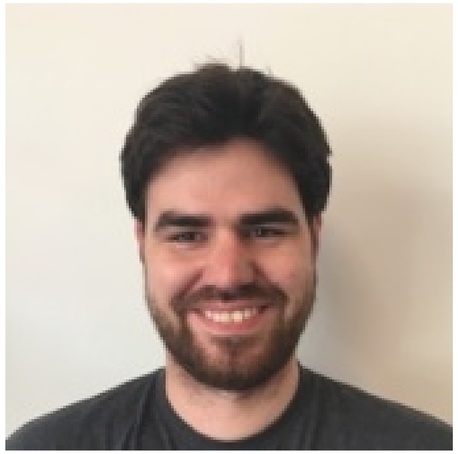
Rust Intro

Programming with confidence

Who are we?

Ruben Nijveld

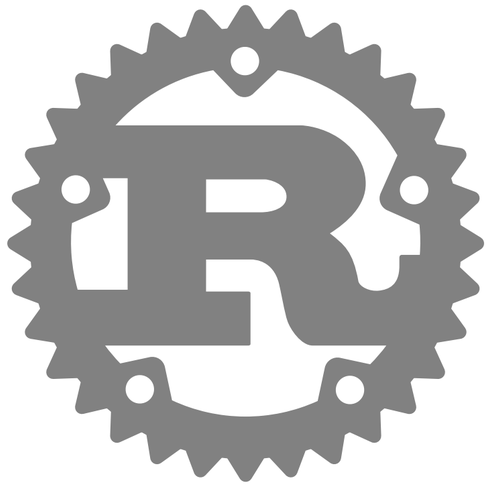


Folkert de Vries

Today

- Get a feeling of the language, but it will take time to fully learn Rust
- Trusting your tools, so that you can focus on the important stuff
- Core concepts that can help you even if you never write another line of Rust
- Most of this information can also be found in the rust book

Today

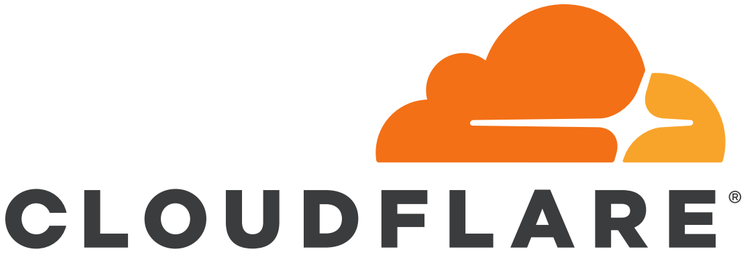


- **Tooling**
  - Using Rust in day to day work
- **Language**
  - Programming in Rust and core concepts
- **Productivity**
- **Reliability**
- **Performance**

History

- 2006: Personal project by Graydon Hoare at Mozilla
- 2010: Mozilla announced Rust to the public
- 2012: First alpha release
- 2015: Rust 1.0 released
- 2018: Rust 2018 edition
- 2020: Mozilla restructures
- 2021: Rust Foundation

Users of Rust



The Rust toolchain

- `rustup`: manage and install toolchains
- `rustc`: the Rust compiler
- `cargo`: dependency management and package (crate) compilation and distribution
- `rustdoc`: The documentation generation tool (`cargo doc`)
- `rustfmt`: Formatter for Rust source files (`cargo fmt`)
- `clippy`: Additional linting beyond compiler warnings (`cargo clippy`)

A new project

```
$ cargo init hello-world
```

```
$ cd hello-world
$ cargo run
```

```
   Compiling hello-world v0.1.0 (/home/ruben/Projects/hello-world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.74s
     Running `target/debug/hello-world`
Hello, world!
```

Hello, world!

```
1   fn main() {
2       println!("Hello, world! fib(6) = {}", fib(6));
3   }
4
5   fn fib(n: u64) -> u64 {
6       if n <= 1 {
7           n
8       } else {
9           fib(n - 1) + fib(n - 2)
10      }
11  }
```

```
   Compiling hello-world v0.1.0 (/home/ruben/Projects/hello-world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `target/debug/hello-world`
Hello, world! fib(6) = 8
```

Variables

```
1   fn main() {
2       let some_x = 5;
3       println!("some_x = {}", some_x);
4       some_x = 6;
5       println!("some_x = {}", some_x);
6   }
```

```
   Compiling hello-world v0.1.0 (/home/ruben/Projects/hello-world)
error[E0384]: cannot assign twice to immutable variable `some_x`
 --> src/main.rs:4:5
  |
2 |     let some_x = 5;
  |         ------
  |         |
  |         first assignment to `some_x`
  |         help: consider making this binding mutable: `mut some_x`
3 |     println!("some_x = {}", some_x);
4 |     some_x = 6;
  |     ^^^^^^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `hello-world` due to previous error
```

