Using the Borrow checker to enforce Invariants

The borrow checker, while added to enforce memory ownership, can model other problems and prevent API misuse.

```
/// Doors can be open or closed, and you need the right key to lock or unlc
/// one. Modelled with a Shared key and Owned door.
pub struct DoorKey {
    pub key_shape: u32,
}
pub struct LockedDoor {
    lock_shape: u32,
}
pub struct OpenDoor {
    lock_shape: u32,
}

fn open_door(key: &DoorKey, door: LockedDoor) -> Result<OpenDoor, LockedDoc
    if door.lock_shape == key.key_shape {
        Ok(OpenDoor { lock_shape: door.lock_shape })
    } else {
        Err(door)
    }
}

fn close_door(key: &DoorKey, door: OpenDoor) -> Result<LockedDoor, OpenDoor
    if door.lock_shape == key.key_shape {
        Ok(LockedDoor { lock_shape: door.lock_shape })
    } else {
        Err(door)
    }
}

fn main() {
    let key = DoorKey { key_shape: 7 };
    let closed_door = LockedDoor { lock_shape: 7 };
    let opened_door = open_door(&key, closed_door);
    if let Ok(opened_door) = opened_door {
        println!("Opened the door with key shape '{}'", key.key_shape);
    } else {
        eprintln!(
            "Door wasn't opened! Your key only opens locks with shape '{}'"
            key.key_shape
        );
    }
}
```

▼ *Speaker Notes*

- We've seen the borrow checker prevent memory safety bugs (use-after-free, data races).
- We've also used types to shape and restrict APIs already using the Typestate pattern.
- Language features are often introduced for a specific purpose.

  Over time, users may develop ways of using a feature in ways that were not predicted when they were introduced.

  Java 5 introduced Generics in 2004 with the main stated purpose of enabling type-safe collections.

  Adoption was slow at first, but some new projects began designing their APIs around generics from the beginning.

  Since then, users and developers of the language expanded the use of generics to other areas of type-safe API design:

  - Class information can be held onto via Java's `Class<T>` or Guava's `TypeToken<T>`.
  - The Builder pattern can be implemented using Recursive Generics. We aim to do something similar here: Even though the borrow checker was introduced to prevent use-after-free and data races, we treat it as just another API design tool.

  It can be used to model program properties that have nothing to do with preventing memory safety bugs.
- To use the borrow checker as a problem solving tool, we will need to "forget" that the original purpose of it is to prevent mutable aliasing in the context of preventing use-after-frees and data races.

  We should imagine working within situations where the rules are the same but the meaning is slightly different.
- This example uses ownership and borrowing are used to model the state of a physical door.

  `open_door` **consumes** a `LockedDoor` and returns a new `OpenDoor`. The old `LockedDoor` value is no longer available.

  If the wrong key is used, the door is left locked. It is returned as an `Err` case of the `Result`.

  It is a compile-time error to try and use a door that has already been opened.

- The rules of the borrow checker exist to prevent memory safety bugs, but the underlying logical system does not “know” what memory is.

  All the borrow checker does is enforce a specific set of rules of how users can order operations.

  This is just one case of piggy-backing onto the rules of the borrow checker to design APIs to be harder or impossible to misuse.

Lifetimes and Borrows: the Abstract Rules

```rust
// An internal data type to have something to hold onto.
pub struct Internal;
// The "outer" data.
pub struct Data(Internal);

fn shared_use(value: &Data) -> &Internal {
    &value.0
}
fn exclusive_use(value: &mut Data) -> &mut Internal {
    &mut value.0
}
fn deny_future_use(value: Data) {}

fn demo_exclusive() {
    let mut value = Data(Internal);
    let shared = shared_use(&value);
    // let exclusive = exclusive_use(&mut value); // ❌🔨
    let shared_again = &shared;
}

fn demo_denied() {
    let value = Data(Internal);
    deny_future_use(value);
    // let shared = shared_use(&value); // ❌🔨
}

fn main() {}
```

▼ *Speaker Notes*

- This example re-frames the borrow checker rules away from references and towards semantic meaning in non-memory-safety settings.

  Nothing is being mutated, nothing is being sent across threads.

- In rust's borrow checker we have access to three different ways of "taking" a value:

  - Owned value `T`. Value is dropped when the scope ends, unless it is not returned to another scope.
  - Shared Reference `&T`. Allows aliasing but prevents mutable access while shared references are in use.
  - Mutable Reference `&mut T`. Only one of these is allowed to exist for a value at any one point, but can be used to create shared references.

`demo_exclusive` : Because the `shared` value is still aliased after the `exclusive` reference is taken.

`demo_denied` : Because `value` is consumed the line before the `shared_again_again` reference is taken from `&value` .

- Remember that every `&T` and `&mut T` has a lifetime, just one the user doesn't have to annotate or think about most of the time.

  We rarely specify lifetimes because the Rust compiler allows us to *elide* them in most cases. See: Lifetime Elision

Single-use values

Sometimes we want values that *can only be used once*. One critical example of this is in cryptography: A "Nonce."

```
pub struct Key(/* specifics omitted */);
/// A single-use number suitable for cryptographic purposes.
pub struct Nonce(u32);
/// A cryptographically sound random generator function.
pub fn new_nonce() -> Nonce {
    Nonce(4) // chosen by a fair dice roll, https://xkcd.com/221/
}
/// Consume a nonce, but not the key or the data.
pub fn encrypt(nonce: Nonce, key: &Key, data: &[u8]) {}

fn main() {
    let nonce = new_nonce();
    let data_1: [u8; 4] = [1, 2, 3, 4];
    let data_2: [u8; 4] = [4, 3, 2, 1];
    let key = Key(/* specifics omitted */);

    // The key and data can be re-used, copied, etc. but the nonce cannot.
    encrypt(nonce, &key, &data_1);
    // encrypt(nonce, &key, &data_2); // 🛠❌
}
```

▼ *Speaker Notes*

- Problem: How can we guarantee a value is used only once?
- Motivation: A nonce is a piece of random, unique data used in cryptographic protocols to prevent replay attacks.

  Background: In practice people have ended up accidentally re-using nonces. Most commonly, this causes the cryptographic protocol to completely break down and stop fulfilling its function.

  Depending on the specifics of nonce reuse and cryptography at hand, private keys can also become computable by attackers.
- Rust has an obvious tool for achieving the invariant "Once you use this, you can't use it again": passing a value as an *owned argument*.
- Highlight: the `encrypt` function takes `nonce` by value (an owned argument), but `key` and `data` by reference.
- The technique for single-use values is as follows:

    - Don't implement `Clone` / `Copy` traits or equivalent methods, so a user can't duplicate data we want to keep unique.
    - Make the interior type opaque (like with the newtype pattern), so the user cannot modify an existing value on their own.

