because the pattern match will fail
        maybe = None;
    } else {
        maybe = Some(value - 1);
    }
}
```

for

Rust's `for` loop is to iterate over collections that implement the `Iterator` trait.

```
// iterate through a collection
let numbers = vec![1, 2, 3];
for num in numbers {
    // values are moved into this loop
}

// .into_iter() is implicitly called when using `for`
let numbers = vec![1, 2, 3];
for num in numbers.into_iter() {
    // values are moved into this loop
}

// use .iter() to borrow the values
let numbers = vec![1, 2, 3];
for num in numbers.iter() {
    // &1
    // &2
    // &3
}

// ranges can be used to iterate over numbers
for i in 1..3 {     // exclusive range
    // 1
    // 2
}
```

loop

The `loop` keyword is used for infinite loops. Prefer using `loop` instead of `while` when you specifically want to loop endlessly.

```
loop { /* forever */ }

// loops can be labled
'outer: loop {
    'inner: loop {
        continue 'outer;    // immediately begin the next 'outer loop
        break 'inner;       // exit out of just the 'inner loop
    }
}

// loops are expressions
let mut iterations = 0;
let total = loop {
```

```
    iterations += 1;
    if iterations == 5 {
        // using `break` with a value will evaluate the loop
        break iterations;
    }
};
// total == 5
```

Structures

Structures allow data to be grouped into a single unit.

Click for more details

```
struct Foo;          // define a structure containing no data
let foo = Foo;       // create a new `Foo`

struct Dimension(i32, i32, i32);     // define a "tuple struct" containing 3 data points
let container = Dimension(1, 2, 3);  // create a new `Dimension`
let (w, d, h) = (container.0, container.1, container.2);
// w, d, h, now accessible

// define a structure containing two pieces of information
struct Baz {
    field_1: i32,   // an i32
    field_2: bool,  // a bool
}
// create a new `Baz`
let baz = Baz {
    field_1: 0,     // all fields must be defined
    field_2: true,
};
```

impl Blocks

`impl` blocks allow functionality to be associated with a structure or enumeration.

Click for more details

```
struct Bar {
    inner: bool,
}

// `impl` keyword to implement functionality
impl Bar {
    // `Self` is an alias for the name of the structure
    pub fn new() -> Self {
```

```
        // create a new `Bar`
        Self { inner: false }
    }
    // `pub` (public) functions are accessible outside the module
    pub fn make_bar() -> Bar {
        Bar { inner: false }
    }

    // use `&self` to borrow an instance of `Bar`
    fn is_true(&self) -> bool {
        self.inner
    }

    // use `&mut self` to mutably borrow an instance of `Bar`
    fn make_true(&mut self) {
        self.inner = true;
    }

    // use `self` to move data out of `Bar`
    fn into_inner(self) -> bool {
        // `Bar` will be destroyed after returning `self.inner`
        self.inner
    }
}

let mut bar = Bar::new();           // make a new `Bar`
bar.make_true();                    // change the inner value
assert_eq!(bar.is_true(), true);    // get the inner value
let value = bar.into_inner();       // move the inner value out of `Bar`

// `bar` was moved into `bar.into_inner()` and can no longer be used
```

Matching on Structures

Structures can be used within `match` expressions and all or some of a structure's values can be matched upon.

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    // match when ...
    // ... x == 0 && y == 0
    Point { x: 0, y: 0 } => (),
    // ... x == 0 and then ignore y
    Point { x: 0, .. } => (),
```

```
    // ... y == 0 and then ignore x
    Point { y: 0, .. } => (),
    // ... x == 0 and then capture y while checking if y == 0; bind y
    Point { x: 0, y } if y == 2 => println!("{y}"),
    // ... the product of x and y is 100; bind x and y
    Point { x, y } if x * y == 100 => println!("({x},{y})"),
    // ... none of the above are satisfied; bind x and y
    Point { x, y } => println!("({x},{y})"),
    // ... none of the above are satisfied while also ignoring x and y
    // (this will never match because `Point {x, y}` matches everything)
    _ => (),
}
```

Destructuring Assignment

Destructuring assignment allows structure fields to be accessed based on patterns. Doing so moves data out of the structure.

```
struct Member {
    name: String,
    address: String,
    phone: String,
}

let m = Member {
    name: "foo".to_string(),
    address: "bar".to_string(),
    phone: "phone".to_string(),
};

// move `name` out of the structure; ignore the rest
let Member { name, .. } = m;
// move `name` out of the structure; bind to `id`; ignore the rest
let Member { name: id, .. } = m;
// move `name` out of the structure; bind to `id`
let id = m.name;
```

Enumerations

Rust enumerations can have multiple choices, called variants, with each variant optionally containing data. Enumerations can only represent one variant at a time and are useful for storing data based on different conditions. Example use cases include messages, options to functions, and different types of errors.

`impl` blocks can also be used on enumerations.

Click for more details

```
// a structure wrapping a usize which represents an error code
struct ErrorCode(usize);

// enumerations are created with the `enum` keyword
enum ProgramError {
    // a single variant
    EmptyInput,
    // a variant containing String data
    InvalidInput(String),
    // a variant containing a struct
    Code(ErrorCode),
}

// create one ProgramError of each variant
let empty = ProgramError::EmptyInput;
let invalid = ProgramError::InvalidInput(String::from("whoops!"));
let error_code = ProgramError::Code(ErrorCode(9));
```

Match on Enums

```
enum ProgramError {
    // a single variant
    EmptyInput,
    // a variant containing String data
    InvalidInput(String),
    // a variant containing a struct
    Code(ErrorCode),
}

let some_error: ProgramError = some_fallible_fn();

match some_error {
    // match on the ...
    // ... EmptyInput variant
    ProgramError::EmptyInput => (),
    // ... InvalidInput variant only when the String data is == "123"; bind `input`
    ProgramError::InvalidInput(input) if input == "123" => (),
    // ... InvalidInput variant containing any other String data not capture above; bind `
input`
    ProgramError::InvalidInput(input) => (),
    // ... Code variant having an ErrorCode of 1
    ProgramError::Code(ErrorCode(1)) => (),
    // ... Code variant having any other ErrorCode not captured above; bind `other`
    ProgramError::Code(other) => (),
}

// enumeration variant names can be brought into scope with `use` ...
```

```
use ProgramError::*;
match some_error {
    // ... only need to specify the variant names now
    EmptyInput => (),
    InvalidInput(input) if input == "123" => (),
    InvalidInput(input) => (),
    Code(ErrorCode(1)) => (),
    Code(other) => (),
}
```

Tuples

Tuples offer a way to group unrelated data into an anonymous data type. Since tuples are anonymous, try to keep the number of elements limited to avoid ambiguities.

```
// create a new tuple named `tup` containing 4 pieces of data
let tup = ('a', 'b', 1, 2);
// tuple members are accessed by index
assert_eq!(tup.0, 'a');
assert_eq!(tup.1, 'b');
assert_eq!(tup.2, 1);
assert_eq!(tup.3, 2);

// tuples can be destructured into individual variables
let (a, b, one, two) = (tup.0, tup.1, tup.2, tup.3);
let (a, b, one, two) = ('a', 'b', 1, 2);
// a, b, one, two now can be used as individual variables

// tuple data types are just existing types surrounded by parentheses
fn double(point: (i32, i32)) -> (i32, i32) {
    (point.0 * 2, point.1 * 2)
}
```

Arrays

Arrays in Rust are fixed size. Most of the time you'll want to work with a `slice` or `Vector`, but arrays can be useful with fixed buffer sizes.