Variables

```
1   fn main() {
2       let mut some_x = 5;
3       println!("some_x = {}", some_x);
4       some_x = 6;
5       println!("some_x = {}", some_x);
6   }
```

```
   Compiling hello-world v0.1.0 (/home/ruben/Projects/hello-world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.26s
     Running `target/debug/hello-world`
some_x = 5
some_x = 6
```

Assigning a type to a variable

```
1   fn main() {
2     let x: i32 = 20;
3   }
```

- Rust is strongly and strictly typed
- Variables use type inference, so no need to specify a type
- We can be explicit in our types (and sometimes have to be)

Integers

Length	Signed	Unsigned
8 bit	`i8`	`u8`
16 bit	`i16`	`u16`
32 bit	`i32`	`u32`
64 bit	`i64`	`u64`
128 bit	`i128`	`u128`
architecture based	`isize`	`usize`

- Rust prefers explicit integer sizes
- Use `isize` and `usize` sparingly

Literals

```rust
fn main() {
  let x = 42; // decimal as i32
  let y = 42u64; // decimal as u64
  let z = 42_000; // underscore separator

  let u = 0xff; // hexadecimal
  let v = 0o77; // octal
  let w = 0b0100_1101; // binary
  let q = b'A'; // byte syntax (stored as u8)
}
```

Floating points and floating point literals

```
1   fn main() {
2     let x = 2.0; // f64
3     let y = 1.0f32; // f32
4   }
```

- `f32`: single precision (32 bit) floating point number
- `f64`: double precision (64 bit) floating point number

Numerical operations

```
fn main() {
  let sum = 5 + 10;
  let difference = 10 - 3;
  let mult = 2 * 8;
  let div = 2.4 / 3.5;
  let int_div = 10 / 3; // 3
  let remainder = 20 % 3;
}
```

- These expressions do overflow/underflow checking in debug
- In release builds these expressions are wrapping, for efficiency
- You cannot mix and match types here, not even between different integer types

```
fn main() {
  let invalid_div = 2.4 / 5;
  let invalid_add = 20u32 + 40u64;
}
```

Booleans and boolean operations

```
fn main() {
    let yes: bool = true;
    let no: bool = false;
    let not = !no;
    let and = yes && no;
    let or = yes || no;
}
```

Comparison operators

```
fn main() {
    let x = 10;
    let y = 20;
    x < y; // true
    x > y; // false
    x <= y; // true
    x >= y; // false
    x == y; // false
    x != y; // true
}
```

Note: as with numerical operators, you cannot compare different integer and float types with each other

```
fn main() {
    3.0 < 20; // invalid
    30u64 > 20i32; // invalid
}
```

Characters

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let heart_eyed_cat = '😻';
}
```

- A character is a 32 bit unicode scalar value
- Almost identical to a unicode code point, but excludes code points only valid in UTF-16 context
- Very much unlike C/C++ where char is 8 bits

Tuples

```
1   fn main() {
2       let tup: (i32, f32, char) = (1, 2.0, 'a');
3   }
```

- Group multiple values into a single compound type
- Fixed size
- Different types per element
- Create a tuple by writing a comma-separated list of values inside parentheses

```
1   fn main() {
2       let tup = (1, 2.0, 'Z');
3       let (a, b, c) = tup;
4       println!("({}, {}, {})", a, b, c);
5
6       let another_tuple = (true, 42);
7       println!("{}", another_tuple.1);
8   }
```

- Tuples can be destructured to get to their individual values
- You can also access individual elements using the period operator followed by a zero based index

Arrays

```
1   fn main() {
2       let arr: [i32; 3] = [1, 2, 3];
3       println!("{}", arr[0]);
4       let [a, b, c] = arr;
5       println!("[{}, {}, {}]", a, b, c);
6   }
```

- Also a collection of multiple values, but this time all of the same type
- Always a fixed length at compile time (similar to tuples)
- Use square brackets to access an individual value
- Destructuring as with tuples
- Rust always checks array bounds when accessing a value in an array

Control flow

```
fn main() {
    let mut x = 0;
    loop {
        if x < 5 {
            println!("x: {}", x);
            x += 1;
        } else {
            break;
        }
    }

    let mut y = 5;
    while y > 0 {
        y -= 1;
        println!("y: {}", x);
    }

    for i in [1, 2, 3, 4, 5] {
        println!("i: {}", i);
    }
}
```

Functions

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn returns_nothing() -> () {
    println!("Nothing to report");
}

fn also_returns_nothing() {
    println!("Nothing to report");
}
```

- The function boundary must always be explicitly annotated with types
- Within the function body type inference may be used
- A function that returns nothing has the return type *unit* (`()`)
- The function body contains a series of statements optionally ending with an expression

Statements

- Expressions evaluate to a resulting value
- Statements are instructions that perform some action and do not return a value
- A definition of any kind (function definition etc.)
- The `let var = value;` statement
- Almost everything else is an expression