- Ask: What are we missing from the newtype pattern in the slide's code?

  Expect: Module boundary.

  Demonstrate: Without a module boundary a user can construct a nonce on their own.

  Fix: Put `Key`, `Nonce`, and `new_nonce` behind a module.

More to Explore

- Cryptography Nuance: A nonce might still be used twice if it was created through pseudo-random process with no actual randomness. That can't be prevented through this method. This API design prevents one nonce duplication, but not all logic bugs.

Mutually Exclusive References / “Aliasing XOR Mutability”

We can use the mutual exclusion of `&T` and `&mut T` references to prevent data from being used before it is ready.

```
pub struct QueryResult;
pub struct DatabaseConnection {/* fields omitted */}

impl DatabaseConnection {
    pub fn new() -> Self {
        Self {}
    }
    pub fn results(&self) -> &[QueryResult] {
        &[] // fake results
    }
}

pub struct Transaction<'a> {
    connection: &'a mut DatabaseConnection,
}

impl<'a> Transaction<'a> {
    pub fn new(connection: &'a mut DatabaseConnection) -> Self {
        Self { connection }
    }
    pub fn query(&mut self, _query: &str) {
        // Send the query over, but don't wait for results.
    }
    pub fn commit(self) {
        // Finish executing the transaction and retrieve the results.
    }
}

fn main() {
    let mut db = DatabaseConnection::new();

    // The transaction `tx` mutably borrows `db`.
    let mut tx = Transaction::new(&mut db);
    tx.query("SELECT * FROM users");

    // This won't compile because `db` is already mutably borrowed by `tx`.
    // let results = db.results(); // ❌🔨

    // The borrow of `db` ends when `tx` is consumed by `commit()`.
    tx.commit();

    // Now it is possible to borrow `db` again.
    let results = db.results();
}
```

- Motivation: In this database API queries are kicked off for asynchronous execution and the results are only available once the whole transaction is finished.

  A user might think that queries are executed immediately, and try to read results before they are made available. This API misuse could make the app read incomplete or incorrect data.

  While an obvious misunderstanding, situations such as this can happen in practice.

  Ask: Has anyone misunderstood an API by not reading the docs for proper use?

  Expect: Examples of early-career or in-university mistakes and misunderstandings.

  As an API grows in size and user base, a smaller percentage of users has deep knowledge of the system the API represents.

- This example shows how we can use Aliasing XOR Mutability to prevent this kind of misuse.

- The code might read results before they are ready if the programmer assumes that the queries execute immediately rather than kicked off for asynchronous execution.

- The constructor for the `Transaction` type takes a mutable reference to the database connection, and stores it in the returned `Transaction` value.

  The explicit lifetime here doesn't have to be intimidating, it just means "`Transaction` is outlived by the `DatabaseConnection` that was passed to it" in this case.

  The reference is mutable to completely lock out the `DatabaseConnection` from other usage, such as starting further transactions or reading the results.

- While a `Transaction` exists, we can't touch the `DatabaseConnection` variable that was created from it.

  Demonstrate: uncomment the `db.results()` line. Doing so will result in a compile error, as `db` is already mutably borrowed.

- Note: The query results not being public and placed behind a getter function lets us enforce the invariant "users can only look at query results if there is no active transactions."

  If the query results were placed in a public struct field, this invariant could be violated.

PhantomData 1/4: De-duplicating Same Data & Semantics

The newtype pattern can sometimes come up against the DRY principle, how do we solve this?

```
pub struct UserId(u64);
impl ChatUser for UserId { /* ... */ }

pub struct PatronId(u64);
impl ChatUser for PatronId { /* ... */ }

pub struct ModeratorId(u64);
impl ChatUser for ModeratorId { /* ... */ }
impl ChatModerator for ModeratorId { /* ... */ }

pub struct AdminId(u64);
impl ChatUser for AdminId { /* ... */ }
impl ChatModerator for AdminId { /* ... */ }
impl ChatAdmin for AdminId { /* ... */ }

// And so on ...
fn main() {}
```

▼ *Speaker Notes*

- Problem: We want to use the newtype pattern to differentiate permissions, but we're having to implement the same traits over and over again for the same data.
- Ask: Assume the details of each implementation here are the same between types, what are ways we can avoid repeating ourselves?

  Expect:

  - Make this an enum, not distinct data types.
  - Bundle the user ID with permission tokens like `struct Admin(u64, UserPermission, ModeratorPermission, AdminPermission);`
  - Adding a type parameter which encodes permissions.
  - Mentioning `PhantomData` ahead of schedule (it's in the title).

PhantomData 2/4: Type-level tagging

Let's solve the problem from the previous slide by adding a type parameter.

```
// use std::marker::PhantomData;

pub struct ChatId<T> { id: u64, tag: T }

pub struct UserTag;
pub struct AdminTag;

pub trait ChatUser {/* ... */}
pub trait ChatAdmin {/* ... */}

impl ChatUser for UserTag {/* ... */}
impl ChatUser for AdminTag {/* ... */} // Admins are users
impl ChatAdmin for AdminTag {/* ... */}

// impl <T> Debug for UserTag<T> {/* ... */}
// impl <T> PartialEq for UserTag<T> {/* ... */}
// impl <T> Eq for UserTag<T> {/* ... */}
// And so on ...

impl <T: ChatUser> ChatId<T> {/* All functionality for users and above */}
impl <T: ChatAdmin> ChatId<T> {/* All functionality for only admins */}

fn main() {}
```

▼ *Speaker Notes*

- Here we're using a type parameter and gating permissions behind "tag" types that implement different permission traits.

  Tag types, or marker types, are zero-sized types that have some semantic meaning to users and API designers.

- Ask: What issues does having it be an actual instance of that type pose?

  Answer: If it's not a zero-sized type (like `()` or `struct MyTag;`), then we're allocating more memory than we need to when all we care for is type information that is only relevant at compile-time.

- Demonstrate: remove the `tag` value entirely, then compile!

  This won't compile, as there's an unused (phantom) type parameter.

  This is where `PhantomData` comes in!

```
pub struct ChatId<T> {
    id: u64,
    tag: PhantomData<T>,
}
```

- `PhantomData<T>` is a zero-sized type with a type parameter. We can construct values of it like other ZSTs with `let phantom: PhantomData<UserTag> = PhantomData;` or with the `PhantomData::default()` implementation.

  Demonstrate: implement `From<u64>` for `ChatId<T>`, emphasizing the construction of `PhantomData`

```
impl<T> From<u64> for ChatId<T> {
    fn from(value: u64) -> Self {
        ChatId {
            id: value,
            // Or `PhantomData::default()`
            tag: PhantomData,
        }
    }
}
```

- `PhantomData` can be used as part of the Typestate pattern to have data with the same structure but different methods, e.g., have `TaggedData<Start>` implement methods or trait implementations that `TaggedData<End>` doesn't.

