Generic Functions

Rust supports generics, which lets you abstract algorithms or data structures (such as sorting or a binary tree) over the types used or stored.

```
fn pick<T>(cond: bool, left: T, right: T) -> T {
    if cond { left } else { right }
}

fn main() {
    println!("picked a number: {:?}", pick(true, 222, 333));
    println!("picked a string: {:?}", pick(false, 'L', 'R'));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- It can be helpful to show the monomorphized versions of `pick`, either before talking about the generic `pick` in order to show how generics can reduce code duplication, or after talking about generics to show how monomorphization works.

```
fn pick_i32(cond: bool, left: i32, right: i32) -> i32 {
    if cond { left } else { right }
}

fn pick_char(cond: bool, left: char, right: char) -> char {
    if cond { left } else { right }
}
```

- Rust infers a type for T based on the types of the arguments and return value.
- In this example we only use the primitive types `i32` and `char` for `T`, but we can use any type here, including user-defined types:

```
struct Foo {
    val: u8,
}

pick(false, Foo { val: 7 }, Foo { val: 99 });
```

- This is similar to C++ templates, but Rust partially compiles the generic function immediately, so that function must be valid for all types matching the constraints. For example, try modifying `pick` to return `left + right` if `cond` is false. Even if only the `pick` instantiation with integers is used, Rust still considers it invalid. C++ would let you do this.
- Generic code is turned into non-generic code based on the call sites. This is a zero-cost abstraction: you get exactly the same result as if you had hand-coded the data structures without the abstraction.

Trait Bounds

When working with generics, you often want to require the types to implement some trait, so that you can call this trait's methods.

You can do this with `T: Trait`:

```
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
}

struct NotCloneable;

fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{pair:?}");
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

- Try making a `NotCloneable` and passing it to `duplicate`.
- When multiple traits are necessary, use `+` to join them.
- Show a `where` clause, students will encounter it when reading code.

```
fn duplicate<T>(a: T) -> (T, T)
where
    T: Clone,
{
    (a.clone(), a.clone())
}
```

  - It declutters the function signature if you have many parameters.
  - It has additional features making it more powerful.
    - If someone asks, the extra feature is that the type on the left of ":" can be arbitrary, like `Option<T>`.
- Note that Rust does not (yet) support specialization. For example, given the original `duplicate`, it is invalid to add a specialized `duplicate(a: u32)`.

Generic Data Types

You can use generics to abstract over the concrete field type. Returning to the exercise for the previous segment:

```
pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}

impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- *Q:* Why `L` is specified twice in `impl<L: Logger> .. VerbosityFilter<L>` ? Isn't that redundant?
  - This is because it is a generic implementation section for generic type. They are independently generic.
  - It means these methods are defined for any `L` .
  - It is possible to write `impl VerbosityFilter<StderrLogger> { .. }` .
    - `VerbosityFilter` is still generic and you can use `VerbosityFilter<f64>` , but methods in this block will only be available for `VerbosityFilter<StderrLogger>` .
- Note that we don't put a trait bound on the `VerbosityFilter` type itself. You can put bounds there as well, but generally in Rust we only put the trait bounds on the impl blocks.

Generic Traits

Traits can also be generic, just like types and functions. A trait's parameters get concrete types when it is used. For example the `From<T>` trait is used to define type conversions:

```
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}
```

```
#[derive(Debug)]
struct Foo(String);

impl From<u32> for Foo {
    fn from(from: u32) -> Foo {
        Foo(format!("Converted from integer: {from}"))
    }
}

impl From<bool> for Foo {
    fn from(from: bool) -> Foo {
        Foo(format!("Converted from bool: {from}"))
    }
}

fn main() {
    let from_int = Foo::from(123);
    let from_bool = Foo::from(true);
    dbg!(from_int);
    dbg!(from_bool);
}
```

▼ *Speaker Notes*

- The `From` trait will be covered later in the course, but its definition in the `std` docs is simple, and copied here for reference.
- Implementations of the trait do not need to cover all possible type parameters. Here, `Foo::from("hello")` would not compile because there is no `From<&str>` implementation for `Foo`.
- Generic traits take types as "input", while associated types are a kind of "output" type. A trait can have multiple implementations for different input types.
- In fact, Rust requires that at most one implementation of a trait match for any type T. Unlike some other languages, Rust has no heuristic for choosing the "most specific" match. There is work on adding this support, called specialization.

`impl Trait`

Similar to trait bounds, an `impl Trait` syntax can be used in function arguments and return values:

```
// Syntactic sugar for:
//   fn add_42_millions<T: Into<i32>>(x: T) -> i32 {
fn add_42_millions(x: impl Into<i32>) -> i32 {
    x.into() + 42_000_000
}

fn pair_of(x: u32) -> impl std::fmt::Debug {
    (x + 1, x - 1)
}

fn main() {
    let many = add_42_millions(42_i8);
    dbg!(many);
    let many_more = add_42_millions(10_000_000);
    dbg!(many_more);
    let debuggable = pair_of(27);
    dbg!(debuggable);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

`impl Trait` allows you to work with types which you cannot name. The meaning of `impl Trait` is a bit different in the different positions.

- For a parameter, `impl Trait` is like an anonymous generic parameter with a trait bound.
- For a return type, it means that the return type is some concrete type that implements the trait, without naming the type. This can be useful when you don't want to expose the concrete type in a public API.

  Inference is hard in return position. A function returning `impl Foo` picks the concrete type it returns, without writing it out in the source. A function returning a generic type like `collect<B>() -> B` can return any type satisfying `B`, and the caller may need to choose one, such as with `let x: Vec<_> = foo.collect()` or with the turbofish, `foo.collect::<Vec<_>>()`.