Click for more details

```
let array = [0; 3]; // array size 3, all elements initialized to 0
let array: [i32; 5] = [1, 2, 3, 4, 5];
let slice: &[i32] = &array[..];
```

Slices

Slices are *views* into a chunk of contiguous memory. They provide convenient high-performance operations for existing data.

Click for more details

```
let mut nums = vec![1, 2, 3, 4, 5];
let num_slice = &mut nums[..];  // make a slice out of the Vector
num_slice.first();              // Some(&1)
num_slice.last();               // Some(&5)
num_slice.reverse();            // &[5, 4, 3, 2, 1]
num_slice.sort();               // &[1, 2, 3, 4, 5]

// get a view of "chunks" having 2 elements each
let mut chunks = num_slice.chunks(2);
chunks.next(); // Some(&[1, 2])
chunks.next(); // Some(&[3, 4])
chunks.next(); // Some(&[5])
```

Slice Patterns

Slice patterns allow matching on slices given specific conditions while also ensuring no indexing errors occur.

```
let chars = vec!['A', 'B', 'C', 'D'];

// two ways to create a slice from a Vector
let char_slice = &chars[..];
let char_slice = chars.as_slice();

match char_slice {
    // match ...
    // ... the first and last element. minimum element count == 2
    [first, .., last] => println!("{first}, {last}"),
    // ... one and only one element
    [single] => println!("{single}"),
    // ... an empty slice
    [] => (),
}

match char_slice {
    // match ...
    // ... the first two elements. minimum elements == 2
    [one, two, ..] => println!("{one}, {two}"),
    // ... the last element. minimum elements == 1
    [.., last] => println!("{last}"),
```

```
    // ... an empty slice
    [] => (),
}

let nums = vec![7, 8, 9];
match nums.as_slice() {
    // match ...
    // First element only if element is == 1 or == 2 or == 3,
    // with remaining slice bound to `rest`. minimum elements == 1
    [first @ 1..=3, rest @ ..] => println!("{rest:?}"),
    // ... one element, only if == 5 or == 6
    [single] if single == &5 || single == &6 => (),
    // ... two and only two elements
    [a, b] => (),
    // Two-element slices are captured in the previous match
    // arm, so this arm will match either:
    //   * One element
    //   * More than two elements
    [s @ ..] => println!("one element, or 2+ elements {s:?}"),
    // ... empty slice
    [] => (),
}
```

Option

Rust doesn't have the concept of `null`, but the `Option` type is a more powerful alternative. Existence of a value is `Some` and abscense of a value is `None`. Semantically, `Option` is used when there is the possibility of some data not existing, such as "no search results found".

Click for more details

```
// Option in the standard library is defined as simply a generic enumeration:
enum Option<T> {
    Some(T),    // data exists
    None,       // no data exists
}
// an Option's variants are available for use without specifying Option::Some / Option::No
ne

// create an Option containing usize data
let maybe_number: Option<usize> = Some(1);
// add 1 to the data, but only if the option is `Some` (this is a no-op if it is `None`)
let plus_one: Option<usize> = maybe_number.map(|num| num + 1);

// `if let` can be used to access the inner value of an Option
if let Some(num) = maybe_number {
    // use `num`
```

```
} else {
    // we have `None`
}

// Options can be used with `match`
match maybe_number {
    // match when ...
    // ... there is some data and it is == 1
    Some(1) => (),
    // ... there is some data not covered above; bind the value to `n`
    Some(n) => (),
    // ... there is no data
    None => (),
}

// since `if let` is an expression, we can use it to conditionally destructure an Option
let msg = if let Some(num) = maybe_number {
    format!("We have a {num}")
} else {
    format!("We have None")
};
assert_eq!(msg, "We have a 1");

// combinators can be used to easily manipulate Options
let maybe_number = Some(3);
let odd_only = maybe_number      // take `maybe_number`
    .and_then(|n| Some(n * 3))   // then access the inner value, multiply by 3, and make a
new Option
    .map(|n| n - 1)              // then take the inner value and subtract 1
    .filter(|n| n % 2 == 1)      // then if the inner value is odd, keep it
    .unwrap_or(1);               // then unwrap the inner value if it exists; otherwise us
e 1
assert_eq!(odd_only, 1);

// same as above but with named functions instead of inline closures
let maybe_number = Some(4);
let odd_only = maybe_number      // take `maybe_number`
    .and_then(triple)            // then run the `triple` function with the inner value
    .map(minus_one)              // then transform the inner value with the `minus_one` fu
nction
    .filter(is_odd)              // then filter the value using the `is_odd` function
    .unwrap_or(1);               // then unwrap the inner value if it exists; otherwise us
e 1
assert_eq!(odd_only, 11);

fn triple(n: i32) -> Option<i32> {
    Some(n * 3)
}

fn minus_one(n: i32) -> i32 {
    n - 1
}

fn is_odd(n: &i32) -> bool {
```

```
    n % 2 == 1
}
```

Result / Error Handling

Rust doesn't have exceptions. All errors are handled using a return value and the `Result` type. Helper crates are available to automatically generate errors and make propagation easier:

- `anyhow` / `eyre` / `miette`: use in binary projects to easily propagate any type of error
- `thiserror`: use in library projects to easily create specific error types

Click for more details

```
// Result in the standard library is defined as simply a generic enumeration:
enum Result<T, E> {
    Ok(T),      // operation succeeded
    Err(E),     // operation failed
}
// a Results's variants are available for use without specifying Result::Ok / Result::Err

// create a Result having a success type of i32 and an error type of String
let maybe_number: Result<i32, String> = Ok(11);

// Combinators can be used to easily manipulate Results. The following sequence
// transformed the inner value by multiplying it by 3 if it is an Ok. If
// `maybe_number` is an Err then the error returned will be the supplied String.
let maybe_number = maybe_number
    .map(|n| n * 3)
    .map_err(|e| String::from("don't have a number"));

// We can use `if let` to conditionally destructure a Result.
// Here we are specifically looking for an error to report.
if let Err(e) = maybe_number.as_ref() {
    eprintln!("error: {e}");
}

// Results and Options can be changed back and forth using `.ok`
let odd_only = maybe_number      // take `maybe_number`
    .ok()                        // transform it into an Option
    .filter(|n| n % 2 == 1)      // apply a filter
    .ok_or_else(||               // transform the Option back into a Result
        String::from("not odd!") // if the Option is None, use this String for the Err
    );

// `match` is commonly used when working with Results
match odd_only {
    Ok(odd) => println!("odd number: {odd}"),
```

```
    Err(e) => eprintln!("error: {e}"),
}
```

Question Mark Operator

`Results` can be verbose and cumbersome to use when there are multiple failure points. The question mark operator (`?`) makes `Result` easier to work with by doing one of two things:

- if `Ok`: unwrap the value
- if `Err`: map the error to the specified `Err` return type and then return from the function

```
use std::error::Error;
use std::fs::File;
use std::io::{self, Read};
use std::path::Path;

// This function has 3 failure points and uses the question mark operator
// to automatically propagate appropriate errors.
fn read_num_using_questionmark(path: &Path) -> Result<u8, Box<dyn Error>> {
    // make a buffer
    let mut buffer = String::new();

    // Using the `?` will automatically give us an open file on
    // success, and automatically return a `Box<dyn Error>` on failure.
    let mut file = File::open(path)?;

    // We aren't concerned about the return value for this function,
    // however we still need to handle the error with question mark.
    file.read_to_string(&mut buffer)?;

    // remove any whitespace
    let buffer = buffer.trim();

    // We wrap this function call in an `Ok` because `?` will
    // automatically unwrap an `Ok` variant, but our function
    // signature requires a `Result`.
    Ok(u8::from_str_radix(buffer, 10)?)
}

// Same function as above, but without using question mark.
// This function also demonstrates different error handling strategies.
fn read_num_no_questionmark(path: &Path) -> Result<u8, Box<dyn Error>> {
    // make a buffer
    let mut buffer = String::new();

    // possible error when opening file (type annotation shown for clarity)
```

```
    let file: Result<File, io::Error> = File::open(path);

    // match on `file` to see what happened
    match file {
        // when open was successful ...
        Ok(mut file) => {
            // ... read data into a buffer ...
            if let Err(e) = file.read_to_string(&mut buffer) {
                // ... if that fails, return a boxed Err
                return Err(Box::new(e));
            }
        }
        // failed to open file, return `dyn Error` using `.into()`
        Err(e) => return Err(e.into()),
    }

    // remove any whitespace (yay no failure point!)
    let buffer = buffer.trim();

    // convert to u8 while manually mapping a possible conversion error
    u8::from_str_radix(buffer, 10).map_err(|e| e.into())
}

// calling the function is the same regardless of technique chosen
let num: Result<u8, _> = read_num_using_questionmark(Path::new("num.txt"));
if num.is_ok() {        // `.is_ok` will tell us if we have an Ok variant
    println!("number was successfully read");
}

// use some combinators on the result
let num = num             // take `num`
    .map(|n| n + 1)       // map an `Ok` variant by adding 1 to the value
    .ok()                 // transform to an `Option`
    .and_then(|n|         // and then ...
        n.checked_mul(2)  // double the inner value (this returns an Option)
    )
    .ok_or_else(||        // transform back into `Result` ...
        // ... using this error message if the multiplication failed
        format!("doubling exceeds size of u8")
    );

// use `match` to print out the result on an appropriate output stream
match num {
    Ok(n) => println!("{n}"),
    Err(e) => eprintln!("{e}"),
}
```

The `From` trait can also be utilized with errors and the question mark operator:

```
// a target error type
enum JobError {
    Expired,
    Missing,
    Other(u8),
}

// similar implementation from previous example
impl From<u8> for JobError {
    fn from(code: u8) -> Self {
        match code {
            1 => Self::Expired,
            2 => Self::Missing,
            c => Self::Other(c),
        }
    }
}

// arbitrary structure
struct Job;

impl Job {
    // function that returns an error code as u8
    fn whoops(&self) -> Result<(), u8> {
        Err(2)
    }
}
// function potentially returns a JobError
fn execute_job(job: Job) -> Result<(), JobError> {
    // use question mark to convert potential errors into a JobError
    Ok(job.whoops()?)
}

let status = execute_job(Job); // JobError::Missing
```

Iterator

The `Iterator` trait provides a large amount of functionality for iterating over collections using combinators.

Click for more details

```
// create a new vector ...
let nums = vec![1, 2, 3, 4, 5];
// ... then turn it into an iterator and multiply each element by 3 ...
let tripled = nums.iter().map(|n| n * 3);
```

```
// ... then filter out all the even numbers ...
let odds_only = tripled.filter(|n| n % 2 == 1);
// ... and finally collect the odd numbers into a new Vector
let new_vec: Vec<i32> = odds_only.collect();    // type annotation required

// same steps as above, but chaining it all together:
// create a new Vector
let nums = vec![1, 2, 3, 4, 5];
let tripled_odds_only = nums    // take the `nums` vector
    .iter()                     // then turn it into an iterator
    .filter_map(|n| {           // then perform a filter and map operation on each element
        let n = n * 3;          // multiply the element by 3
        if n % 2 == 1 {
            Some(n)             // keep if odd
        } else {
            None                // discard if even
        }
    })
    .collect::<Vec<i32>>();     // collect the remaining numbers into a Vector

// This example takes a vector of (x,y) points where all even indexes
// are the x-coordinates, and all odd indexes are y-coordinates and
// creates an iterator over tuples of the points.

// Points Vector: x, y, x, y, x, y, x, y, x, y
let points = vec![0, 0, 2, 1, 4, 3, 6, 5, 8, 7];

// `step_by` will skip every other index; iteration starts at index 0
let x = points.iter().step_by(2);

// use `skip` to skip 1 element so we start at index 1
// then skip every other index starting from index 1
let y = points.iter().skip(1).step_by(2);

// `zip` takes two iterators and generates a new one by taking
// alternating elements from each iterator. `enumerate` provides
// the iteration count as an index
let points = x.zip(y).enumerate();
for (i, point) in points {
    println!("{i}: {point:?}");
    // 0: (0, 0)
    // 1: (2, 1)
    // 2: (4, 3)
    // 3: (6, 5)
    // 4: (8, 7)
}

// create a new Vector
let nums = vec![1, 2, 3];
let sum = nums      // take `nums`
    .iter()         // create an iterator
    .sum::<i32>();  // add all elements together and return an i32
assert_eq!(sum, 6);
```

Ownership & Borrowing

All data in Rust is owned by some data structure or some function and the data can be borrowed by other functions or other data structures. This system enables compile-time tracking of how long data lives which in turn enables compile-time memory management without runtime overhead.

Click for more details

```
// borrow a str
fn print(msg: &str) {
    println!("{msg}");
}

// borrow a str, return a slice of the borrowed str
fn trim(msg: &str) -> &str {
    msg.trim()
}

// borrow a str, return an owned String
fn all_caps(msg: &str) -> String {
    msg.to_ascii_uppercase()
}

// function takes ownership of `msg` and is responsible
// for destroying it
fn move_me(msg: String) {
    println!("{msg}");
    // `msg` destroyed
}

// borrow "foo"
print("foo");

// borrow " bar "; return a new slice
let trimmed: &str = trim(" bar ");      // "bar"

// borrow "baz"; return a new String
let cruise_control: String = all_caps("baz");   // "BAZ"

// create owned String
let foo: String = String::from("foo");
// Move the String (foo) into move_me function.
// The `move_me` function will destroy `foo` since
// ownership was transferred.
let moved = move_me(foo);

// `foo` no longer exists
```

```
// println!("{foo}");
// ERROR: `foo` was moved into `move_me`
```

Lifetimes

Lifetimes allow specifying to the compiler that some data already exists. This allows creation of structures containing borrowed data or to return borrowed data from a function.

Click for more details

```
// Use lifetimes to indicate borrowed data stored in structures.
// Both structures and enumerations can have multiple lifetimes.
struct Email<'a, 'b> {
    subject: &'a str,
    body: &'b str,
}

let sample_subject = String::from("cheat sheet");
let sample_body = String::from("lots of code");
// `sample_subject` and `sample_body` are required to stay in memory
// as long as `email` exists.
let email = Email {
    subject: &sample_subject,
    body: &sample_body,
};

// dbg!(sample_subject);
// dbg!(email);
// ERROR: cannot move `sample_subject` into dbg macro because
//        `email` still needs it
```

```
// Lifetime 'a indicates borrowed data stored in the enum.
// The compiler uses 'a to enforce the following:
//  1. &str data must exist prior to creating a `StrCompare`
//  2. &str data must still exist after destruction of `StrCompare`
#[derive(Debug)]
enum StrCompare<'a> {
    Equal,
    Longest(&'a str),
}

// determine which &str is longer
// Lifetime annotations indicate that both `a` and `b` are
// borrowed and they will both exist for the same amount of time.
fn longest<'s>(a: &'s str, b: &'s str) -> StrCompare<'s> {
    if a.len() > b.len() {
```

```
        StrCompare::Longest(a)
    } else if a.len() < b.len() {
        StrCompare::Longest(b)
    } else {
        StrCompare::Equal
    }
}

// ERROR: the following block will not compile (see comments)

// new scope: lifetime (1)
{
    let a = String::from("abc");       // lifetime (1)
    let longstring: StrCompare;        // lifetime (1)

    // new scope: lifetime (2)
    {
        let b = String::from("1234"); // lifetime (2)
        longstring = longest(&a, &b);
        // end scope; lifetime (2) data dropped (destroyed):
        // `b` no longer exists
    }

    // `b` was previously dropped, but might still be needed here
    // as part of the `StrCompare` enumeration
    println!("{longstring:?}");     // ERROR: `b` doesn't live long enough

    // lifetime (1) data dropped in reverse creation order:
    // `longstring` no longer exists
    // `a` no longer exists
}

// FIXED: `a` and `b` now have same lifetime

// new scope: lifetime (1)
{
    let a = String::from("abc");       // lifetime (1)
    let b = String::from("1234");      // lifetime (1)
    let longstring = longest(&a, &b);  // lifetime (1)
    println!("{longstring:?}");

    // lifetime (1) data dropped in reverse creation order:
    // `longstring` dropped
    // `b` dropped
    // `a` dropped
}
```

Traits

Traits declare behavior that may be implemented by any structures or enumerations. Traits are similar to interfaces in other programming languages.