Example statements

```
fn my_fun() {
  println!("{}", 5);
}
```

```
let x = 10;
```

```
let x = (let y = 10); // invalid
```

Expressions

- Expressions make up most of the Rust code you write
- Includes all control flow such as `if` and `while`
- Includes scoping braces (`{` and `}`)
- An expression can be turned into a statement by adding a semicolon (`;`)

```
fn main() {
  let y = {
    let x = 3;
    x + 1
  };
  println!("{}", y); // 4
}
```

Expressions - control flow

- Control flow expressions as a statement do not need to end with a semicolon if they return *unit* (`()`)
- Remember: A block/function can end with an expression, but it needs to have the correct type

```
fn main() {
    let y = 11;
    // if as an expression
    let x = if y < 10 {
        42
    } else {
        24
    };

    // if as a statement
    if x == 42 {
        println!("Foo");
    } else {
        println!("Bar");
    }
}
```

Scope

- We just mentioned the scope braces (`{` and `}`)
- Variable scopes are actually very important for how Rust works

```
fn main() {
  println!("Hello, {}", name);  // invalid: name is not yet defined
  let name = "world";  // from this point name is in scope
  println!("Hello, {}", name);
} // name goes out of scope
```

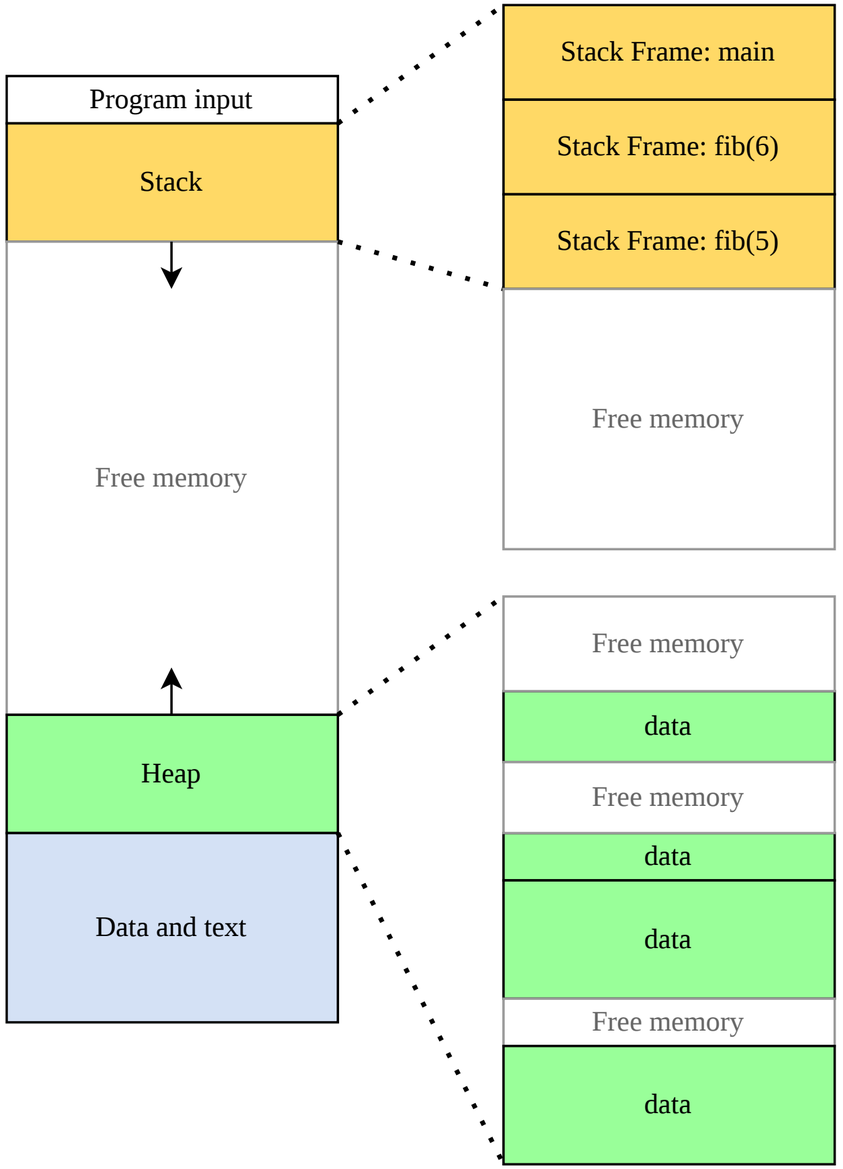
Scope

As soon as a scope ends, all variables for that scope can be removed from the stack

```
fn main() { // nothing in scope here
  let i = 10; // i is now in scope
  if i > 5 {
    let j = 20; // j is now also in scope
    println!("i = {}, j = {}", i, j);
  } // j is no longer in scope, i still remains
  println!("i = {}", i);
} // i is no longer in scope
```