What is the type of `debuggable`? Try `let debuggable: () = ..` to see what the error message shows.

dyn Trait

In addition to using traits for static dispatch via generics, Rust also supports using them for type-erased, dynamic dispatch via trait objects:

```
struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    lives: i8,
}

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

impl Pet for Cat {
    fn talk(&self) -> String {
        String::from("Miau!")
    }
}

// Uses generics and static dispatch.
fn generic(pet: &impl Pet) {
    println!("Hello, who are you? {}", pet.talk());
}

// Uses type-erasure and dynamic dispatch.
fn dynamic(pet: &dyn Pet) {
    println!("Hello, who are you? {}", pet.talk());
}

fn main() {
    let cat = Cat { lives: 9 };
    let dog = Dog { name: String::from("Fido"), age: 5 };

    generic(&cat);
    generic(&dog);

    dynamic(&cat);
    dynamic(&dog);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Generics, including `impl Trait`, use monomorphization to create a specialized instance of the function for each different type that the generic is instantiated with. This means that calling a trait method from within a generic function still uses static dispatch, as the compiler has full type information and can resolve which type's trait implementation to use.
- When using `dyn Trait`, it instead uses dynamic dispatch through a virtual method table (vtable). This means that there's a single version of `fn dynamic` that is used regardless of what type of `Pet` is passed in.
- When using `dyn Trait`, the trait object needs to be behind some kind of indirection. In this

- At runtime, a `&dyn Pet` is represented as a "fat pointer", i.e. a pair of two pointers: One pointer points to the concrete object that implements `Pet`, and the other points to the vtable for the trait implementation for that type. When calling the `talk` method on `&dyn Pet` the compiler looks up the function pointer for `talk` in the vtable and then invokes the function, passing the pointer to the `Dog` or `Cat` into that function. The compiler doesn't need to know the concrete type of the `Pet` in order to do this.
- A `dyn Trait` is considered to be "type-erased", because we no longer have compile-time knowledge of what the concrete type is.

Exercise: Generic `min`

In this short exercise, you will implement a generic `min` function that determines the minimum of two values, using the `Ord` trait.

```
use std::cmp::Ordering;

// TODO: implement the `min` function used in the tests.

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 10 minutes.

- Show students the `Ord` trait and `Ordering` enum.

Solution

```rust
use std::cmp::Ordering;

fn min<T: Ord>(l: T, r: T) -> T {
    match l.cmp(&r) {
        Ordering::Less | Ordering::Equal => l,
        Ordering::Greater => r,
    }
}

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}
```

Welcome Back

Including 10 minute breaks, this session should take about 2 hours and 50 minutes. It contains:

Segment	Duration
Standard Library Types	1 hour
Closures	30 minutes
Standard Library Traits	1 hour

Standard Library Types

This segment should take about 1 hour. It contains:

Slide	Duration
Standard Library	3 minutes
Documentation	5 minutes
Option	10 minutes
Result	5 minutes
String	5 minutes
Vec	5 minutes
HashMap	5 minutes
Exercise: Counter	20 minutes

▼ *Speaker Notes*

For each of the slides in this section, spend some time reviewing the documentation pages, highlighting some of the more common methods.

Standard Library

Rust comes with a standard library which helps establish a set of common types used by Rust libraries and programs. This way, two libraries can work together smoothly because they both use the same `String` type.

In fact, Rust contains several layers of the Standard Library: `core`, `alloc` and `std`.

- `core` includes the most basic types and functions that don't depend on `libc`, allocator or even the presence of an operating system.
- `alloc` includes types which require a global heap allocator, such as `Vec`, `Box` and `Arc`.
- Embedded Rust applications often only use `core`, and sometimes `alloc`.

Documentation

Rust comes with extensive documentation. For example:

- All of the details about loops.
- Primitive types like `u8`.
- Standard library types like `Option` or `BinaryHeap`.

Use `rustup doc --std` or https://std.rs to view the documentation.

In fact, you can document your own code:

```
/// Determine whether the first argument is divisible by the second argument.
///
/// If the second argument is zero, the result is false.
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;
    }
    lhs % rhs == 0
}
```

The contents are treated as Markdown. All published Rust library crates are automatically documented at `docs.rs` using the rustdoc tool. It is idiomatic to document all public items in an API using this pattern.

To document an item from inside the item (such as inside a module), use `//!` or `/*! .. */`, called "inner doc comments":

```
//! This module contains functionality relating to divisibility of integers.
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Show students the generated docs for the `rand` crate at https://docs.rs/rand.

Option

We have already seen some use of `Option<T>`. It stores either a value of type `T` or nothing. For example, `String::find` returns an `Option<usize>`.

```
fn main() {
    let name = "Löwe 老虎 Léopard Gepardi";
    let mut position: Option<usize> = name.find('é');
    dbg!(position);
    assert_eq!(position.unwrap(), 14);
    position = name.find('Z');
    dbg!(position);
    assert_eq!(position.expect("Character not found"), 0);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- `Option` is widely used, not just in the standard library.
- `unwrap` will return the value in an `Option`, or panic. `expect` is similar but takes an error message.
    - You can panic on None, but you can't "accidentally" forget to check for None.
    - It's common to `unwrap` / `expect` all over the place when hacking something together, but production code typically handles `None` in a nicer fashion.
- The "niche optimization" means that `Option<T>` often has the same size in memory as `T`, if there is some representation that is not a valid value of T. For example, a reference cannot be NULL, so `Option<&T>` automatically uses NULL to represent the `None` variant, and thus can be stored in the same memory as `&T`.

Result

`Result` is similar to `Option`, but indicates the success or failure of an operation, each with a different enum variant. It is generic: `Result<T, E>` where `T` is used in the `Ok` variant and `E` appears in the `Err` variant.

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Dear diary: {contents} ({bytes} bytes)");
            } else {
                println!("Could not read file content");
            }
        }
        Err(err) => {
            println!("The diary could not be opened: {err}");
        }
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- As with `Option`, the successful value sits inside of `Result`, forcing the developer to explicitly extract it. This encourages error checking. In the case where an error should never happen, `unwrap()` or `expect()` can be called, and this is a signal of the developer intent too.
- `Result` documentation is a recommended read. Not during the course, but it is worth mentioning. It contains a lot of convenience methods and functions that help functional-style programming.
- `Result` is the standard type to implement error handling as we will see on Day 4.