Click for more details

```
// create a new trait
trait Notify {
    // implementers must define this function
    fn notify(&self) -> &str;
}

struct Phone {
    txt: String,
}

struct Email {
    subject: String,
    body: String,
}

// implement the `Notify` trait for the `Phone` struct
impl Notify for Phone {
    fn notify(&self) -> &str {
        &self.txt
    }
}

// implement the `Notify` trait for the `Email` struct
impl Notify for Email {
    fn notify(&self) -> &str {
        &self.subject
    }
}

// create a new Phone
let phone = Phone {
    txt: String::from("foo"),
};

// create a new Email
let email = Email {
    subject: String::from("my email"),
    body: String::from("bar"),
};

phone.notify();     // "foo"
email.notify();     // "bar"
```

Associated Types

Associated types allow trait implementers to easily set a specific type for use in a trait.

Click for more details

```
trait Compute {
    // associated type to be defined by an implementer
    type Target;
    // use Self::Target to refer to the associated type
    fn compute(&self, rhs: Self::Target) -> Self::Target;
}

struct Add(i32);
struct Sub(f32);

impl Compute for Add {
    // set the associated type to i32
    type Target = i32;
    fn compute(&self, rhs: Self::Target) -> Self::Target {
        self.0 + rhs
    }
}

impl Compute for Sub {
    // set the associated type to f32
    type Target = f32;
    fn compute(&self, rhs: Self::Target) -> Self::Target {
        self.0 - rhs
    }
}

let add = Add(1);
let two = add.compute(1);
let sub = Sub(1.0);
let zero = sub.compute(1.0);
```

Trait Objects

Trait objects can be used to insert multiple objects of different types into a single collection. They are also useful when boxing closures or working with unsized types.

Click for more details

```
// create a trait to refill some resource
trait Refill {
    fn refill(&mut self);
}
```

```
// some structures to work with
struct Player { health_points: i32 }
struct MagicWand { magic_points: i32 }
struct Vehicle { fuel_remaining: i32 }

// set the maximum values for the structures
impl Player { const MAX_HEALTH: i32 = 100; }
impl MagicWand { const MAX_MAGIC: i32 = 100; }
impl Vehicle { const MAX_FUEL: i32 = 300; }

// trait implementations for all 3 structures
impl Refill for Player {
    fn refill(&mut self) {
        self.health_points = Self::MAX_HEALTH;
    }
}
impl Refill for MagicWand {
    fn refill(&mut self) {
        self.magic_points = Self::MAX_MAGIC;
    }
}
impl Refill for Vehicle {
    fn refill(&mut self) {
        self.fuel_remaining = Self::MAX_FUEL;
    }
}

// instantiate some structures
let player = Player { health_points: 50 };
let wand = MagicWand { magic_points: 30 };
let vehicle = Vehicle { fuel_remaining: 0 };

// let objects = vec![player, wand, vehicle];
// ERROR: cannot have a Vector containing different types

// Type annotation is required here. `dyn` keyword indicates
// "dynamic dispatch" and is also required for trait objects.
let mut objects: Vec<Box<dyn Refill>> =
    vec![
        Box::new(player),                // must be boxed
        Box::new(wand),
        Box::new(vehicle)
    ];

// iterate over the collection and refill all of the resources
for obj in objects.iter_mut() {
    obj.refill();
}
```

Default

The `Default` trait allows a default version of a structure to be easily created.

Click for more details

```
struct Foo {
    a: usize,
    b: usize,
}

// Default is available without `use`
impl Default for Foo {
    fn default() -> Self {
        Self { a: 0, b: 0 }
    }
}

// make a new Foo
let foo = Foo::default();

// make a new Foo with specific values set
// and use default values for the rest
let foo = Foo {
    a: 10,
    ..Default::default() // b: 0
};

// we might have a Foo ...
let maybe_foo: Option<Foo> = None;
// ... if not, use the default one
let definitely_foo = maybe_foo.unwrap_or_default();
```

From / Into

`From` and `Into` traits allow non-fallible conversion between different types. If the conversion can fail, the `TryFrom` and `TryInto` traits will perform fallible conversions. Always prefer implementing `From` because it will automatically give you an implementation of `Into`.

Click for more details on From

Click for more details on TryFrom

```
// this will be our target type
enum Status {
    Broken(u8),
```

```
    Working,
}

// we want to convert from a `u8` into a `Status`
impl From<u8> for Status {
    // function parameter must be the starting type
    fn from(code: u8) -> Self {
        match code {
            // pick a variant based on the code
            0 => Status::Working,
            c => Status::Broken(code),
        }
    }
}

// use `.into()` to convert the `u8` into a Status
let status: Status = 0.into();  // Status::Working
// use `Status::from()` to convert from a `u8` into a Status
let status = Status::from(1);   // Status::Broken(1)
```

Generics

Rust data structures and functions only operate on a single data type. Generics provide a way to automatically generate duplicated functions and data structures appropriate for the data types in use.

Click for more details

```
// Here we define a structure generic over type T.
// T has no trait bounds, so any type can be used here.
struct MyVec<T> {
    inner: Vec<T>,     // Vector of type T
}

// define a structure generic over type T where
// type T implements the Debug trait
struct MyVec<T: std::fmt::Debug> {
    inner: Vec<T>,
}

// define a structure generic over type T where
// type T implements both the Debug and Display traits
struct MyVec<T>
where
    T: std::fmt::Debug + std::fmt::Display,
{
    inner: Vec<T>,
}
```

```
// create a new MyVec having type `usize`
let nums: MyVec<usize> = MyVec { inner: vec![1, 2, 3] };

// create a new MyVec with type inference
let nums = MyVec { inner: vec![1, 2, 3] };

// let nums = MyVec { inner: vec![] };
// ERROR: type annotations required because no inner data type provided

let nums: MyVec<String> = MyVec { inner: vec![] };
// OK using type annotations
```

```
// use the `Add` trait
use std::ops::Add;
// pub trait Add<Rhs = Self> {
//     type Output;
//     fn add(self, rhs: Rhs) -> Self::Output;
// }

// Here we define a function that is generic over type T.
// Type T has the following properties:
//   - Must implement the `Add` trait
//   - The associated type `Output` must
//     be the same type T
fn sum<T: Add<Output = T>>(lhs: T, rhs: T) -> T {
    lhs + rhs
}
let two = sum(1, 1);              // call the function
let two = sum::<f64>(1.0, 1.0);   // fully-qualified syntax

// let four_ish = sum(2, 2.0);
// ERROR: 2 is an integer and 2.0 is a floating point number,
//        but the generic function requires both types be the same
```

Operator Overloading

Rust enables developers to overload existing operators. The operators are defined as traits in the link below.