PhantomData 3/4: Lifetimes for External Resources

The invariants of external resources often match what we can do with lifetime rules.

```
// use std::marker::PhantomData;

/// Direct FFI to a database library in C.
/// We got this API as is, we have no influence over it.
mod ffi {
    pub type DatabaseHandle = u8; // maximum 255 databases open at the same

    fn database_open(name: *const std::os::raw::c_char) -> DatabaseHandle {
        unimplemented!()
    }
    // ... etc.
}

struct DatabaseConnection(ffi::DatabaseHandle);
struct Transaction<'a>(&'a mut DatabaseConnection);

impl DatabaseConnection {
    fn new_transaction(&mut self) -> Transaction<'_> {
        Transaction(self)
    }
}

fn main() {}
```

▼ *Speaker Notes*

- Remember the transaction API from the Aliasing XOR Mutability example.

  We held onto a mutable reference to the database connection within the transaction type to lock out the database while a transaction is active.

  In this example, we want to implement a `Transaction` API on top of an external, non-Rust API.

  We start by defining a `Transaction` type that holds onto `&mut DatabaseConnection`.

- Ask: What are the limits of this implementation? Assume the `u8` is accurate implementation-wise and enough information for us to use the external API.

  Expect:

  - Indirection takes up 7 bytes more than we need to on a 64-bit platform, as well

- Problem: We want the transaction to borrow the database connection that created it, but we don't want the `Transaction` object to store a real reference.
- Ask: What happens when we remove the mutable reference in `Transaction` while keeping the lifetime parameter?

  Expect: Unused lifetime parameter!
- Like with the type tagging from the previous slides, we can bring in `PhantomData` to capture this unused lifetime parameter for us.

  The difference is that we will need to use the lifetime alongside another type, but that other type does not matter too much.
- Demonstrate: change `Transaction` to the following:

```
pub struct Transaction<'a> {
    connection: DatabaseConnection,
    _phantom: PhantomData<&mut 'a ()>,
}
```

  Update the `DatabaseConnection::new_transaction()` method:

```
fn new_transaction<'a>(&'a mut self) -> Transaction<'a> {
    Transaction { connection: DatabaseConnection(self.0), _phantom:
PhantomData }
}
```

  This gives an owned database connection that is tied to the `DatabaseConnection` that created it, but with less runtime memory footprint that the store-a-reference version did.

  Because `PhantomData` is a zero-sized type (like `()` or `struct MyZeroSizedType;`), the size of `Transaction` is now the same as `u8`.

  The implementation that held onto a reference instead was as large as a `usize`.

More to Explore

- This way of encoding relationships between types and values is very powerful when combined with unsafe, as the ways one can manipulate lifetimes becomes almost arbitrary. This is also dangerous, but when combined with tools like external

- The GhostCell (2021) paper and its relevant implementation show this kind of work off. While the borrow checker is restrictive, there are still ways to use escape hatches and then *show that the ways you used those escape hatches are consistent and safe.*

PhantomData 4/4: OwnedFd & BorrowedFd

`BorrowedFd` is a prime example of `PhantomData` in action.

```rust
use std::marker::PhantomData;
use std::os::raw::c_int;

mod libc_ffi {
    use std::os::raw::{c_char, c_int};
    pub unsafe fn open(path: *const c_char, oflag: c_int) -> c_int {
        3
    }
    pub unsafe fn close(fd: c_int) {}
}

struct OwnedFd {
    fd: c_int,
}

impl OwnedFd {
    fn try_from_fd(fd: c_int) -> Option<Self> {
        if fd < 0 {
            return None;
        }
        Some(OwnedFd { fd })
    }

    fn as_fd<'a>(&'a self) -> BorrowedFd<'a> {
        BorrowedFd { fd: self.fd, _phantom: PhantomData }
    }
}

impl Drop for OwnedFd {
    fn drop(&mut self) {
        unsafe { libc_ffi::close(self.fd) };
    }
}

struct BorrowedFd<'a> {
    fd: c_int,
    _phantom: PhantomData<&'a ()>,
}

fn main() {
    // Create a file with a raw syscall with write-only and create permissi
    let fd = unsafe { libc_ffi::open(c"c_str.txt".as_ptr(), 065) };
    // Pass the ownership of an integer file descriptor to an `OwnedFd`.
    // `OwnedFd::drop()` closes the file descriptor.
    let owned_fd =
        OwnedFd::try_from_fd(fd).expect("Could not open file with syscall!"

    // Create a `BorrowedFd` from an `OwnedFd`.
    // `BorrowedFd::drop()` does not close the file because it doesn't own
    let borrowed_fd: BorrowedFd<'_> = owned_fd.as_fd();
    // std::mem::drop(owned_fd); // ❌🔨
    std::mem::drop(borrowed_fd);
    let second_borrowed = owned_fd.as_fd();
    // owned_fd will be dropped here, and the file will be closed.
}
```

- A file descriptor represents a specific process's access to a file.

  Reminder: Device and OS-specific features are exposed as if they were files on unix-style systems.

- `OwnedFd` is an owned wrapper type for a file descriptor. It *owns* the file descriptor, and closes it when dropped.

  Note: We have our own implementation of it here, draw attention to the explicit `Drop` implementation.

  `BorrowedFd` is its borrowed counterpart, it does not need to close the file when it is dropped.

  Note: We have not explicitly implemented `Drop` for `BorrowedFd`.

- `BorrowedFd` uses a lifetime captured with a `PhantomData` to enforce the invariant "if this file descriptor exists, the OS file descriptor is still open even though it is not responsible for closing that file descriptor."

  The lifetime parameter of `BorrowedFd` demands that there exists another value in your program that lasts as long as that specific `BorrowedFd` or outlives it (in this case an `OwnedFd`).

  Demonstrate: Uncomment the `std::mem::drop(owned_fd)` line and try to compile to show that `borrowed_fd` relies on the lifetime of `owned_fd`.

  This has been encoded by the API designers to mean *that other value is what keeps the access to the file open*.

  Because Rust's borrow checker enforces this relationship where one value must last at least as long as another, users of this API do not need to worry about handling this correct file descriptor aliasing and closing logic themselves.