String

`String` is a growable UTF-8 encoded string:

```
fn main() {
    let mut s1 = String::new();
    s1.push_str("Hello");
    println!("s1: len = {}, capacity = {}", s1.len(), s1.capacity());

    let mut s2 = String::with_capacity(s1.len() + 1);
    s2.push_str(&s1);
    s2.push('!');
    println!("s2: len = {}, capacity = {}", s2.len(), s2.capacity());

    let s3 = String::from("🇨🇭");
    println!("s3: len = {}, number of chars = {}", s3.len(), s3.chars().count());
}
```

`String` implements `Deref<Target = str>`, which means that you can call all `str` methods on a `String`.

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `String::new` returns a new empty string, use `String::with_capacity` when you know how much data you want to push to the string.
- `String::len` returns the size of the `String` in bytes (which can be different from its length in characters).
- `String::chars` returns an iterator over the actual characters. Note that a `char` can be different from what a human will consider a "character" due to grapheme clusters.
- When people refer to strings they could either be talking about `&str` or `String`.
- When a type implements `Deref<Target = T>`, the compiler will let you transparently call methods from `T`.
  - We haven't discussed the `Deref` trait yet, so at this point this mostly explains the structure of the sidebar in the documentation.
  - `String` implements `Deref<Target = str>` which transparently gives it access to `str`'s methods.
  - Write and compare `let s3 = s1.deref();` and `let s3 = &*s1;`.
- `String` is implemented as a wrapper around a vector of bytes, many of the operations you see supported on vectors are also supported on `String`, but with some extra guarantees.
- Compare the different ways to index a `String`:
  - To a character by using `s3.chars().nth(i).unwrap()` where `i` is in-bound, out-of-bounds.
  - To a substring by using `s3[0..4]`, where that slice is on character boundaries or not.
- Many types can be converted to a string with the `to_string` method. This trait is automatically implemented for all types that implement `Display`, so anything that can be formatted can also be converted to a string.

Vec

`Vec` is the standard resizable heap-allocated buffer:

```
fn main() {
    let mut v1 = Vec::new();
    v1.push(42);
    println!("v1: len = {}, capacity = {}", v1.len(), v1.capacity());

    let mut v2 = Vec::with_capacity(v1.len() + 1);
    v2.extend(v1.iter());
    v2.push(9999);
    println!("v2: len = {}, capacity = {}", v2.len(), v2.capacity());

    // Canonical macro to initialize a vector with elements.
    let mut v3 = vec![0, 0, 1, 2, 3, 4];

    // Retain only the even elements.
    v3.retain(|x| x % 2 == 0);
    println!("{v3:?}");

    // Remove consecutive duplicates.
    v3.dedup();
    println!("{v3:?}");
}
```

`Vec` implements `Deref<Target = [T]>`, which means that you can call slice methods on a `Vec`.

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `Vec` is a type of collection, along with `String` and `HashMap`. The data it contains is stored on the heap. This means the amount of data doesn't need to be known at compile time. It can grow or shrink at runtime.
- Notice how `Vec<T>` is a generic type too, but you don't have to specify `T` explicitly. As always with Rust type inference, the `T` was established during the first `push` call.
- `vec![...]` is a canonical macro to use instead of `Vec::new()` and it supports adding initial elements to the vector.
- To index the vector you use `[ ]`, but they will panic if out of bounds. Alternatively, using `get` will return an `Option`. The `pop` function will remove the last element.

HashMap

Standard hash map with protection against HashDoS attacks:

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

fn main() {
    let mut page_counts = HashMap::new();
    page_counts.insert("Adventures of Huckleberry Finn", 207);
    page_counts.insert("Grimms' Fairy Tales", 751);
    page_counts.insert("Pride and Prejudice", 303);

    if !page_counts.contains_key("Les Misérables") {
        println!(
            "We know about {} books, but not Les Misérables.",
            page_counts.len()
        );
    }

    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        match page_counts.get(book) {
            Some(count) => println!("{book}: {count} pages"),
            None => println!("{book} is unknown."),
        }
    }

    // Use the .entry() method to insert a value if nothing is found.
    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        let page_count: &mut i32 = page_counts.entry(book).or_insert(0);
        *page_count += 1;
    }

    dbg!(page_counts);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `HashMap` is not defined in the prelude and needs to be brought into scope.
- Try the following lines of code. The first line will see if a book is in the hashmap and if not return an alternative value. The second line will insert the alternative value in the hashmap if the book is not found.

```
let pc1 = page_counts
    .get("Harry Potter and the Sorcerer's Stone")
    .unwrap_or(&336);
let pc2 = page_counts
    .entry("The Hunger Games")
    .or_insert(374);
```

- Unlike `vec!`, there is unfortunately no standard `hashmap!` macro.
  - Although, since Rust 1.56, HashMap implements `From<[(K, V); N]>`, which allows us to easily initialize a hash map from a literal array:

```
let page_counts = HashMap::from([
  ("Harry Potter and the Sorcerer's Stone".to_string(), 336),
  ("The Hunger Games".to_string(), 374),
]);
```

- Alternatively HashMap can be built from any `Iterator` which yields key-value tuples.
- This type has several “method-specific” return types, such as `std::collections::hash_map::Keys`. These types often appear in searches of the Rust docs. Show students the docs for this type, and the helpful link back to the `keys` method.

Exercise: Counter

In this exercise you will take a very simple data structure and make it generic. It uses a `std::collections::HashMap` to keep track of which values have been seen and how many times each one has appeared.

The initial version of `Counter` is hard coded to only work for `u32` values. Make the struct and its methods generic over the type of value being tracked, that way `Counter` can track any type of value.

If you finish early, try using the `entry` method to halve the number of hash lookups required to implement the `count` method.