Click for more details, and a list of all operators

```
// the `Add` trait is used for the `+` operator
use std::ops::Add;

struct Centimeters(f64);

// implement Add for Centimeters + Centimeters
```

```
impl Add<Self> for Centimeters {
    // Self (capital S) refers to the type specified
    // in the `impl` block (Centimeters)
    type Output = Self;

    // self (lowercase S) refers to an instance of Centimeters.
    // Using `Self` makes it easier to change the types
    // later if needed.
    fn add(self, rhs: Self) -> Self::Output {
        Self(self.0 + rhs.0)
    }

    // equivalent to the above
    fn add(self, rhs: Centimeters) -> Centimeters {
        Centimeters(self.0 + rhs.0)
    }

}

fn add_distance(a: Centimeters, b: Centimeters) -> Centimeters {
    // When `+` is used, it simply calls the `add` function
    // defined as part of the `Add` trait. Since we already
    // access the inner value using .0 in the trait, we can
    // just do a + b here.
    a + b
}

let length = Centimeters(20.0);
let distance = add_distance(length, Centimeters(10.0));
```

Index

The `Index` trait is used for indexing operations. Implementing this trait on a structure permits accessing it's fields using indexing.

Click for more details

```
// the `Index` trait is used for indexing operations `[]`
use std::ops::Index;

// this will be our index
enum Temp {
    Current,
    Max,
    Min,
}

// sample structure to be indexed into
struct Hvac {
    current_temp: f64,
```

```
    max_temp: f64,
    min_temp: f64,
}

// implement Index where the index is Temp and the structure is Hvac
impl Index<Temp> for Hvac {
    // output type matches the data we will return from the structure
    type Output = f64;

    // `index` function parameter must be the type to be
    // used as an index
    fn index(&self, temp: Temp) -> &Self::Output {
        use Temp::*;    // use the variants for shorter code
        match temp {
            // now just access the structure fields
            // based on provided variant
            Current => &self.current_temp,
            Max => &self.max_temp,
            Min => &self.min_temp,
        }
    }
}

// create a new Hvac
let env = Hvac {
    current_temp: 30.0,
    max_temp: 60.0,
    min_temp: 0.0,
};
// get the current temperature using an Index
let current = env[Temp::Current];
// get the max temperature using an Index
let max = env[Temp::Max];
```

Concurrent Programming

Rust provides multiple techniques to approach concurrent programming. Computation-heavy workloads can use OS threads while idle workloads can use asynchronous programming. Concurrent-aware types and data structures allow wrapping existing structures which enables them to be utilized in a concurrent context.

Threads

Rust provides the ability to create OS threads via the `std::thread` module. Any number of threads can be created, however performance will be optimal when using the same number of thread as there are processing cores in the system.

Click for more details

```
use std::thread::{self, JoinHandle};

// The thread::spawn function will create a new thread and return
// a `JoinHandle` type that can be used to wait for this thread
// to finish working.
let thread_1 = thread::spawn(|| {});

// JoinHandle is generic over the return type from the thread
let thread_2: JoinHandle<usize> = thread::spawn(|| 1);

// wait for both threads to finish work
thread_1.join();
thread_2.join();
```

Channels

Channels provide a way to communicate between two points and are used for transferring data between threads. They have two ends: a *sender* and a *receiver*. The sender is used to send/write data into the channel, and the receiver is used to receive/read data out of the channel.

Click for more details

```
// The crossbeam_channel crate provides better performance
// and ergonomics compared to the standard library.
use crossbeam_channel::unbounded;

// create a channel with unlimited capacity
let (sender, receiver) = unbounded();

// data can be "sent" on the `sender` end
// and "received" on the `receiver` end
sender.send("Hello, channel!").unwrap();

// use `.recv` to read a message
match receiver.recv() {
    Ok(msg) => println!("{msg}"),
    Err(e) => println!("{e}"),
}
```

Using channels with threads:

```
// The crossbeam_channel crate provides better performance
// and ergonomics compared to the standard library.
use crossbeam_channel::unbounded;
```

```
use std::thread;

// create a channel with unlimited capacity
let (sender, receiver) = unbounded();

// clone the receiving ends so they can be sent to different threads
let (r1, r2) = (receiver.clone(), receiver.clone());

// move `r1` into this thread with the `move` keyword
let thread_1 = thread::spawn(move || match r1.recv() {
    Ok(msg) => println!("thread 1 msg: {msg}"),
    Err(e) => eprintln!("thread 1 error: {e}"),
});

// move `r2` into this thread with the `move` keyword
let thread_2 = thread::spawn(move || match r2.recv() {
    Ok(msg) => println!("thread 2 msg: {msg}"),
    Err(e) => eprintln!("thread 2 error: {e}"),
});

// send 2 messages into the channel
sender.send("Hello 1").unwrap();
sender.send("Hello 2").unwrap();

// wait for the threads to finish
thread_1.join();
thread_2.join();
```

Mutex

Mutex (short for **mut**ually **ex**clusive) allows data to be shared across multiple threads by using a locking mechanism. When a thread *locks* the Mutex, it will have exclusive access to the underlying data. Once processing is completed, the Mutex is *unlocked* and other threads will be able to access it.

Click for more details

```
// The parking_lot crate provides better performance
// and ergonomics compared to the standard library.
use parking_lot::Mutex;
// `Arc` is short for Atomic reference-counted pointer
// (thread safe pointer)
use std::sync::Arc;
use std::thread;

// data we will share between threads
struct Counter(usize);

// make a new Counter starting from 0
```

```
let counter = Counter(0);
// wrap the counter in a Mutex and wrap the Mutex in an Arc
let shared_counter = Arc::new(Mutex::new(counter));

// make some copies of the pointer:
// recommended syntax - clear to see that we are cloning a pointer (Arc)
let thread_1_counter = Arc::clone(&shared_counter);
// ok too, but not as clear as above; shared_counter could be anything
let thread_2_counter = shared_counter.clone();

// spawn a thread
let thread_1 = thread::spawn(move || {
    // lock the counter so we can access it
    let mut counter = thread_1_counter.lock();
    counter.0 += 1;
    // lock is automatically unlocked when dropped
});

let thread_2 = thread::spawn(move || {
    // new scopes can be introduced to drop the lock ...
    {
        let mut counter = thread_2_counter.lock();
        counter.0 += 1;
        // ... lock automatically unlocked
    }
    let mut counter = thread_2_counter.lock();
    counter.0 += 1;
    // we can also call `drop()` directly to unlock
    drop(counter);
});

// wait for threads to finish
thread_1.join();
thread_2.join();
// counter is now at 3
assert_eq!(shared_counter.lock().0, 3);
```

Async

Rust's asynchronous programming consists of two parts:

- A `future` which represents an asynchronous operation that should be ran
- An `executor` (or `runtime`) which is responsible for managing and running `futures` (as `tasks`)

There are `async` versions of many existing structures:

- streams (iterators)

- adapters (combinators)
- channels
- mutex

Click for more details

```
// Use the `futures` crate and `FutureExt` when working with async.
// `FutureExt` provides combinators similar to `Option` and `Result`
use futures::future::{self, FutureExt};

// asynchronous functions start with the `async` keyword
async fn add_one(n: usize) -> usize {
    n + 1
}

// the `tokio` crate provides a commonly used executor
#[tokio::main]
async fn main() {
    // async functions are lazy--no computation happens yet

    let one = async { 1 };          // inline future
    let two = one.map(|n| n + 1);   // add 1 to the future

    let three = async { 3 };        // inline future
    let four = three.then(add_one); // run async function on future

    // `join` will wait on both futures to complete.
    // `.await` begins execution of the futures.
    let result = future::join(two, four).await;

    assert_eq!(result, (2, 4))
}
```