Token Types

Types with private constructors can be used to act as proof of invariants.

```
pub mod token {
    // A public type with private fields behind a module boundary.
    pub struct Token { proof: () }

    pub fn get_token() -> Option<Token> {
        Some(Token { proof: () })
    }
}

pub fn protected_work(token: token::Token) {
    println!("We have a token, so we can make assumptions.")
}

fn main() {
    if let Some(token) = token::get_token() {
        // We have a token, so we can do this work.
        protected_work(token);
    } else {
        // We could not get a token, so we can't call `protected_work`.
    }
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 95 minutes.

- Motivation: We want to be able to restrict user's access to functionality until they've performed a specific task.

  We can do this by defining a type the API consumer cannot construct on their own, through the privacy rules of structs and modules.

  Newtypes use the privacy rules in a similar way, to restrict construction unless a value is guaranteed to hold up an invariant at runtime.

- Ask: What is the purpose of the `proof: ()` field here?

  Without `proof: ()`, `Token` would have no private fields and users would be able to construct values of `Token` arbitrarily.

  Demonstrate: Try to construct the token manually in `main` and show the compilation error. Demonstrate: Remove the `proof` field from `Token` to show how users would be able to construct `Token` if it had no private fields.

access the `proof` field.

The API developer gets to define methods and functions that produce these tokens. The user does not.

The token becomes a proof that one has met the API developer's conditions of access for those tokens.

- Ask: How might an API developer accidentally introduce ways to circumvent this?

  Expect answers like "serialization implementations", other parser/"from string" implementations, or an implementation of `Default`.

Permission Tokens

Token types work well as a proof of checked permission.

```
mod admin {
    pub struct AdminToken(());

    pub fn get_admin(password: &str) -> Option<AdminToken> {
        if password == "Password123" { Some(AdminToken(())) } else { None }
    }
}

// We don't have to check that we have permissions, because
// the AdminToken argument is equivalent to such a check.
pub fn add_moderator(_: &admin::AdminToken, user: &str) {}

fn main() {
    if let Some(token) = admin::get_admin("Password123") {
        add_moderator(&token, "CoolUser");
    } else {
        eprintln!("Incorrect password! Could not prove privileges.")
    }
}
```

▼ *Speaker Notes*

- This example shows modelling gaining administrator privileges for a chat client with a password and giving a user a moderator rank once those privileges are gained. The `AdminToken` type acts as "proof of correct user privileges."

  The user asked for a password in-code and if we get the password correct, we get a `AdminToken` to perform administrator actions within a specific environment (here, a chat client).

  Once the permissions are gained, we can call the `add_moderator` function.

  We can't call that function without the token type, so by being able to call it at all all we can assume we have permissions.

- Demonstrate: Try to construct the `AdminToken` in `main` again to reiterate that the foundation of useful tokens is preventing their arbitrary construction.

Token Types with Data: Mutex Guards

Sometimes, a token type needs additional data. A mutex guard is an example of a token that represents permission + data.

```
use std::sync::{Arc, Mutex, MutexGuard};

fn main() {
    let mutex = Arc::new(Mutex::new(42));
    let try_mutex_guard: Result<MutexGuard<'_, _>, _> = mutex.lock();
    if let Ok(mut guarded) = try_mutex_guard {
        // The acquired MutexGuard is proof of exclusive access.
        *guarded = 451;
    }
}
```

▼ *Speaker Notes*

- Mutexes enforce mutual exclusion of read/write access to a value. We've covered Mutexes earlier in this course already (See: RAII/Mutex), but here we're looking at `MutexGuard` specifically.
- `MutexGuard` is a value generated by a `Mutex` that proves you have read/write access at that point in time.

  `MutexGuard` also holds onto a reference to the `Mutex` that generated it, with `Deref` and `DerefMut` implementations that give access to the data of `Mutex` while the underlying `Mutex` keeps that data private from the user.
- If `mutex.lock()` does not return a `MutexGuard`, you don't have permission to change the value within the mutex.

  Not only do you have no permission, but you have no means to access the mutex data unless you gain a `MutexGuard`.

  This contrasts with C++, where mutexes and lock guards do not control access to the data itself, acting only as a flag that a user must remember to check every time they read or manipulate data.
- Demonstrate: make the `mutex` variable mutable then try to dereference it to change its value. Show how there's no deref implementation for it, and no other way to get to the data held by it other than getting a mutex guard.

Variable-Specific Tokens (Branding 1/4)

What if we want to tie a token to a specific variable?

```
struct Bytes {
    bytes: Vec<u8>,
}
struct ProvenIndex(usize);

impl Bytes {
    fn get_index(&self, ix: usize) -> Option<ProvenIndex> {
        if ix < self.bytes.len() { Some(ProvenIndex(ix)) } else { None }
    }
    fn get_proven(&self, token: &ProvenIndex) -> u8 {
        unsafe { *self.bytes.get_unchecked(token.0) }
    }
}

fn main() {
    let data_1 = Bytes { bytes: vec![0, 1, 2] };
    if let Some(token_1) = data_1.get_index(2) {
        data_1.get_proven(&token_1); // Works fine!

        // let data_2 = Bytes { bytes: vec![0, 1] };
        // data_2.get_proven(&token_1); // Panics! Can we prevent this?
    }
}
```

▼ *Speaker Notes*

- What if we want to tie a token to a *specific variable* in our code? Can we do this in Rust's type system?
- Motivation: We want to have a Token Type that represents a known, valid index into a byte array.

  Once we have these proven indexes we would be able to avoid bounds checks entirely, as the tokens would act as the *proof of an existing index*.

  Since the index is known to be valid, `get_proven()` can skip the bounds check.

  In this example there's nothing stopping the proven index of one array being used on a different array. If an index is out of bounds in this case, it is undefined behavior.
- Demonstrate: Uncomment the `data_2.get_proven(&token_1);` line.

  The code here panics! We want to prevent this "crossover" of token types for indexes at compile time.

Expect students to not reach a good implementation from this, but be willing to experiment and follow through on suggestions.

- Ask: What are the alternatives, why are they not good enough?

  Expect runtime checking of index bounds, especially as both `Vec::get` and `Bytes::get_index` already uses runtime checking.

  Runtime bounds checking does not prevent the erroneous crossover in the first place, it only guarantees a panic.

- The kind of token-association we will be doing here is called Branding. This is an advanced technique that expands applicability of token types to more API designs.

- `GhostCell` is a prominent user of this, later slides will touch on it.

PhantomData and Lifetime Subtyping (Branding 2/4)

Idea:

- Use a lifetime as a unique brand for each token.
- Make lifetimes sufficiently distinct so that they don't implicitly convert into each other.