Memory management

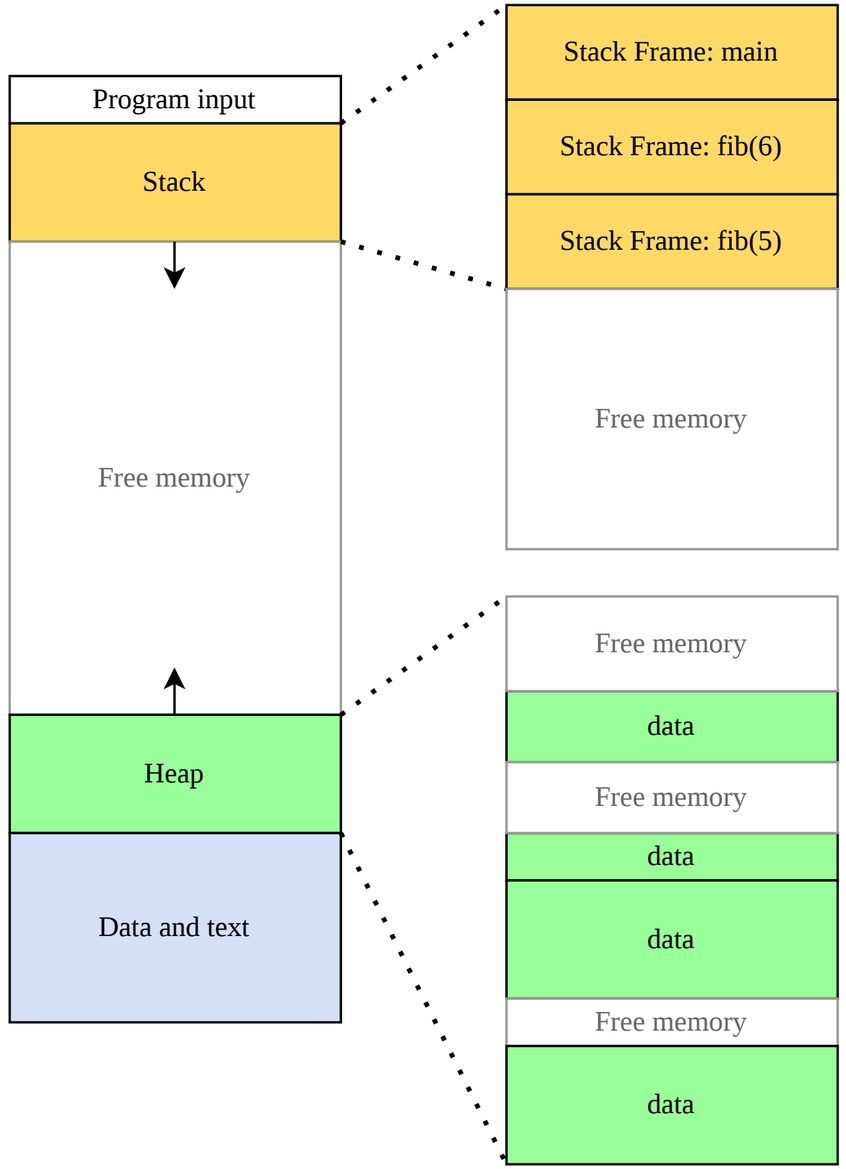
- Most of what we have seen so far is stack-based and small in size
- All these primitive types are `Copy`: create a copy on the stack every time we need them somewhere else
- We don’t want to pass a copy all the time
  - Large data that we do not want to copy
  - Modifying original data
- What about datastructures with a variable size?