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

/// Counter counts the number of times each value of type T has been seen.
struct Counter {
    values: HashMap<u32, u64>,
}

impl Counter {
    /// Create a new Counter.
    fn new() -> Self {
        Counter {
            values: HashMap::new(),
        }
    }

    /// Count an occurrence of the given value.
    fn count(&mut self, value: u32) {
        if self.values.contains_key(&value) {
            *self.values.get_mut(&value).unwrap() += 1;
        } else {
            self.values.insert(value, 1);
        }
    }

    /// Return the number of times the given value has been seen.
    fn times_seen(&self, value: u32) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);

    for i in 10..20 {
        println!("saw {} values equal to {}", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("apple");
    strctr.count("orange");
    strctr.count("apple");
    println!("got {} apples", strctr.times_seen("apple"));
}
```

Solution

```
use std::collections::HashMap;
use std::hash::Hash;

/// Counter counts the number of times each value of type T has been seen.
struct Counter<T> {
    values: HashMap<T, u64>,
}

impl<T: Eq + Hash> Counter<T> {
    /// Create a new Counter.
    fn new() -> Self {
        Counter { values: HashMap::new() }
    }

    /// Count an occurrence of the given value.
    fn count(&mut self, value: T) {
        *self.values.entry(value).or_default() += 1;
    }

    /// Return the number of times the given value has been seen.
    fn times_seen(&self, value: T) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);

    for i in 10..20 {
        println!("saw {} values equal to {}", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("apple");
    strctr.count("orange");
    strctr.count("apple");
    println!("got {} apples", strctr.times_seen("apple"));
}
```

Closures

This segment should take about 30 minutes. It contains:

Slide	Duration
Closure Syntax	3 minutes
Capturing	5 minutes
Closure Traits	10 minutes
Exercise: Log Filter	10 minutes

Closure Syntax

Closures are created with vertical bars: `|..| ..`.

```
fn main() {
    // Argument and return type can be inferred for lightweight syntax:
    let double_it = |n| n * 2;
    dbg!(double_it(50));

    // Or we can specify types and bracket the body to be fully explicit:
    let add_1f32 = |x: f32| -> f32 { x + 1.0 };
    dbg!(add_1f32(50.));
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- The arguments go between the `|..|`. The body can be surrounded by `{ .. }`, but if it is a single expression these can be omitted.
- Argument types are optional, and are inferred if not given. The return type is also optional, but can only be written if using `{ .. }` around the body.
- The examples can both be written as mere nested functions instead – they do not capture any variables from their lexical environment. We will see captures next.

More to Explore

- The ability to store functions in variables doesn't just apply to closures, regular functions can be put in variables and then invoked the same way that closures can: Example in the playground.
    - The linked example also demonstrates that closures that don't capture anything can also coerce to a regular function pointer.

Capturing

A closure can capture variables from the environment where it was defined.

```
fn main() {
    let max_value = 5;
    let clamp = |v| {
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- By default, a closure captures values by reference. Here `max_value` is captured by `clamp`, but still available to `main` for printing. Try making `max_value` mutable, changing it, and printing the clamped values again. Why doesn't this work?
- If a closure mutates values, it will capture them by mutable reference. Try adding `max_value += 1` to `clamp`.
- You can force a closure to move values instead of referencing them with the `move` keyword. This can help with lifetimes, for example if the closure must outlive the captured values (more on lifetimes later).

  This looks like `move |v| ..`. Try adding this keyword and see if `main` can still access `max_value` after defining `clamp`.
- By default, closures will capture each variable from an outer scope by the least demanding form of access they can (by shared reference if possible, then exclusive reference, then by move). The `move` keyword forces capture by value.

Closure traits

Closures or lambda expressions have types which cannot be named. However, they implement special `Fn`, `FnMut`, and `FnOnce` traits:

The special types `fn(..) -> T` refer to function pointers - either the address of a function, or a closure that captures nothing.

```
fn apply_and_log(
    func: impl FnOnce(&'static str) -> String,
    func_name: &'static str,
    input: &'static str,
) {
    println!("Calling {func_name}({input}): {}", func(input))
}

fn main() {
    let suffix = "-itis";
    let add_suffix = |x| format!("{x}{suffix}");
    apply_and_log(&add_suffix, "add_suffix", "senior");
    apply_and_log(&add_suffix, "add_suffix", "appendix");

    let mut v = Vec::new();
    let mut accumulate = |x| {
        v.push(x);
        v.join("/")
    };
    apply_and_log(&mut accumulate, "accumulate", "red");
    apply_and_log(&mut accumulate, "accumulate", "green");
    apply_and_log(&mut accumulate, "accumulate", "blue");

    let take_and_reverse = |prefix| {
        let mut acc = String::from(prefix);
        acc.push_str(&v.into_iter().rev().collect::<Vec<_>>().join("/"));
        acc
    };
    apply_and_log(take_and_reverse, "take_and_reverse", "reversed: ");
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

An `Fn` (e.g. `add_suffix`) neither consumes nor mutates captured values. It can be called needing only a shared reference to the closure, which means the closure can be executed repeatedly and even concurrently.

An `FnMut` (e.g. `accumulate`) might mutate captured values. The closure object is accessed via exclusive reference, so it can be called repeatedly but not concurrently.

If you have an `FnOnce` (e.g. `take_and_reverse`), you may only call it once. Doing so consumes the closure and any values captured by move.

`FnMut` is a subtype of `FnOnce`. `Fn` is a subtype of `FnMut` and `FnOnce`. I.e. you can use an `FnMut` wherever an `FnOnce` is called for, and you can use an `Fn` wherever an `FnMut` or `FnOnce` is called for.

When you define a function that takes a closure, you should take `FnOnce` if you can (i.e. you call it once), or `FnMut` else, and last `Fn`. This allows the most flexibility for the caller.

In contrast, when you have a closure, the most flexible you can have is `Fn` (which can be passed to a consumer of any of the 3 closure traits), then `FnMut`, and lastly `FnOnce`.

The compiler also infers `Copy` (e.g. for `add_suffix`) and `Clone` (e.g. `take_and_reverse`), depending on what the closure captures. Function pointers (references to `fn` items) implement `Copy` and `Fn`.

Exercise: Log Filter

Building on the generic logger from this morning, implement a `Filter` which uses a closure to filter log messages, sending those which pass the filtering predicate to an inner logger.

```
pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

// TODO: Define and implement `Filter`.

fn main() {
    let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("yikes"));
    logger.log(5, "FYI");
    logger.log(1, "yikes, something went wrong");
    logger.log(2, "uhoh");
}
```

Solution

```rust
pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages matching a filtering predicate.
struct Filter<L, P> {
    inner: L,
    predicate: P,
}

impl<L, P> Filter<L, P>
where
    L: Logger,
    P: Fn(u8, &str) -> bool,
{
    fn new(inner: L, predicate: P) -> Self {
        Self { inner, predicate }
    }
}
impl<L, P> Logger for Filter<L, P>
where
    L: Logger,
    P: Fn(u8, &str) -> bool,
{
    fn log(&self, verbosity: u8, message: &str) {
        if (self.predicate)(verbosity, message) {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("yikes"));
    logger.log(5, "FYI");
    logger.log(1, "yikes, something went wrong");
    logger.log(2, "uhoh");
}
```

▼ *Speaker Notes*

- Note that the `P: Fn(u8, &str) -> bool` bound on the first `Filter` impl block isn't strictly necessary, but it helps with type inference when calling `new`. Demonstrate removing it and showing how the compiler now needs type annotations for the closure passed to `new`.