Streams provide `Iterator`-like functionality to asynchronous streams of values.

```
// Use the `futures` crate when working with async.
use futures::future;

// `StreamExt` provides combinators similar to `Iterator`
use futures::stream::{self, StreamExt};

// the `tokio` crate provides a commonly used executor
#[tokio::main]
async fn main() {
    let nums = vec![1, 2, 3, 4];
    // create a stream from the Vector
    let num_stream = stream::iter(nums);
```

```
    let new_nums = num_stream            // take num_stream
        .map(|n| n * 3)                  // multiply each value by 3
        .filter(|n|                      // filter ...
            future::ready(n % 2 == 0)    // ... only take even numbers
        )
        .then(|n| async move { n + 1 })  // run async function on each value
        .collect::<Vec<_>>().await;      // collect into a Vector

    assert_eq!(new_nums, vec![7, 13]);

    stream::iter(vec![1, 2, 3, 4])
        .for_each_concurrent(          // perform some action concurrently
            2,                         // maximum number of in-flight tasks
            |n| async move {           // action to take
                // some potentially
                // async code here
            }
        ).await;                       // run on the executor
}
```

Modules

Code in Rust is organized into modules. Modules can be created inline with code, or using the filesystem where each file is a module or each directory is a module (containing more files).

Modules are accessed as *paths* starting either from the root or from the current module. This applies to both inline modules and modules as separate files.

Click for more details

Inline Modules

```
// private module: only accessible within the same scope (file / module)
mod sample {
    // bring an inner module to this scope
    pub use public_fn as inner_public_fn;   // rename to inner_public_fn

    // default: private
    fn private_fn() {}

    // public to parent module
    pub fn public_fn() {}

    // public interface to private_fn
    pub fn public_interface() {
        private_fn();            // sample::private_fn
```

```
        inner::limited_super();
        inner::limited_module();
    }

    // public module: accessible via `sample`
    pub mod inner {
        fn private_fn() {}

        pub fn public_fn() {}

        pub fn public_interface() {
            private_fn();               // inner::private_fn

            super::hidden::public_fn(); // `inner` and `hidden` are in
                                        // the same scope, so this is Ok.
        }

        // public only to the immediate parent module
        pub(super) fn limited_super() {}

        // public only to the specified ancestor module
        pub(in crate::sample) fn limited_module() {}

        // public to the entire crate
        pub(crate) fn limited_crate() {}
    }

    // private module: can only be accessed by `sample`
    mod hidden {
        fn private_fn() {}

        pub fn public_fn() {}

        pub fn public_interface() {
            private_fn()    // hidden::private_fn
        }

        // It's not possible to access module `sample::hidden` from outside of
        // `sample`, so `fn limited_crate` is public only to the `sample`
        // module.
        pub(crate) fn limited_crate() {}
    }
}

fn main() {
    // functions can be accessed by their path
    sample::public_fn();
    sample::public_interface();
    sample::inner_public_fn();         // sample::inner::public_fn

    // ERROR: private_fn() is private
    // sample::private_fn();

    // nested modules can be accessed by their path
```

```
    sample::inner::public_fn();
    sample::inner::public_interface();
    sample::inner::limited_crate();

    // ERROR: private_fn() is private
    // sample::inner::private_fn();

    // ERROR: limited_super() is only public within `sample`
    // sample::inner::limited_super();

    // ERROR: limited_module() is only public within `sample`
    // sample::inner::limited_module();

    // ERROR: `hidden` module is private
    // sample::hidden::private_fn();

    // ERROR: `hidden` module is private
    // sample::hidden::public_fn();

    // ERROR: `hidden` module is private
    // sample::hidden::public_interface();

    // `use` brings specific items into scope
    {
        // a single function
        use sample::public_fn;
        public_fn();

        // begin path from crate root
        use crate::sample::public_interface;
        public_interface();

        // rename an item
        use sample::inner::public_fn as other_public_fn;
        other_public_fn();
    }
    {
        // multiple items from a single module
        use sample::{public_fn, public_interface};
        public_fn();
        public_interface();
    }
    {
        // `self` in this context refers to the `inner` module
        use sample::inner::{self, public_fn};
        public_fn();
        inner::public_interface();
    }
    {
        // bring everything from `sample` into this scope
        use sample::*;
        public_fn();
        public_interface();
        inner::public_fn();
```

```
        inner::public_interface();
    }
    {
        // paths can be combined
        use sample::{
            public_fn,
            inner::public_fn as other_public_fn
        };
        public_fn();        // sample::public_fn
        other_public_fn()   // inner::public_fn
    }
}
```

Modules as Files

Cargo.toml

```
[lib]
name = "sample"
path = "lib/sample.rs"
```

Module directory structure

```
.
|-- lib
    |-- sample.rs
    |-- file.rs
    |-- dir/
        |-- mod.rs
        |-- public.rs
        |-- hidden.rs
```

./lib/sample.rs: this is the path indicated by `path` in `Cargo.toml`

```
// a module in a single file named `file.rs`
pub mod file;

// a module in a directory named `dir`
pub mod dir;

// functions / enums / structs / etc can be defined here also
pub fn foo() {}
```

./lib/file.rs

```
pub fn foo() {}
```

A file named `mod.rs` is required when creating a module from a directory. This file specifies item visibility and specifies additional modules.

./lib/dir/mod.rs:

```
// a module in a single file named `hidden.rs`
mod hidden;

// a module in a single file named `public.rs`
pub mod public;

pub fn foo() {}
```

./lib/dir/hidden.rs

```
pub fn foo() {}
```

./lib/dir/public.rs

```
pub fn foo() {}
```

./src/main.rs

```
fn main() {
    sample::file::foo();
    sample::dir::public::foo();
    sample::dir::foo();
    // ERROR: `hidden` module not marked as `pub`
    // sample::dir::hidden::foo();
}
```

Testing

Rust supports testing both private and public functions and will also test examples present in documentation.

Click for more details

Test Module

Using a dedicated `test` module within each file for testing is common:

```
use std::borrow::Cow;

// a function to test
fn capitalize_first_letter<'a>(input: &'a str) -> Cow<'a, str> {
    use unicode_segmentation::UnicodeSegmentation;
    // do nothing if the string is empty
    if input.is_empty() {
        Cow::Borrowed(input)
    } else {
        let graphemes = input.graphemes(true).collect::<Vec<&str>>();
        if graphemes.len() >= 1 {
            let first = graphemes[0];
            let capitalized = first.to_uppercase();
            let remainder = graphemes[1..]
                .iter()
                .map(|s| s.to_owned())
                .collect::<String>();
            Cow::Owned(format!("{capitalized}{remainder}"))
        } else {
            Cow::Borrowed(input)
        }
    }
}

// another function to test
fn is_local_phone_number(num: &str) -> bool {
    use regex::Regex;
    let re = Regex::new(r"[0-9]{3}-[0-9]{4}").unwrap();
    re.is_match(num)
}

// Use the `test` configuration option to only compile the `test` module
// when running `cargo test`.
#[cfg(test)]
mod test {
    // scoping rules require us to use whichever functions we are testing
    use super::{is_local_phone_number, capitalize_first_letter};

    // use the #[test] annotation to mark the function as a test
```

```
    #[test]
    fn accepts_valid_numbers() {
        // assert! will check if the value is true, and panic otherwise.
        // Test failure is marked by a panic in the test function.
        assert!(is_local_phone_number("123-4567"));
    }

    #[test]
    fn rejects_invalid_numbers() {
        // we can use multiple assert! invocations
        assert!(!is_local_phone_number("123-567"));
        assert!(!is_local_phone_number("12-4567"));
        assert!(!is_local_phone_number("-567"));
        assert!(!is_local_phone_number("-"));
        assert!(!is_local_phone_number("1234567"));
        assert!(!is_local_phone_number("one-four"));
    }

    #[test]
    fn rejects_invalid_numbers_alternate() {
        // We can also put the test data into a Vector
        // and perform the assert! in a loop.
        let invalid_numbers = vec![
            "123-567",
            "12-4567",
            "-567",
            "-",
            "1234567",
            "one-four",
        ];
        for num in invalid_numbers.iter() {
            assert!(!is_local_phone_number(num));
        }
    }

    #[test]
    fn capitalizes_first_letter_with_multiple_letter_input() {
        let result = capitalize_first_letter("test");
        // assert_eq! will check if the left value is
        // equal to the right value
        assert_eq!(result, String::from("Test"));
    }

    #[test]
    fn capitalizes_first_letter_with_one_letter_input() {
        let result = capitalize_first_letter("t");
        assert_eq!(result, String::from("T"));
    }

    #[test]
    fn capitalize_only_letters() {
        let data = vec![
            ("3test", "3test"),
            (".test", ".test"),
```

```
            ("-test", "-test"),
            (" test", " test"),
        ];
        for (input, expected) in data.iter() {
            let result = capitalize_first_letter(input);
            assert_eq!(result, *expected);
        }
    }
}
```