Strings

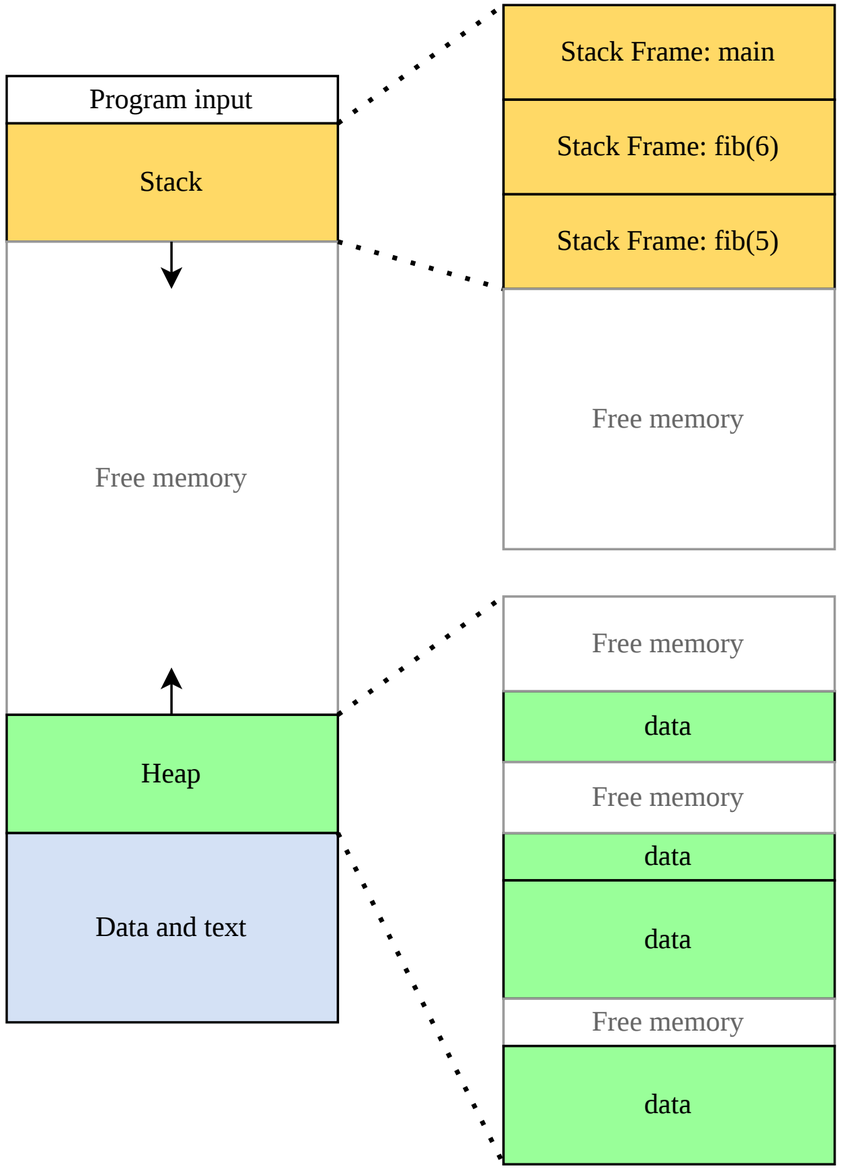
- A more interesting data structure in Rust is the `String`
- We have seen string literals already, but these are hardcoded and stored in the data and text segment of your program (e.g. `"Hello, world!"`)
- Strings meanwhile can be manipulated and resized at runtime

```
1   fn main() {
2       let mut s = String::from("Hello");
3       s.push_str(", world!");
4       println!("{}", s);
5   }
```



Memory cleanup

- For a mutable and growable `String`, we need to allocate memory on the heap
- A memory allocation algorithm takes care of us for this (i.e. `malloc()`)
- We need a way to give the memory back once we are done with it
- Option 1: Garbage Collection
- Option 2: Do it manually and make sure you don't make any mistakes
- Option 3: Ownership



Ownership

```
1   let x = 5;
2   let y = x;
3   println!("{}", x);
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in
4.00s
     Running `target/debug/playground`
5
```

```
1   let s1 = String::from("hello");
2   let s2 = s1;
3   println!("{}, world!", s1);
```

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:4:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`,
which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after
move

For more information about this error, try `rustc --explain
E0382`.
```

Ownership

- There is always ever only one owner of a value
- Once the owner gets removed from the stack, any associated values on the heap will be cleaned up as well
- Rust has *move semantics* for non-copy types

```
1   fn main() {
2       let s1 = String::from("hello");
3       let len = calculate_length(s1);
4       println!("The length of '{}' is {}.", s1, len);
5   }
6
7   fn calculate_length(s: String) -> usize {
8       s.len()
9   }
```

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:4:43
  |
2 | let s1 = String::from("hello");
  |     -- move occurs because `s1` has type `String`,
which does not implement the `Copy` trait
3 | let len = calculate_length(s1);
  |                            -- value moved here
4 | println!("The length of '{}' is {}.", s1, len);
  |                                       ^^ value borrowed
here after move

For more information about this error, try `rustc --explain
E0382`.
```

Moving out of a function

We can return a value to move it out of the function

```
fn main() {
    let s1 = String::from("hello");
    let (len, s1) = calculate_length(s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: String) -> (usize, String) {
    (s.len(), s)
}
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 5.42s
     Running `target/debug/playground`
The length of 'hello' is 5.
```

Clone

- Many types in Rust are `Clone`-able
- Use can use clone to create an explicit copy (in contrast to `Copy` which creates an implicit copy).
- Creating a clone can be expensive and could take a long time, so be careful
- Not very efficient if a clone is short-lived like in this example

```
fn main() {
    let x = String::from("hello");
    let len = get_length(x.clone());
    println!("{}: {}", x, len);
}

fn get_length(arg: String) -> usize {
    arg.len()
}
```

Borrowing

- We can make an analogy with real life: if somebody owns something you can borrow it from them, but eventually you have to give it back
- If a value is borrowed, it is not moved and the ownership stays with the original owner
- To borrow in rust, we create a *reference*

```
fn main() {
    let x = String::from("hello");
    let len = get_length(&x);
    println!("{}: {}", x, len);
}

fn get_length(arg: &String) -> usize {
    arg.len()
}
```