Standard Library Traits

This segment should take about 1 hour. It contains:

Slide	Duration
Comparisons	5 minutes
Operators	5 minutes
From and Into	5 minutes
Casting	5 minutes
Read and Write	5 minutes
Default, struct update syntax	5 minutes
Exercise: ROT13	30 minutes

▼ *Speaker Notes*

As with the standard-library types, spend time reviewing the documentation for each trait.

This section is long. Take a break midway through.

Comparisons

These traits support comparisons between values. All traits can be derived for types containing fields that implement these traits.

`PartialEq` and `Eq`

`PartialEq` is a partial equivalence relation, with required method `eq` and provided method `ne`. The `==` and `!=` operators will call these methods.

```
struct Key {
    id: u32,
    metadata: Option<String>,
}
impl PartialEq for Key {
    fn eq(&self, other: &Self) -> bool {
        self.id == other.id
    }
}
```

`Eq` is a full equivalence relation (reflexive, symmetric, and transitive) and implies `PartialEq`. Functions that require full equivalence will use `Eq` as a trait bound.

`PartialOrd` and `Ord`

`PartialOrd` defines a partial ordering, with a `partial_cmp` method. It is used to implement the `<`, `<=`, `>=`, and `>` operators.

```
use std::cmp::Ordering;
#[derive(Eq, PartialEq)]
struct Citation {
    author: String,
    year: u32,
}
impl PartialOrd for Citation {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        match self.author.partial_cmp(&other.author) {
            Some(Ordering::Equal) => self.year.partial_cmp(&other.year),
            author_ord => author_ord,
        }
    }
}
```

`Ord` is a total ordering, with `cmp` returning `Ordering`.

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `PartialEq` can be implemented between different types, but `Eq` cannot, because it is reflexive:

```rust
struct Key {
    id: u32,
    metadata: Option<String>,
}
impl PartialEq<u32> for Key {
    fn eq(&self, other: &u32) -> bool {
        self.id == *other
    }
}
```

- In practice, it's common to derive these traits, but uncommon to implement them.
- When comparing references in Rust, it will compare the value of the things pointed to, it will NOT compare the references themselves. That means that references to two different things can compare as equal if the values pointed to are the same:

```rust
fn main() {
    let a = "Hello";
    let b = String::from("Hello");
    assert_eq!(a, b);
}
```

Operators

Operator overloading is implemented via traits in `std::ops`:

```
#[derive(Debug, Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

impl std::ops::Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self {
        Self { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Point { x: 10, y: 20 };
    let p2 = Point { x: 100, y: 200 };
    println!("{p1:?} + {p2:?} = {:?}", p1 + p2);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Discussion points:

- You could implement `Add` for `&Point`. In which situations is that useful?
  - Answer: `Add:add` consumes `self`. If type `T` for which you are overloading the operator is not `Copy`, you should consider overloading the operator for `&T` as well. This avoids unnecessary cloning on the call site.
- Why is `Output` an associated type? Could it be made a type parameter of the method?
  - Short answer: Function type parameters are controlled by the caller, but associated types (like `Output`) are controlled by the implementer of a trait.
- You could implement `Add` for two different types, e.g. `impl Add<(i32, i32)> for Point` would add a tuple to a `Point`.

The `Not` trait (`!` operator) is notable because it does not "boolify" like the same operator in C-family languages; instead, for integer types it negates each bit of the number, which arithmetically is equivalent to subtracting it from -1: `!5 == -6`.

`From` and `Into`

Types implement `From` and `Into` to facilitate type conversions. Unlike `as`, these traits correspond to lossless, infallible conversions.

```
fn main() {
    let s = String::from("hello");
    let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
    let one = i16::from(true);
    let bigger = i32::from(123_i16);
    println!("{s}, {addr}, {one}, {bigger}");
}
```

`Into` is automatically implemented when `From` is implemented:

```
fn main() {
    let s: String = "hello".into();
    let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
    let one: i16 = true.into();
    let bigger: i32 = 123_i16.into();
    println!("{s}, {addr}, {one}, {bigger}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- That's why it is common to only implement `From`, as your type will get `Into` implementation too.
- When declaring a function argument input type like "anything that can be converted into a `String`", the rule is opposite, you should use `Into`. Your function will accept types that implement `From` and those that *only* implement `Into`.

Casting

Rust has no *implicit* type conversions, but does support explicit casts with `as`. These generally follow C semantics where those are defined.

```
fn main() {
    let value: i64 = 1000;
    println!("as u16: {}", value as u16);
    println!("as i16: {}", value as i16);
    println!("as u8: {}", value as u8);
}
```

The results of `as` are *always* defined in Rust and consistent across platforms. This might not match your intuition for changing sign or casting to a smaller type – check the docs, and comment for clarity.

Casting with `as` is a relatively sharp tool that is easy to use incorrectly, and can be a source of subtle bugs as future maintenance work changes the types that are used or the ranges of values in types. Casts are best used only when the intent is to indicate unconditional truncation (e.g. selecting the bottom 32 bits of a `u64` with `as u32`, regardless of what was in the high bits).

For infallible casts (e.g. `u32` to `u64`), prefer using `From` or `Into` over `as` to confirm that the cast is in fact infallible. For fallible casts, `TryFrom` and `TryInto` are available when you want to handle casts that fit differently from those that don't.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Consider taking a break after this slide.

`as` is similar to a C++ static cast. Use of `as` in cases where data might be lost is generally discouraged, or at least deserves an explanatory comment.

This is common in casting integers to `usize` for use as an index.