```
use std::marker::PhantomData;

#[derive(Default)]
struct InvariantLifetime<'id>(PhantomData<&'id ()>); // The main focus

struct Wrapper<'a> { value: u8, invariant: InvariantLifetime<'a> }

fn lifetime_separator<T>(value: u8, f: impl for<'a> FnOnce(Wrapper<'a>) ->
    f(Wrapper { value, invariant: InvariantLifetime::default() })
}

fn try_coerce_lifetimes<'a>(left: Wrapper<'a>, right: Wrapper<'a>) {}

fn main() {
    lifetime_separator(1, |wrapped_1| {
        lifetime_separator(2, |wrapped_2| {
            // We want this to NOT compile
            try_coerce_lifetimes(wrapped_1, wrapped_2);
        });
    });
}
```

▼ *Speaker Notes*

- In Rust, lifetimes can have subtyping relations between one another.

  This kind of relation allows the compiler to determine if one lifetime outlives another.

  Determining if a lifetime outlives another also allows us to say *the shortest common lifetime is the one that ends first*.

  This is useful in many cases, as it means two different lifetimes can be treated as if they were the same in the regions they do overlap.

  This is usually what we want. But here we want to use lifetimes as a way to distinguish values so we say that a token only applies to a single variable without

- **Goal**: We want two lifetimes that the rust compiler cannot determine if one outlives the other.

  We are using `try_coerce_lifetimes` as a compile-time check to see if the lifetimes have a common shorter lifetime (AKA being subtyped).

- Note: This slide compiles, by the end of this slide it should only compile when `subtyped_lifetimes` is commented out.

- There are two important parts of this code:

  - The `impl for<'a>` bound on the closure passed to `lifetime_separator`.
  - The way lifetimes are used in the parameter for `PhantomData`.

`for<'a>` bound on a Closure

- We are using `for<'a>` as a way of introducing a lifetime generic parameter to a function type and asking that the body of the function to work for all possible lifetimes.

  What this also does is remove some ability of the compiler to make assumptions about that specific lifetime for the function argument, as it must meet rust's borrow checking rules regardless of the "real" lifetime its arguments are going to have. The caller is substituting in actual lifetime, the function itself cannot.

  This is analogous to a forall (∀) quantifier in mathematics, or the way we introduce `<T>` as type variables, but only for lifetimes in trait bounds.

  When we write a function generic over a type `T`, we can't determine that type from within the function itself. Even if we call a function `fn foo<T, U>(first: T, second: U)` with two arguments of the same type, the body of this function cannot determine if `T` and `U` are the same type.

  This also prevents *the API consumer* from defining a lifetime themselves, which would allow them to circumvent the restrictions we want to impose.

PhantomData and Lifetime Variance

- We already know `PhantomData`, which can introduce a formal no-op usage of an otherwise unused type or a lifetime parameter.

Expect mentions of the Typestate pattern, tying together the lifetimes of owned values.

- Ask: In other languages, what is subtyping?

  Expect mentions of inheritance, being able to use a value of type `B` when a asked for a value of type `A` because `B` is a "subtype" of `A`.

- Rust does have Subtyping! But only for lifetimes.

  Ask: If one lifetime is a subtype of another lifetime, what might that mean?

  A lifetime is a "subtype" of another lifetime when it *outlives* that other lifetime.

- The way that lifetimes used by `PhantomData` behave depends not only on where the lifetime "comes from" but on how the reference is defined too.

  The reason this compiles is that the **Variance** of the lifetime inside of `InvariantLifetime` is too lenient.

  Note: Do not expect to get students to understand variance entirely here, just treat it as a kind of ladder of restrictiveness on the ability of lifetimes to establish subtyping relations.

- Ask: How can we make it more restrictive? How do we make a reference type more restrictive in rust?

  Expect or demonstrate: Making it `&'id mut ()` instead. This will not be enough!

  We need to use a **Variance** on lifetimes where subtyping cannot be inferred except on *identical lifetimes*. That is, the only subtype of `'a` the compiler can know is `'a` itself.

  Note: Again, do not try to get the whole class to understand variance. Treat it as a ladder of restrictiveness for now.

  Demonstrate: Move from `&'id ()` (covariant in lifetime and type), `&'id mut ()` (covariant in lifetime, invariant in type), `*mut &'id mut ()` (invariant in lifetime and type), and finally `*mut &'id ()` (invariant in lifetime but not type).

  Those last two should not compile, which means we've finally found candidates for how to bind lifetimes to `PhantomData` so they can't be compared to one another in this context.

  Reason: `*mut` means mutable raw pointer. Rust has mutable pointers! But you cannot reason about them in safe rust. Making this a mutable raw pointer to a

- Wrap up: We've introduced ways to stop the compiler from deciding that lifetimes are "similar enough" by choosing a Variance for a lifetime in `PhantomData` that is restrictive enough to prevent this slide from compiling.

  That is, we can now create variables that can exist in the same scope as each other, but whose types are automatically made different from one another per-variable without much boilerplate.

More to Explore

- The `for<'a>` quantifier is not just for function types. It is a **Higher-ranked trait bound**.

Implementing Branded Types (Branding 3/4)

Constructing branded types is different to how we construct non-branded types.

```
struct ProvenIndex<'id>(usize, InvariantLifetime<'id>);

struct Bytes<'id>(Vec<u8>, InvariantLifetime<'id>);

impl<'id> Bytes<'id> {
    fn new<T>(
        // The data we want to modify in this context.
        bytes: Vec<u8>,
        // The function that uniquely brands the lifetime of a `Bytes`
        f: impl for<'a> FnOnce(Bytes<'a>) -> T,
    ) -> T {
        f(Bytes(bytes, InvariantLifetime::default()),)
    }

    fn get_index(&self, ix: usize) -> Option<ProvenIndex<'id>> {
        if ix < self.0.len() { Some(ProvenIndex(ix,
InvariantLifetime::default())) }
        else { None }
    }

    fn get_proven(&self, ix: &ProvenIndex<'id>) -> u8 {
        debug_assert!(ix.0 < self.0.len());
        unsafe { *self.0.get_unchecked(ix.0) }
    }
}
```

▼ *Speaker Notes*

- Motivation: We want to have “proven indexes” for a type, and we don’t want those indexes to be usable by different variables of the same type. We also don’t want those indexes to escape a scope.

  Our Branded Type will be `Bytes` : a byte array.

  Our Branded Token will be `ProvenIndex` : an index known to be in range.

- There are several notable parts to this implementation:

  - `new` does not return a `Bytes` , instead asking for “starting data” and a use-once Closure that is passed a `Bytes` when it is called.
  - That `new` function has a `for<'a>` on its trait bound

- Ask: Why does `new` not return a `Bytes`?

  Answer: Because we need `Bytes` to have a unique lifetime controlled by the API.

- Ask: So what if `new()` returned `Bytes`, what is the specific harm that it would cause?

  Answer: Think about the signature of that hypothetical `new()` method:

  ```
  fn new<'a>() -> Bytes<'a> { ... }
  ```

  This would allow the API user to choose what the lifetime `'a` is, removing our ability to guarantee that the lifetimes between different instances of `Bytes` are unique and unable to be subtyped to one another.

- Ask: Why do we need both a `get_index` and a `get_proven`?

  Expect "Because we can't know if an index is occupied at compile time"

  Ask: Then what's the point of the proven indexes?

  Answer: Avoiding bounds checking while keeping knowledge of what indexes are occupied specific to individual variables, unable to erroneously be used on the wrong one.

  Note: The focus is not on only on avoiding overuse of bounds checks, but also on preventing that "cross over" of indexes.

Branded Types in Action (Branding 4/4)

```rust
use std::marker::PhantomData;

#[derive(Default)]
struct InvariantLifetime<'id>(PhantomData<*mut &'id ()>);
struct ProvenIndex<'id>(usize, InvariantLifetime<'id>);

struct Bytes<'id>(Vec<u8>, InvariantLifetime<'id>);

impl<'id> Bytes<'id> {
    fn new<T>(
        // The data we want to modify in this context.
        bytes: Vec<u8>,
        // The function that uniquely brands the lifetime of a `Bytes`
        f: impl for<'a> FnOnce(Bytes<'a>) -> T,
    ) -> T {
        f(Bytes(bytes, InvariantLifetime::default()))
    }

    fn get_index(&self, ix: usize) -> Option<ProvenIndex<'id>> {
        if ix < self.0.len() {
            Some(ProvenIndex(ix, InvariantLifetime::default()))
        } else {
            None
        }
    }

    fn get_proven(&self, ix: &ProvenIndex<'id>) -> u8 {
        self.0[ix.0]
    }
}

fn main() {
    let result = Bytes::new(vec![4, 5, 1], move |mut bytes_1| {
        Bytes::new(vec![4, 2], move |mut bytes_2| {
            let index_1 = bytes_1.get_index(2).unwrap();
            let index_2 = bytes_2.get_index(1).unwrap();
            bytes_1.get_proven(&index_1);
            bytes_2.get_proven(&index_2);
            // bytes_2.get_proven(&index_1); // ❌🔨
            "Computations done!"
        })
    });
    println!("{result}");
}
```

▼ *Speaker Notes*

- We now have the implementation ready, we can now write a program where token

- Demonstration: Uncomment the `bytes_2.get_proven(&index_1);` line and show that it does not compile when we use indexes from different variables.

- Ask: What operations can we perform that we can guarantee would produce a proven index?

  Expect a "push" implementation, suggested demo:

  ```
  fn push(&mut self, value: u8) -> ProvenIndex<'id> {
      self.0.push(value);
      ProvenIndex(self.0.len() - 1, InvariantLifetime::default())
  }
  ```

- Ask: Can we make this not just about a byte array, but as a general wrapper on `Vec<T>`?

  Trivial: Yes!

  Maybe demonstrate: Generalising `Bytes<'id>` into `BrandedVec<'id, T>`

- Ask: What other areas could we use something like this?

- The resulting token API is **highly restrictive**, but the things that it makes possible to prove as safe within the Rust type system are meaningful.

More to Explore

- GhostCell, a structure that allows for safe cyclic data structures in Rust (among other previously difficult to represent data structures), uses this kind of token type to make sure cells can't "escape" a context where we know where operations similar to those shown in these examples are safe.

  This "Branded Types" sequence of slides is based off their `BrandedVec` implementation in the paper, which covers many of the implementation details of this use case in more depth as a gentle introduction to how `GhostCell` itself is implemented and used in practice.

  GhostCell also uses formal checks outside of Rust's type system to prove that the things it allows within this kind of context (lifetime branding) are safe.

Welcome to Unsafe Rust

IMPORTANT: THIS MODULE IS IN AN EARLY STAGE OF DEVELOPMENT

Please do not consider this module of Comprehensive Rust to be complete. With that in mind, your feedback, comments, and especially your concerns, are very welcome.

To comment on this module's development, please use the GitHub issue tracker.

The `unsafe` keyword is easy to type, but hard to master. When used appropriately, it forms a useful and indeed essential part of the Rust programming language.

By the end of this deep dive, you'll know how to work with `unsafe` code, review others' changes that include the `unsafe` keyword, and produce your own.

What you'll learn:

- What the terms undefined behavior, soundness, and safety mean
- Why the `unsafe` keyword exists in the Rust language
- How to write your own code using `unsafe` safely
- How to review `unsafe` code

Links to other sections of the course

The `unsafe` keyword has treatment in:

- *Rust Fundamentals*, the main module of Comprehensive Rust, includes a session on Unsafe Rust in its last day.
- *Rust in Chromium* discusses how to interoperate with C++. Consult that material if you are looking into FFI.
- *Bare Metal Rust* uses unsafe heavily to interact with the underlying host, among other things.

Setting Up

Local Rust installation

You should have a Rust compiler installed that supports the 2024 edition of the language, which is any version of rustc higher than 1.84.

```
$ rustc --version
rustc 1.87
```

(Optional) Create a local instance of the course

```
$ git clone --depth=1 https://github.com/google/comprehensive-rust.git
Cloning into 'comprehensive-rust'...
...
$ cd comprehensive-rust
$ cargo install-tools
...
$ cargo serve # then open http://127.0.0.1:3000/ in a browser
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

Ask everyone to confirm that everyone is able to execute `rustc` with a version older that 1.87.

For those people who do not, tell them that we'll resolve that in the break.

Motivations

We know that writing code without the guarantees that Rust provides ...

> "Use-after-free (UAF), integer overflows, and out of bounds (OOB) reads/writes comprise 90% of vulnerabilities with OOB being the most common."
>
> –— **Jeff Vander Stoep and Chong Zang**, Google. "Queue the Hardening Enhancements"

... so why is `unsafe` part of the language?

This segment should take about 20 minutes. It contains:

Slide	Duration
Motivations	1 minute
Interoperability	5 minutes
Data Structures	5 minutes
Performance	5 minutes

▼ *Speaker Notes*

This slide should take about 1 minute.

The `unsafe` keyword exists because there is no compiler technology available today that makes it obsolete. Compilers cannot verify everything.

TODO: Refactor this content into multiple slides as this slide is intended as an introduction to the motivations only, rather than to be an elaborate discussion of the whole problem.