References (immutable)

```
1   fn main() {
2       let s = String::from("hello");
3       change(&s);
4       println!("{}", s);
5   }
6
7   fn change(some_string: &String) {
8       some_string.push_str(", world");
9   }
```

```
   Compiling playground v0.0.1 (/playground)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- help: consider changing this to be a mutable reference: `&mut String`
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be borrowed as
mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `playground` due to previous error
```

References (mutable)

```
1   fn main() {
2       let mut s = String::from("hello");
3       change(&mut s);
4       println!("{}", s);
5   }
6
7   fn change(some_string: &mut String) {
8       some_string.push_str(", world");
9   }
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 2.55s
     Running `target/debug/playground`
hello, world
```

- A mutable reference can even fully replace the original value
- To do this, you can use the dereference operator (`*`) to modify the value:

```
1   *some_string = String::from("Goodbye");
```

Rules for borrowing and references

- You may only ever have one mutable reference at the same time
- You may have any number of immutable references at the same time as long as there is no mutable reference
- References cannot *live* longer than their owners
- A reference will always at all times point to a valid value

Why? Because we no longer have to think about the nasty details of manual memory management and we can focus on what the program is really about.

Reference example

```rust
fn main() {
    let mut s = String::from("hello");
    let s1 = &s;
    let s2 = &s;
    let s3 = &mut s;
    println!("{} - {} - {}", s1, s2, s3);
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:5:14
  |
3 |     let s1 = &s;
  |              -- immutable borrow occurs here
4 |     let s2 = &s;
5 |     let s3 = &mut s;
  |              ^^^^^^ mutable borrow occurs here
6 |     println!("{} - {} - {}", s1, s2, s3);
  |                              -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `playground` due to previous error
```

Returning references

You can return references, but the value borrowed from must exist at least as long

```
1   fn give_me_a_ref() -> &String {
2       let s = String::from("Hello, world!");
3       &s
4   }
```

```
   Compiling playground v0.0.1 (/playground)
error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:23
  |
1 | fn give_me_a_ref() -> &String {
  |                       ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
help: consider using the `'static` lifetime
  |
1 | fn give_me_a_ref() -> &'static String {
  |                       ~~~~~~~~

For more information about this error, try `rustc --explain E0106`.
error: could not compile `playground` due to previous error
```

Returning references

You can return references, but the value borrowed from must exist at least as long

```
fn give_me_a_ref(input: &(String, i32)) -> &String {
    &input.0
}
```

```
fn give_me_a_value() -> String {
    let s = String::from("Hello, world!");
    s
}
```

A special kind of reference: slices

References aren’t limited to single values, with a slice we can reference a contiguous sequence of elements.

- With contiguous we mean: something layout out in memory in a repeating pattern
- How would a function look that returns a reference to a part of an array?

```
fn middle(s: &[i32; 5]) -> ? {
    // ?
}
```

Slices

- In general the syntax for a slice is `[T]` with `T` being a specific type
- Slices are never owned, so are only ever seen as references (`&[T]`)

```
fn middle(s: &[i32]) -> &[i32] {
    &s[1..4]
}

fn main() {
    let data = [1, 2, 3, 4, 5];
    let mid = middle(&data);
    println!("[{}, {}, {}]", mid[0], mid[1], mid[2]);
}
```

Structuring data (structs)

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

fn main() {
    let p = Point {
        x: 2,
        y: 1,
    };
    println!("({}, {})", p.x, p.y);
}
```

- We've previously seen tuples to store multiple values within a single type
- A `struct` is an alternative way to store multiple values within a single type
- This time you can access fields with a period followed by the name of the field, instead of having to remember a field number

Debug

```
1   #[derive(Debug)]
2   struct Point {
3       x: i32,
4       y: i32,
5   }
6   fn main() {
7       let p = Point {
8           x: 2,
9           y: 1,
10      };
11      dbg!(&p);
12      println!("my location: {:?}", p);
13  }
```

```
     Running `target/debug/playground`
[src/main.rs:11] &p = Point {
    x: 2,
    y: 1,
}
my location: Point { x: 2, y: 1 }
```

Methods

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn new() -> Rectangle {
        Rectangle { width: 10, height: 10 }
    }
    fn area(&self) -> u32 { self.width * self.height }
}

fn main() {
    let rect1 = Rectangle::new();
    println!("Area: {}", rect1.area());
}
```

- We can add one or more `impl` blocks to add methods to our structs
- If there is no `self` parameter then the function is not a method but an *associated function*
- The type of `self` determines how the method will use its value
- The `self` parameter is also called the *receiver*