Read and Write

Using `Read` and `BufRead`, you can abstract over `u8` sources:

```
use std::io::{BufRead, BufReader, Read, Result};

fn count_lines<R: Read>(reader: R) -> usize {
    let buf_reader = BufReader::new(reader);
    buf_reader.lines().count()
}

fn main() -> Result<()> {
    let slice: &[u8] = b"foo\nbar\nbaz\n";
    println!("lines in slice: {}", count_lines(slice));

    let file = std::fs::File::open(std::env::current_exe()?)?;
    println!("lines in file: {}", count_lines(file));
    Ok(())
}
```

Similarly, `Write` lets you abstract over `u8` sinks:

```
use std::io::{Result, Write};

fn log<W: Write>(writer: &mut W, msg: &str) -> Result<()> {
    writer.write_all(msg.as_bytes())?;
    writer.write_all("\n".as_bytes())
}

fn main() -> Result<()> {
    let mut buffer = Vec::new();
    log(&mut buffer, "Hello")?;
    log(&mut buffer, "World")?;
    println!("Logged: {buffer:?}");
    Ok(())
}
```

The `Default` Trait

The `Default` trait produces a default value for a type.

```
#[derive(Debug, Default)]
struct Derived {
    x: u32,
    y: String,
    z: Implemented,
}

#[derive(Debug)]
struct Implemented(String);

impl Default for Implemented {
    fn default() -> Self {
        Self("John Smith".into())
    }
}

fn main() {
    let default_struct = Derived::default();
    dbg!(default_struct);

    let almost_default_struct =
        Derived { y: "Y is set!".into(), ..Derived::default() };
    dbg!(almost_default_struct);

    let nothing: Option<Derived> = None;
    dbg!(nothing.unwrap_or_default());
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- It can be implemented directly or it can be derived via `#[derive(Default)]`.
- A derived implementation will produce a value where all fields are set to their default values.
  - This means all types in the struct must implement `Default` too.
- Standard Rust types often implement `Default` with reasonable values (e.g. `0`, `""`, etc).
- The partial struct initialization works nicely with default.
- The Rust standard library is aware that types can implement `Default` and provides convenience methods that use it.
- The `..` syntax is called struct update syntax.

Exercise: ROT13

In this example, you will implement the classic "ROT13" cipher. Copy this code to the playground, and implement the missing bits. Only rotate ASCII alphabetic characters, to ensure the result is still valid UTF-8.

```
use std::io::Read;

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
}

// Implement the `Read` trait for `RotDecoder`.

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn joke() {
        let mut rot =
            RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "To get to the other side!");
    }

    #[test]
    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13 };
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

What happens if you chain two `RotDecoder` instances together, each rotating by 13 characters?

Solution

```rust
use std::io::Read;

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
}

impl<R: Read> Read for RotDecoder<R> {
    fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize> {
        let size = self.input.read(buf)?;
        for b in &mut buf[..size] {
            if b.is_ascii_alphabetic() {
                let base = if b.is_ascii_uppercase() { 'A' } else { 'a' } as u8;
                *b = (*b - base + self.rot) % 26 + base;
            }
        }
        Ok(size)
    }
}

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn joke() {
        let mut rot =
            RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "To get to the other side!");
    }

    #[test]
    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13 };
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

Welcome to Day 3

Today, we will cover:

- Memory management, lifetimes, and the borrow checker: how Rust ensures memory safety.
- Smart pointers: standard library pointer types.

Schedule

Including 10 minute breaks, this session should take about 2 hours and 20 minutes. It contains:

Segment	Duration
Welcome	3 minutes
Memory Management	1 hour
Smart Pointers	55 minutes

Memory Management

This segment should take about 1 hour. It contains:

Slide	Duration
Review of Program Memory	5 minutes
Approaches to Memory Management	10 minutes
Ownership	5 minutes
Move Semantics	5 minutes
Clone	2 minutes
Copy Types	5 minutes
Drop	10 minutes
Exercise: Builder Type	20 minutes

Review of Program Memory

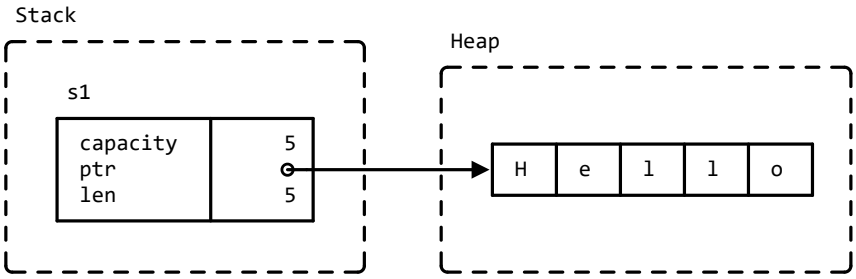
Programs allocate memory in two ways:

- Stack: Continuous area of memory for local variables.
  - Values have fixed sizes known at compile time.
  - Extremely fast: just move a stack pointer.
  - Easy to manage: follows function calls.
  - Great memory locality.
- Heap: Storage of values outside of function calls.
  - Values have dynamic sizes determined at runtime.
  - Slightly slower than the stack: some book-keeping needed.
  - No guarantee of memory locality.

Example

Creating a `String` puts fixed-sized metadata on the stack and dynamically sized data, the actual string, on the heap:

```
fn main() {
    let s1 = String::from("Hello");
}
```



▼ *Speaker Notes*

This slide should take about 5 minutes.

- Mention that a `String` is backed by a `Vec`, so it has a capacity and length and can grow if mutable via reallocation on the heap.
- If students ask about it, you can mention that the underlying memory is heap allocated using the System Allocator and custom allocators can be implemented using the Allocator API

More to Explore

We can inspect the memory layout with `unsafe` Rust. However, you should point out that this is rightfully unsafe!