Doctests

Rust tests example code present in documentation. This happens automatically when running `cargo test`, but will only operate on library projects.

Click for more details

```
use std::borrow::Cow;
/// Capitalizes the first letter of the input `&str`.
///
/// Only capitalizes the first letter when it appears as the first character
/// of the input. If the first letter of the input `&str` is not a letter
/// that can be capitalized (such as a number or symbol), then no change will occur.
///
/// # Examples
///
/// All code examples here will be tested. Lines within the code
/// fence that begin with a hash (#) will be hidden in the docs.
///
/// ```
/// # use crate_name::capitalize_first_letter;
/// let hello = capitalize_first_letter("hello");
/// assert_eq!(hello, "Hello");
/// ```
fn capitalize_first_letter<'a>(input: &'a str) -> Cow<'a, str> {
    use unicode_segmentation::UnicodeSegmentation;
    // do nothing if the string is empty
    if input.is_empty() {
        Cow::Borrowed(input)
    } else {
        let graphemes = input.graphemes(true).collect::<Vec<&str>>();
        if graphemes.len() >= 1 {
            let first = graphemes[0];
            let capitalized = first.to_uppercase();
            let remainder = graphemes[1..].iter().map(|s| s.to_owned()).collect::<String>
();
            Cow::Owned(format!("{capitalized}{remainder}"))
        } else {
            Cow::Borrowed(input)
        }
```

```
    }
}
```

Standard Library Macros

The standard library provides convenient macros for performing various tasks. A subset is listed below.

Click for more details

Macro	Description
`assert`	Checks if a boolean is `true` at runtime and panics if `false`.
`assert_eq`	Checks if two expressions are equal at runtime and panics if not.
`dbg`	Prints debugging information for the given expression.
`env`	Inserts data from an environment variable at compile time.
`println`	Format and print information (with a newline) to the terminal on `stdout`.
`eprintln`	Format and print information (with a newline) to the terminal on `stderr`.
`print`	Format and print information (with no newline) to the terminal on `stdout`.
`eprint`	Format and print information (with no newline) to the terminal on `stderr`.
`format`	Format information and return a new `String`.
`include_str`	Include data from a file as a `'static str` at compile time.
`include_bytes`	Include data from a file as a byte array at compile time.
`panic`	Triggers a panic on the current thread.
`todo`	Indicates unfinished code; will panic if executed. Will type-check properly during compilation.
`unimplemented`	Indicates code that is not implemented and with no immediate plans to implement; will panic if executed. Will type-check properly during compilation.
`unreachable`	Indicates code that should never be executed. Use when compiler is unable to make this determination. Will type-check properly during compilation.
`vec`	Create a new `Vector`.

Standard Library Derive Macros

`Derive` macros allow functionality to be implemented on structures or enumerations with a single line of code.

Macro	Description
`Clone`	Explicit copy using `.clone()`
`Copy`	Type will be *implicitly* copied by compiler when needed. Requires `Clone` to be implemented.
`Debug`	Enable formatting using `{:?}`
`Default`	Implements `Default` trait
`Hash`	Enables usage with a `Hasher` (ie: keys in a `HashMap`). Requires `PartialEq` and `Eq` to be implemented.
`Eq`	Symmetric equality (`a == b` and `b == a`) and transitive equality (if `a == b` and `b == c` then `a == c`). Requires `PartialEq` to be implemented.
`PartialEq`	Allows comparison with `==`
`Ord`	Transitive ordering (if `a < b` and `b < c` then `a < c`). Requires `Eq` to be implemented.
`PartialOrd`	Allow comparison with `<`, `<=`, `>`, `>=`. Requires `PartialEq` to be implemented.

```
// enable `Foo` to be Debug-printed and
// implicitly copied if needed
#[derive(Debug, Clone, Copy)]
struct Foo(usize);

fn hi(f: Foo) {}

let foo = Foo(1);
hi(foo);   // moved
hi(foo);   // implicit copy
hi(foo);   // implicit copy
hi(foo);   // implicit copy

// enable `Name` to be used as a key in a HashMap
#[derive(Eq, PartialEq, Hash)]
struct Name(String);

let name = Name("cheat sheet".into());
let mut names = std::collections::HashMap::new();
names.insert(name, ());
```

```
// enable `Letters` to be used with comparison operators
#[derive(PartialEq, PartialOrd)]
enum Letters {
    A,
    B,
    C,
}

if Letters::A < Letters::B { /* ... */ }
```

Declarative Macros

Declarative macros operate on code instead of data and are commonly used to write implementation blocks and tests.

Click for more details

```
// use `macro_rules!` to create a macro
macro_rules! name_of_macro_here {
    // Macros consist of one or more "matchers", each having a "transcriber".
    // This is similar to having multiple arms in a `match` expression.
    // Matchers are evaluated from top to bottom.
    () => {};

    (
        // Matchers have metavariables and fragment specifiers
        // which are detailed in the following sections.
    ) => {
        // Transcribers represent the code that will be
        // generated by the macro. Metavariables can be
        // used here for generating code.
    };

    (2) => {};
    (3) => {};
}

name_of_macro_here!();      // first matcher will match
name_of_macro_here!(1);     // second matcher will match
name_of_macro_here!(2);     // third matcher will match
name_of_macro_here!(3);     // fourth matcher will match
```

Valid Positions

Declarative macros can be used in some (but not all!) positions in Rust code.