Methods

```
#[derive(Debug)]
struct Rectangle {
  width: u32,
  height: u32,
}
impl Rectangle {
    fn new() -> Rectangle {
        Rectangle {
            width: 10,
            height: 10,
        }
    }

    fn into_area(self) -> u32 {
        self.width * self.height
    }
}
fn main() {
    let rect = Rectangle::new();
    let area = rect.into_area();
    println!("{:?}: {}", rect, area);
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `rect`
  --> src/main.rs:12:26
   |
10 |     let rect = Rectangle::new();
   |         ---- move occurs because `rect` has type
`Rectangle`, which does not implement the `Copy` trait
11 |     let area = rect.into_area();
   |                     ----------- `rect` moved due to
this method call
12 |     println!("{:?}: {}", rect, area);
   |                          ^^^^ value borrowed here
after move
   |
note: this function takes ownership of the receiver `self`,
which moves `rect`
  --> src/main.rs:6:18
   |
6  |     fn into_area(self) -> u32 { self.width *
self.height }
   |                  ^^^^

For more information about this error, try `rustc --explain
E0382`.
```

Enumerations

One of Rust's most powerful features are `enum`s. An enum is a type that consists of zero (!) or more *variants*.

```
enum IpKind {
    Ipv4,
    Ipv6,
}

fn main() {
    let old = IpKind::Ipv4;
    let new = IpKind::Ipv6;
}
```

Enums

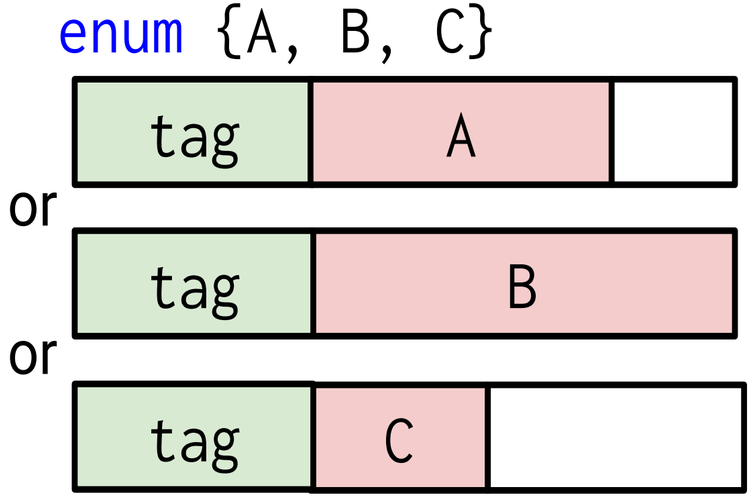
```
#[derive(Debug)]
enum IpKind {
    Ipv4,
    Ipv6,
}

#[derive(Debug)]
struct IpAddress {
    kind: IpKind,
    address: String,
}

fn main() {
    let home = IpAddress {
        kind: IpKind::Ipv4,
        address: String::from("127.0.0.1"),
    };
    println!("{:?}", home);
}
```

- When we want to add some data we might be tempted to use something like the above example
- But this way it’s easy to accidentaly add an ipv6 address to something we said was ipv4

Enums with variant data

```
1   #[derive(Debug)]
2   enum IpAddress {
3     Ipv4(u8, u8, u8, u8),
4     Ipv6(u16, u16, u16, u16, u16, u16, u16, u16),
5   }
6
7   fn main() {
8     let v4 = IpAddress::Ipv4(127, 0, 0, 1);
9     let v6 = IpAddress::Ipv6(0, 0, 0, 0, 0, 0, 0, 1);
10    println!("My home is {:?} and {:?}", v4, v6)
11  }
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in
8.36s
     Running `target/debug/playground`
My home is Ipv4(127, 0, 0, 1) and Ipv6(0, 0, 0, 0, 0, 0, 0,
1)
```



- An enum always takes as much memory as the largest variant
- Often not as bad as it sounds, remember large growing data structures like `String` store most of their data on the heap

Option

Enums are used heavily in Rust. One example is the `Option`.

- An `Option` is an alternative to nullability
- You must always explicitly handle these 'nullable' types

```
enum Option<T> {
  Some(T),
  None,
}

fn main() {
  let some_number = Some(5);
  let some_string = Some(String::from("Hello world"));
  let absent_number: Option<i32> = None;
}
```

```
fn maybe_add(a: Option<i32>, b: Option<i32>) -> Option<i32> {
  todo!();
}
```

Pattern matching

```
fn maybe_add(a: Option<i32>, b: Option<i32>) -> Option<i32> {
    match (a, b) {
        (Some(val_a), Some(val_b)) => Some(val_a + val_b),
        _ => None,
    }
}
```

- The `match` control flow expression is used for pattern matching
- A match expression must be exhaustive
- The underscore (`_`) pattern can be used as a catch-all

Pattern matching with if and while

An `if let` expression can be used as an alternative for exhaustive pattern matching using `match`.

```
let x = Some(42);
if let Some(x) = maybe_x {
    println!("{}", x);
}
```

```
while let Some(x) = get_next() {
    // do something with x
}
```

Result and errors

Another common enum is `Result<T, E>` and it's two variants `Ok(_)` and `Err(_)`.