```rust
fn main() {
    let mut s1 = String::from("Hello");
    s1.push(' ');
    s1.push_str("world");
    // DON'T DO THIS AT HOME! For educational purposes only.
    // String provides no guarantees about its layout, so this could lead to
    // undefined behavior.
    unsafe {
        let (capacity, ptr, len): (usize, usize, usize) = std::mem::transmute(s1);
        println!("capacity = {capacity}, ptr = {ptr:#x}, len = {len}");
    }
}
```

Approaches to Memory Management

Traditionally, languages have fallen into two broad categories:

- Full control via manual memory management: C, C++, Pascal, ...
  - Programmer decides when to allocate or free heap memory.
  - Programmer must determine whether a pointer still points to valid memory.
  - Studies show, programmers make mistakes.
- Full safety via automatic memory management at runtime: Java, Python, Go, Haskell, ...
  - A runtime system ensures that memory is not freed until it can no longer be referenced.
  - Typically implemented with reference counting or garbage collection.

Rust offers a new mix:

> Full control *and* safety via compile time enforcement of correct memory management.

It does this with an explicit ownership concept.

▼ *Speaker Notes*

This slide should take about 10 minutes.

This slide is intended to help students coming from other languages to put Rust in context.

- C must manage heap manually with `malloc` and `free`. Common errors include forgetting to call `free`, calling it multiple times for the same pointer, or dereferencing a pointer after the memory it points to has been freed.
- C++ has tools like smart pointers (`unique_ptr`, `shared_ptr`) that take advantage of language guarantees about calling destructors to ensure memory is freed when a function returns. It is still quite easy to mis-use these tools and create similar bugs to C.
- Java, Go, and Python rely on the garbage collector to identify memory that is no longer reachable and discard it. This guarantees that any pointer can be dereferenced, eliminating use-after-free and other classes of bugs. But, GC has a runtime cost and is difficult to tune properly.

Rust's ownership and borrowing model can, in many cases, get the performance of C, with alloc and free operations precisely where they are required – zero cost. It also provides tools similar to C++'s smart pointers. When required, other options such as reference counting are available, and there are even crates available to support runtime garbage collection (not covered in this class).

Ownership

All variable bindings have a *scope* where they are valid and it is an error to use a variable outside its scope:

```
struct Point(i32, i32);

fn main() {
    {
        let p = Point(3, 4);
        dbg!(p.0);
    }
    dbg!(p.1);
}
```

We say that the variable *owns* the value. Every Rust value has precisely one owner at all times.

At the end of the scope, the variable is *dropped* and the data is freed. A destructor can run here to free up resources.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Students familiar with garbage-collection implementations will know that a garbage collector starts with a set of “roots” to find all reachable memory. Rust’s “single owner” principle is a similar idea.

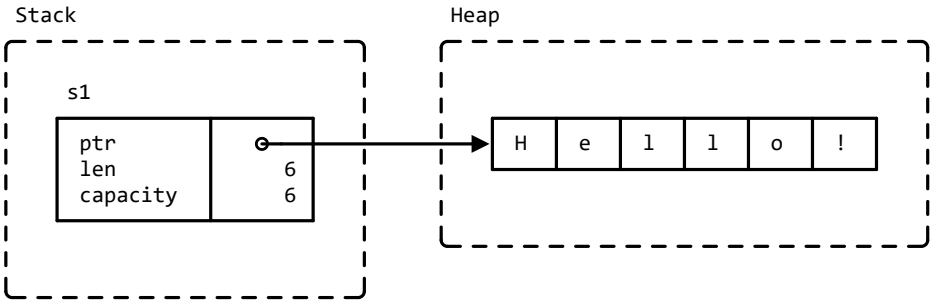
Move Semantics

An assignment will transfer *ownership* between variables:

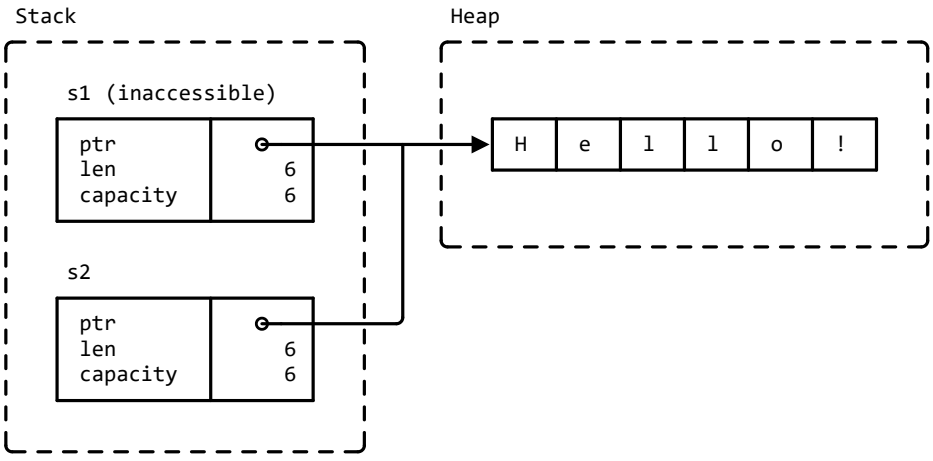
```
fn main() {
    let s1 = String::from("Hello!");
    let s2 = s1;
    dbg!(s2);
    // dbg!(s1);
}
```

- The assignment of `s1` to `s2` transfers ownership.
- When `s1` goes out of scope, nothing happens: it does not own anything.
- When `s2` goes out of scope, the string data is freed.

Before move to `s2`:



After move to `s2`:



When you pass a value to a function, the value is assigned to the function parameter. This transfers ownership:

```
fn say_hello(name: String) {
    println!("Hello {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name);
    // say_hello(name);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Mention that this is the opposite of the defaults in C++, which copies by value unless you use `std::move` (and the move constructor is defined!).
- It is only the ownership that moves. Whether any machine code is generated to manipulate the data itself is a matter of optimization, and such copies are aggressively optimized away.
- Simple values (such as integers) can be marked `Copy` (see later slides).
- In Rust, clones are explicit (by using `clone`).

In the `say_hello` example:

- With the first call to `say_hello`, `main` gives up ownership of `name`. Afterwards, `name` cannot be used anymore within `main`.
- The heap memory allocated for `name` will be freed at the end of the `say_hello` function.
- `main` can retain ownership if it passes `name` as a reference (`&name`) and if `say_hello` accepts a reference as a parameter.
- Alternatively, `main` can pass a clone of `name` in the first call (`name.clone()`).
- Rust makes it harder than C++ to inadvertently create copies by making move semantics the default, and by forcing programmers to make clones explicit.