Interoperability

Language interoperability allows you to:

- Call functions written in other languages from Rust
- Write functions in Rust that are callable from other languages

However, this requires unsafe.

```
unsafe extern "C" {
    safe fn random() -> libc::c_long;
}

fn main() {
    let a = random() as i64;
    println!("{a:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

The Rust compiler can't enforce any safety guarantees for programs that it hasn't compiled, so it delegates that responsibility to you through the unsafe keyword.

The code example we're seeing shows how to call the random function provided by libc within Rust. libc is available to scripts in the Rust Playground.

This uses Rust's *foreign function interface*.

This isn't the only style of interoperability, however it is the method that's needed if you want to work between Rust and some other language in a zero cost way. Another important strategy is message passing.

Message passing avoids unsafe, but serialization, allocation, data transfer and parsing all take energy and time.

Answers to questions

symbols, including `random`, being available to our program.

- *What is the "safe" keyword?*
  It allows callers to call the function without needing to wrap that call in `unsafe`. The `safe` function qualifier was introduced in the 2024 edition of Rust and can only be used within `extern` blocks. It was introduced because `unsafe` became a mandatory qualifier for `extern` blocks in that edition.
- *What is the `std::ffi::c_long` type?*
  According to the C standard, an integer that's at least 32 bits wide. On today's systems, It's an `i32` on Windows and an `i64` on Linux.

Consideration: type safety

Modify the code example to remove the need for type casting later. Discuss the potential UB - long's width is defined by the target.

```
unsafe extern "C" {
    safe fn random() -> i64;
}

fn main() {
    let a = random();
    println!("{a:?}");
}
```

Changes from the original:

```
unsafe extern "C" {
-    safe fn random() -> libc::c_long;
+    safe fn random() -> i64;
}

fn main() {
-    let a = random() as i64;
+    let a = random();
    println!("{a:?}");
}
```

It's also possible to completely ignore the intended type and create undefined behavior in multiple ways. The code below produces output most of the time, but generally results in a stack overflow. It may also produce illegal `char` values. Although `char` is represented in 4 bytes (32 bits), not all bit patterns are permitted as a `char`.

```
unsafe extern "C" {
    safe fn random() -> [char; 2];
}

fn main() {
    let a = random();
    println!("{a:?}");
}
```

Changes from the original:

```
unsafe extern "C" {
-    safe fn random() -> libc::c_long;
+    safe fn random() -> [char; 2];
}

fn main() {
-    let a = random() as i64;
-    println!("{a}");
+    let a = random();
+    println!("{a:?}");
}
```

Attempting to print a `[char; 2]` from randomly generated input will often produce strange output, including:

```
thread 'main' panicked at library/std/src/io/stdio.rs:1165:9:
failed printing to stdout: Bad address (os error 14)


thread 'main' has overflowed its stack
fatal runtime error: stack overflow, aborting
```

Mention that type safety is generally not a large concern in practice. Tools that produce wrappers automatically, i.e. bindgen, are excellent at reading header files and producing values of the correct type.

Consideration: Ownership and lifetime management

While libc's `random` function doesn't use pointers, many do. This creates many more possibilities for unsoundness.

For example, some C libraries expose functions that write to static buffers that are re-used between calls.

```
use std::ffi::{CStr, c_char};
use std::time::{SystemTime, UNIX_EPOCH};

unsafe extern "C" {
    /// Create a formatted time based on time `t`, including trailing
newline.
    /// Read `man 3 ctime` details.
    fn ctime(t: *const libc::time_t) -> *const c_char;
}

unsafe fn format_timestamp<'a>(t: u64) -> &'a str {
    let t = t as libc::time_t;

    unsafe {
        let fmt_ptr = ctime(&t);
        CStr::from_ptr(fmt_ptr).to_str().unwrap()
    }
}

fn main() {
    let now = SystemTime::now().duration_since(UNIX_EPOCH).unwrap();

    let now = now.as_secs();
    let now_fmt = unsafe { format_timestamp(now) };
    print!("now (1): {}", now_fmt);

    let future = now + 60;
    let future_fmt = unsafe { format_timestamp(future) };
    print!("future:  {}", future_fmt);

    print!("now (2): {}", now_fmt);
}
```

Aside: Lifetimes in the `format_timestamp()` function

Neither `'a`, nor `'static`, correctly describe the lifetime of the string that's returned. Rust treats it as an immutable reference, but subsequent calls to `ctime` will overwrite the static buffer that the string occupies.

Consideration: Representation mismatch

Different programming languages have made different design decisions and this can create impedance mismatches between different domains.

whereas `std::string` does not. In C, text is represented by a null-terminated sequence of bytes (`char*`).

```
fn main() {
    let c_repr = b"Hello, C\0";
    let rust_repr = (b"Hello, Rust", 11);

    let c: &str = unsafe {
        let ptr = c_repr.as_ptr() as *const i8;
        std::ffi::CStr::from_ptr(ptr).to_str().unwrap()
    };
    println!("{c}");