- Some functions are fallible, i.e. what they are doing might fail
- `Result` captures this idea in an enum
- The `Ok()` variant indicates success and contains the result of the computation
- The `Err()` variant indicates that an error has occured

```
struct InvalidInput;

fn maybe_add(a: Option<i32>, b: Option<i32>) -> Result<i32, InvalidInput> {
    match (a, b) {
        (Some(val_a), Some(val_b)) => Ok(val_a + val_b),
        _ => Err(InvalidInput),
    }
}
```

Try operator

Working with Results and Errors is done so often, that a special control flow operator was introduced, the try operator (`?`).

- Let’s say we have a function that returns a Result that calls another function returning a Result
- We could handle this with a simple match expression

```
fn maybe_works() -> Result<i32, ()> {
  todo!()
}

fn calls_maybe_works() -> Result<i32, ()> {
  match maybe_works() {
    Ok(o) => Ok(o * 2),
    Err(e) => Err(e),
  }
}
```

Try operator

Working with Results and Errors is done so often, that a special control flow operator was introduced, the try operator (`?`).

- Let’s say we have a function that returns a Result that calls another function returning a Result
- We can write this down more concisely with the try operator

```
fn maybe_works() -> Result<i32, ()> {
  todo!()
}

fn calls_maybe_works() -> Result<i32, ()> {
  let int = maybe_works()?;
  Ok(int * 2)
}
```

Shared behavior: traits

Traits allow you to define some behavior and implement that behavior for different types.

```
trait Animal {
    fn make_sound(&self);
    fn into_picture(self) -> Picture;
}
```

- We have already seen some traits during our tour so far: `Copy`, `Clone`, `Debug`
- Traits indicate behavior, behavior is implemented using methods
- You can implement a trait for any type, including primitive types, tuples and arrays

Implementing existing traits - derives

Some traits can be implemented using the derive macro attribute.

```
#[derive(Debug, Clone, PartialEq, Eq)]
struct Article {
    title: String,
    content: String,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
enum Seasons {
    Spring,
    Summer,
    Autumn,
    Winter,
}
```

Implementing existing traits - impl block

Most of the time you will need to define the behavior yourself.

```
enum Season { Spring, Summer, Autumn, Winter }

impl Default for Season {
    fn default() -> Self {
        Season::Summer
    }
}
```

Trait bounds

When defining a generic type parameter, we can limit the allowed types based on traits.

```
fn print_printable<T: Display>(printable: T) {
    println!("{}", printable);
}
```

or

```
fn print_printable<T>(printable: T) where T: Display {
    println!("{}", printable);
}
```

Modules and crates

- When you start adding all those traits, structs and implementation blocks together, you will need some way to organise them.
- Rust compiles at the level of *crates*: you can think of these as packages
- A crate may consist of multiple files, containing *modules*
- Modules are a way to organise your own code

```
1   mod data {
2       // `./data/user.rs` or `./data/user/mod.rs`
3       mod user;
4
5       struct Example;
6   }
7
8   fn main() {
9       let e = data::Example;
10  }
```

```
   Compiling playground v0.0.1 (/playground)
error[E0603]: unit struct `Example` is private
 --> src/main.rs:9:19
  |
9 |     let e = data::Example;
  |                   ^^^^^^^ private unit struct
  |
note: the unit struct `Example` is defined here
 --> src/main.rs:5:5
  |
5 |     struct Example;
  |     ^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain
E0603`.
```

Visibility

Rust is very private by default, but sometimes you want to open up.

- Almost everything in Rust is private, only the defining module and any child modules may access the item
- You can mark types as `pub` to make them available to everyone
- You can also mark something as `pub(crate)` to make something available within your crate but not outside of it

```
pub mod models {
  pub struct User {
    pub username: String,
    pub(crate) access_level: AccessLevel,
    password_hash: String,
  }
}
```

A tour of the standard library

We've already seen quite a bit, but the standard library offers more.

Dependencies

Not everything is in the standard library. But external packages can be included in the form of crates. You manage these in `cargo.toml`.

```
[package]
name = "example"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
serde_json = "1.0"
```

- You can also add external crates by referencing a directory or git repository
- You can find publicly available dependencies at crates.io
- You can read their documentation at docs.rs
- The crates you add as dependencies appear as if they are modules at the top of your own module structure

More links

Rust has very detailed documentation and lots of places to read more about it.

- Official rust book
- Official reference including detailed syntax of all language features
- Standard library docs
- Rust by example
- docs.rs for other crate documentation
- crates.io for downloading other crates
- lib.rs is a decent alternative for crate discovery

Practice time