More to Explore

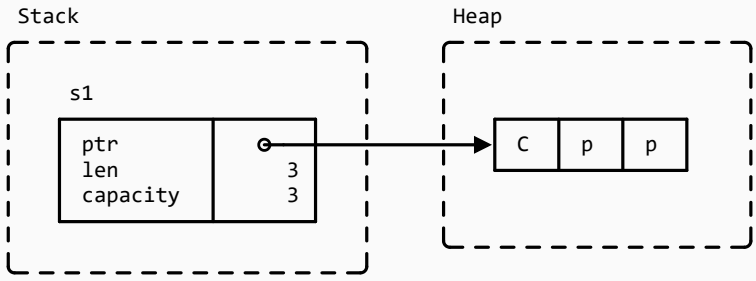
Defensive Copies in Modern C++

Modern C++ solves this differently:

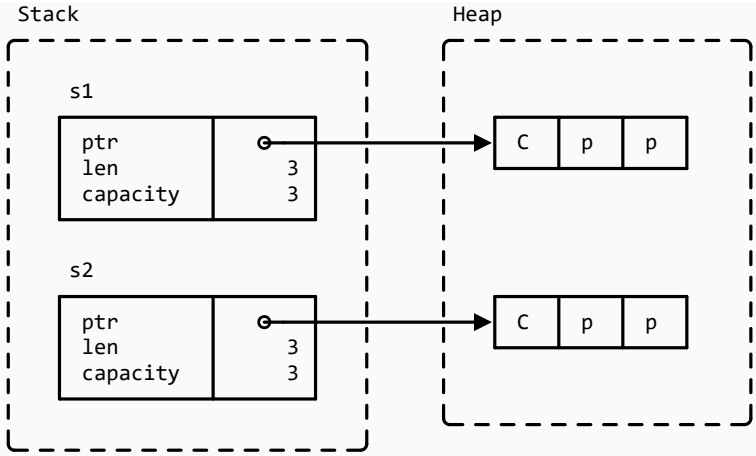
```
std::string s1 = "Cpp";
std::string s2 = s1;  // Duplicate the data in s1.
```

- The heap data from `s1` is duplicated and `s2` gets its own independent copy.
- When `s1` and `s2` go out of scope, they each free their own memory.

Before copy-assignment:



After copy-assignment:



Key points:

- C++ has made a slightly different choice than Rust. Because `=` copies data, the string data has to be cloned. Otherwise we would get a double-free when either string goes out of scope.
- C++ also has `std::move`, which is used to indicate when a value may be moved from. If the example had been `s2 = std::move(s1)`, no heap allocation would take place. After the move, `s1` would be in a valid but unspecified state. Unlike Rust, the programmer is allowed to keep using `s1`.
- Unlike Rust, `=` in C++ can run arbitrary code as determined by the type which is being copied or moved.

Clone

Sometimes you *want* to make a copy of a value. The `Clone` trait accomplishes this.

```
fn say_hello(name: String) {
    println!("Hello {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name.clone());
    say_hello(name);
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

- The idea of `Clone` is to make it easy to spot where heap allocations are occurring. Look for `.clone()` and a few others like `vec!` or `Box::new`.
- It's common to "clone your way out" of problems with the borrow checker, and return later to try to optimize those clones away.
- `clone` generally performs a deep copy of the value, meaning that if you e.g. clone an array, all of the elements of the array are cloned as well.
- The behavior for `clone` is user-defined, so it can perform custom cloning logic if needed.

Copy Types

While move semantics are the default, certain types are copied by default:

```
fn main() {
    let x = 42;
    let y = x;
    dbg!(x); // would not be accessible if not Copy
    dbg!(y);
}
```

These types implement the `Copy` trait.

You can opt-in your own types to use copy semantics:

```
#[derive(Copy, Clone, Debug)]
struct Point(i32, i32);

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;
    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

- After the assignment, both `p1` and `p2` own their own data.
- We can also use `p1.clone()` to explicitly copy the data.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Copying and cloning are not the same thing:

- Copying refers to bitwise copies of memory regions and does not work on arbitrary objects.
- Copying does not allow for custom logic (unlike copy constructors in C++).
- Cloning is a more general operation and also allows for custom behavior by implementing the `Clone` trait.
- Copying does not work on types that implement the `Drop` trait.

In the above example, try the following:

- Add a `String` field to `struct Point`. It will not compile because `String` is not a `Copy` type.
- Remove `Copy` from the `derive` attribute. The compiler error is now in the `println!` for `p1`.
- Show that it works if you clone `p1` instead.

More to Explore

- Shared references are `Copy` / `Clone`, mutable references are not. This is because Rust requires that mutable references be exclusive, so while it's valid to make a copy of a shared reference, creating a copy of a mutable reference would violate Rust's borrowing rules.

The Drop Trait

Values which implement `Drop` can specify code to run when they go out of scope:

```
struct Droppable {
    name: &'static str,
}

impl Drop for Droppable {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("Exiting innermost block");
        }
        println!("Exiting next block");
    }
    drop(a);
    println!("Exiting main");
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

- Note that `std::mem::drop` is not the same as `std::ops::Drop::drop`.
- Values are automatically dropped when they go out of scope.
- When a value is dropped, if it implements `std::ops::Drop` then its `Drop::drop` implementation will be called.
- All its fields will then be dropped too, whether or not it implements `Drop`.
- `std::mem::drop` is just an empty function that takes any value. The significance is that it takes ownership of the value, so at the end of its scope it gets dropped. This makes it a convenient way to explicitly drop values earlier than they would otherwise go out of scope.
    - This can be useful for objects that do some work on `drop`: releasing locks, closing files, etc.

Discussion points:

- Why doesn't `Drop::drop` take `self`?
    - Short-answer: If it did, `std::mem::drop` would be called at the end of the block, resulting in another call to `Drop::drop`, and a stack overflow!
- Try replacing `drop(a)` with `a.drop()`.

Exercise: Builder Type

In this example, we will implement a complex data type that owns all of its data. We will use the “builder pattern” to support building a new value piece-by-piece, using convenience functions.

Fill in the missing pieces.