`Expression`

Right-hand side of expressions or statements.

```
let nums = vec![1, 2, 3];
match vec![1, 2, 3].as_slice() {
    _ => format!("hello"),
}
```

`Statement`

Usually ends with a semicolon.

```
println!("Hello!");
dbg!(9_i64.pow(2));
```

`Pattern`

Match arms or `if let` patterns.

```
if let pat!(x) = Some(1) { }
match Some(1) {
    pat!(x) => (),
    _ => ()
}
```

`Type`

Anywhere you can use a type annotation.

```
macro_rules! Tuple {
    { $A:ty, $B:ty } => { ($A, $B) };
}
type N2 = Tuple!(i32, i32);
let nums: Tuple(i32, char) = (1, 'a');
```

`Item`

Anywhere you can declare a constant, `impl` block, enum, module, etc.

```
macro_rules! constant {
    ($name:ident) => { const $name: &'static str = "Cheat sheet"; }
}
macro_rules! newtype {
    ($name:ident, $typ:ty) => { struct $name($typ); }
}
constant!(NAME);
assert_eq!(NAME, "Cheat sheet");

newtype!(DemoStruct, usize);
let demo = DemoStruct(5);
```

`Associated Item`

Like an `Item`, but specifically within an `impl` block or `trait`.

```
macro_rules! msg {
    ($msg:literal) => {
        pub fn msg() {
            println!("{}", $msg);
        }
    };
}
struct Demo;
// Associated item
impl Demo {
    msg!("demos struct");
}
```

`macro_rules Transcribers`

Declarative macros can be present within other declarative macros.

```
macro_rules! demo {
    () => {
        println!("{}",
            format!("demo{}", '!')
        );
    };
}
demo!()
```

Fragment Specifiers

Declarative macros operate on `metavariables`. Just like a function parameter, metavariables require a name and a type. Here is a list metavariable types that can be used for declarative macros, and code examples of that type.

`$item`

```
macro_rules! demo {
    ($i:item) => { $i };
}
demo!(const a: char = 'g';);
demo! {fn hello(){}}
demo! {mod demo{}}
struct MyNum(i32);
demo! {
    impl MyNum {
        pub fn demo(&self) {
            println!("my num is {}", self.0);
        }
    }
}
```

`$block`

```
macro_rules! demo {
    ($b:block) => { $b };
}

let num = demo!(
    {
        if 1 == 1 { 1 } else { 2 }
    }
);
```

`$stmt`

```
macro_rules! demo {
    ($s:stmt) => { $s };
}

demo!( let a = 5 );
let mut myvec = vec![];
demo!( mybec.push(a) );
```

`$pat` / `$pat_param`

```
macro_rules! demo {
    ($p:pat) => {{
        let num = 3;
        match num {
            $p => (),
            1 => (),
            _ => (),
        }
    }};
}
demo! ( 2 );
```

`$expr`

```
macro_rules! demo {
    ($e:expr) => { $e };
}

demo!( loop {} );
demo!( 2 + 2 );
demo!( {
    panic!();
} );
```

`$ty`

```
macro_rules! demo {
    ($t:ty) => {{
        let d: $t = 4;
        fn add(lhs: $t, rhs: $t) -> $t {
            lhs + rhs
        }
    }};
}
demo!(i32);
demo!(usize);
```

`$ident`

```
macro_rules! demo {
    ($i:ident, $i2:ident) => {
        fn $i() {
            println!("hello");
```

```
        }
        let $i2 = 5;
    };
}
demo!(say_hi, five);
say_hi();
assert_eq!(5, five);
```

`$path`

```
macro_rules! demo {
    ($p:path) => {
        use $p;
    };
}
demo!(std::collections::HashMap);
```

`$tt`

```
macro_rules! demo {
    ($t:tt) => {
        $t {}
    };
}
demo!(loop);
demo!({
    println!("hello");
});
```

`$meta`

```
macro_rules! demo {
    ($m:meta) => {
        $[derive($m)]
        struct MyNum(i32);
    };
}
demo!(Debug);
```

`$lifetime`

```
macro_rules! demo {
    ($l:lifetime) => {
        let a: &$l str = "sample";
    };
```

```
}
demo!('static);
```

`$vis`

```
macro_rules! demo {
    ($v:vis) => {
        $v fn sample() {}
    };
}
demo!(pub);
```

`$literal`

```
macro_rules! demo {
    $(l:literal) => { $l };
}
let five = demo!(5);
let hi = demo!("hello");
```

Repetitions

One of the primary use cases for macros is automatically writing code for multiple inputs. Repetitions are used to accomplish this.

```
macro_rules! demo {
    // zero or more
    (
        // comma (,) is a separator between each `frag`
        $( $metavar:frag ),*
    ) => {
        // using a repetition requires the same repetition symbol
        // as specified in the matcher above
        $( $metavar )*
    }

    // one or more
    (
        // comma (,) is a separator between each `frag`
        $( $metavar:frag ),+
    ) => {
        // using a repetition requires the same repetition symbol
        // as specified in the matcher above
        $( $metavar )+
    }
```

```
    // zero or one
    (
        // no separator possible because only 0 or 1 `frag` may be present
        $( $metavar:frag )?
    ) => {
        // using a repetition requires the same repetition symbol
        // as specified in the matcher above
        $( $metavar )?
    }
}
```

```
macro_rules! demo {
    (
        // zero or one literals
        $( $a:literal )?
    ) => {
        $($a)?
    }
}
demo!();
demo!(1);
```

```
macro_rules! demo {
    (
        // one or more literals separated by a comma
        $( $a:literal ),+
    ) => {
        $(
            println!("{}", $a);
        )+
    }
}
demo!(1);
demo!(1, 2, 3, 4);
```

```
macro_rules! demo {
    (
        // any number of literals separated by a comma
        $( $a:literal ),*
    ) => {
        $(
            println!("{}", $a);
        )*
    }
}
```

```
demo!();
demo!(1);
demo!(1, 2, 3, 4);
```

```
macro_rules! demo {
    (
        // any number of literals separated by a comma
        // and may have a trailing comma at the end
        $( $a:literal ),*
        $(,)?
    ) => {
        $(
            println!("{}", $a);
        )*
    }
}
demo!();
demo!(1);
demo!(1, 2, 3, 4,);
```

Example Macros

Here is an example of a macro to write multiple tests:

```
macro_rules! test_many {
    (
        // name of a function followed by a colon
        $fn:ident:
        // "a literal followed by -> followed by a literal"
        // repeat the above any number of times separated by a comma
        $( $in:literal -> $expect:literal ),*
    ) => {
        // repeat this code for each match
        $(
            // $fn = name of the function
            // $in = input number to the function
            // $expect = expected output from the function
            assert_eq!($fn($in), $expect);
        )*
    }
}

// function under test
fn double(v: usize) -> usize {
    v * 2
}
```

```
// invoking the macro
test_many!(double: 0->0, 1->2, 2->4, 3->6, 4->8);
```

Here is an example of a macro to write multiple implementation blocks:

```
// trait we want to implement
trait BasePay {
    fn base_pay() -> u32;
}

// structures we want the trait implemented on
struct Employee;
struct Supervisor;
struct Manager;

// macro to implement the trait
macro_rules! impl_base_pay {
    (
        // begin repetition
        $(
            // name of stucture for implementation, followed
            // by a colon (:) followed by a number
            $struct:ident: $pay:literal
        ),+ // repeat 1 or more times

        $(,)? // optional trailing comma between entries
    ) => {
        // begin repetition
        $(
            // our impl block using a metavariable
            impl BasePay for $struct {
                fn base_pay() -> u32 {
                    // just return the literal
                    $pay
                }
            }
        )+  // repeat 1 or more times
    }
}

// invoking the macro ...
impl_base_pay!(
    Employee: 10,
    Supervisor: 20,
    Manager: 30,
);

// ... generates
impl BasePay for Employee {
    fn base_pay() -> u32 {
        10
```

```
    }
}
impl BasePay for Supervisor {
    fn base_pay() -> u32 {
        20
    }
}
impl BasePay for Manager {
    fn base_pay() -> u32 {
        30
    }
}
```

Macro Notes

Macros can be invoked from anywhere in a crate, so it is important to use absolute paths to functions, modules, types, etc. when the macro is to be used outside of where it is defined.

For `std`, prefix the `std` crate with two colons like so:

```
use ::std::collections::HashMap;
```

For items that exist in the current crate, use the special `$crate` metavariable:

```
use $crate::modulename::my_item;
```