    let rust: &str = unsafe {
        let ptr = rust_repr.0.as_ptr();
        let bytes = std::slice::from_raw_parts(ptr, rust_repr.1);
        std::str::from_utf8_unchecked(bytes)
    };
    println!("{rust}");
}
```

Data Structures

Some families of data structures are impossible to create in safe Rust.

- graphs
- bit twiddling
- self-referential types
- intrusive data structures

▼ *Speaker Notes*

This slide should take about 5 minutes.

Graphs: General-purpose graphs cannot be created as they may need to represent cycles. Cycles are impossible for the type system to reason about.

Bit twiddling: Overloading bits with multiple meanings. Examples include using the NaN bits in `f64` for some other purpose or the higher-order bits of pointers on `x86_64` platforms. This is somewhat common when writing language interpreters to keep representations within the word size the target platform.

Self-referential types are too hard for the borrow checker to verify.

Intrusive data structures: store structural metadata (like pointers to other elements) inside the elements themselves, which requires careful handling of aliasing.

Performance

TODO: Stub for now

It's easy to think of performance as the main reason for unsafe, but high performance code makes up the minority of unsafe blocks.

Foundations

Some fundamental concepts and terms.

This segment should take about 25 minutes. It contains:

Slide	Duration
What is unsafe?	10 minutes
When is unsafe used?	2 minutes
Data structures are safe	2 minutes
Actions might not be	2 minutes
Less powerful than it seems	10 minutes

What is “unsafety”?

Unsafe Rust is a superset of Safe Rust.

Let’s create a list of things that are enabled by the `unsafe` keyword.

▼ *Speaker Notes*

This slide should take about 6 minutes.

Definitions from authoritative docs:

From the unsafe keyword’s documentation:

> Code or interfaces whose memory safety cannot be verified by the type system.
>
> …
>
> Here are the abilities Unsafe Rust has in addition to Safe Rust:
>
> - Dereference raw pointers
> - Implement unsafe traits
> - Call unsafe functions
> - Mutate statics (including external ones)
> - Access fields of unions

From the reference

> The following language level features cannot be used in the safe subset of Rust:
>
> - Dereferencing a raw pointer.
> - Reading or writing a mutable or external static variable.
> - Accessing a field of a union, other than to assign to it.
> - Calling an unsafe function (including an intrinsic or foreign function).
> - Calling a safe function marked with a target_feature from a function that does not have a target_feature attribute enabling the same features (see attributes.codegen.target_feature.safety-restrictions).
> - Implementing an unsafe trait.
> - Declaring an extern block.

Group exercise

> You may have a group of learners who are not familiar with each other yet. This is a way for you to gather some data about their confidence levels and the psychological safety that they're feeling.

Part 1: Informal definition

> Use this to gauge the confidence level of the group. If they are uncertain, then tailor the next section to be more directed.

Ask the class: **By raising your hand, indicate if you would feel comfortable defining unsafe?**

If anyone's feeling confident, allow them to try to explain.

Part 2: Evidence gathering

Ask the class to spend 3-5 minutes.

- Find a use of the unsafe keyword. What contract/invariant/pre-condition is being established or satisfied?
- Write down terms that need to be defined (unsafe, memory safety, soundness, undefined behavior)

Part 3: Write a working definition

Part 4: Remarks

Mention that we'll be reviewing our definition at the end of the day.

Note: Avoid detailed discussion about precise semantics of memory safety

discussion. It can undermine confidence in less experienced learners.

Perhaps refer people who wish to discuss this to the discussion within the official documentation for pointer types (excerpt below) as a place for further research.

> Many functions in this module take raw pointers as arguments and read from or write to them. For this to be safe, these pointers must be *valid* for the given access.
>
> ...
>
> The precise rules for validity are not determined yet.

When is unsafe used?

The unsafe keyword indicates that the programmer is responsible for upholding Rust's safety guarantees.

The keyword has two roles:

- define pre-conditions that must be satisfied
- assert to the compiler (= promise) that those defined pre-conditions are satisfied

Further references

- The unsafe keyword chapter of the Rust Reference

▼ *Speaker Notes*

This slide should take about 2 minutes.

Places where pre-conditions can be defined (Role 1)

- unsafe functions (`unsafe fn foo() { ... }`). Example: `get_unchecked` method on slices, which requires callers to verify that the index is in-bounds.
- unsafe traits (`unsafe trait`). Examples: `Send` and `Sync` marker traits in the standard library.

Places where pre-conditions must be satisfied (Role 2)

- unsafe blocks (`unafe { ... }`)
- implementing unsafe traits (`unsafe impl`)
- access external items (`unsafe extern`)
- adding unsafe attributes o an item. Examples: `export_name` , `link_section` and `no_mangle` . Usage: `#[unsafe(no_mangle)]`

Data structures are safe ...

Data structures are inert. They cannot do any harm by themselves.

Safe Rust code can create raw pointers:

```
fn main() {
    let n: i64 = 12345;
    let safe = &raw const n;
    println!("{safe:p}");
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

Consider a raw pointer to an integer, i.e., the value `safe` is the raw pointer type `*const i64` . Raw pointers can be out-of-bounds, misaligned, or be null. But the unsafe keyword is not required when creating them.

... but actions on them might not be

```
fn main() {
    let n: i64 = 12345;
    let safe = &n as *const _;
    println!("{safe:p}");
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

Modify the example to de-reference `safe` without an `unsafe` block.

Less powerful than it seems

The `unsafe` keyword does not allow you to break Rust.

```
use std::mem::transmute;

let orig = b"RUST";
let n: i32 = unsafe { transmute(orig) };

println!("{n}")
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

Suggested outline

- Request that someone explains what `std::mem::transmute` does
- Discuss why it doesn't compile
- Fix the code

Expected compiler output

```
   Compiling playground v0.0.1 (/playground)
error[E0512]: cannot transmute between types of different sizes, or
dependently-sized types
 --> src/main.rs:5:27
  |
5 |     let n: i32 = unsafe { transmute(orig) };
  |                           ^^^^^^^^^
  |
  = note: source type: `&[u8; 4]` (64 bits)
  = note: target type: `i32` (32 bits)
```

Suggested change

```
- let n: i32 = unsafe { transmute(orig) };
```

Notes on less familiar Rust

- the `b` prefix on a string literal marks it as byte slice (`&[u8]`) rather than a string slice (`&str`)

Thanks!

Thank you for taking Comprehensive Rust 🦀! We hope you enjoyed it and that it was useful.

We've had a lot of fun putting the course together. The course is not perfect, so if you spotted any mistakes or have ideas for improvements, please get in contact with us on GitHub. We would love to hear from you.

▼ *Speaker Notes*

- Thank you for reading the speaker notes! We hope they have been useful. If you find pages without notes, please send us a PR and link it to issue #1083. We are also very grateful for fixes and improvements to the existing notes.

Glossary

The following is a glossary which aims to give a short definition of many Rust terms. For translations, this also serves to connect the term back to the English original.

allocate:
Dynamic memory allocation on the heap.

array:
A fixed-size collection of elements of the same type, stored contiguously in memory. See Arrays.

associated type:
A type associated with a specific trait. Useful for defining the relationship between types.

Bare-metal Rust:
Low-level Rust development, often deployed to a system without an operating system. See Bare-metal Rust.

block:
See Blocks and *scope*.

borrow:
See Borrowing.

borrow checker:
The part of the Rust compiler which checks that all borrows are valid.

brace:
`{` and `}`. Also called *curly brace*, they delimit *blocks*.

channel:
Used to safely pass messages between threads.

concurrency:
The execution of multiple tasks or processes at the same time. See Welcome to Concurrency in Rust.

constant:
A value that does not change during the execution of a program. See const.

control flow:
The order in which the individual statements or instructions are executed in a program. See Control Flow Basics.

crash:
An unexpected and unhandled failure or termination of a program. See panic.

enumeration:
A data type that holds one of several named constants, possibly with an associated tuple or struct. See enum.

error:
An unexpected condition or result that deviates from the expected behavior. See Error

error handling:
The process of managing and responding to errors that occur during program execution.

function:
A reusable block of code that performs a specific task. See Functions.

garbage collector:
A mechanism that automatically frees up memory occupied by objects that are no longer in use. See Approaches to Memory Management.

generics:
A feature that allows writing code with placeholders for types, enabling code reuse with different data types. See Generics.

immutable:
Unable to be changed after creation. See Variables.

integration test:
A type of test that verifies the interactions between different parts or components of a system. See Other Types of Tests.

library:
A collection of precompiled routines or code that can be used by programs. See Modules.

macro:
Rust macros can be recognized by a `!` in the name. Macros are used when normal functions are not enough. A typical example is `format!`, which takes a variable number of arguments, which isn't supported by Rust functions.

`main` function:
Rust programs start executing with the `main` function.

match:
A control flow construct in Rust that allows for pattern matching on the value of an expression.

memory leak:
A situation where a program fails to release memory that is no longer needed, leading to a gradual increase in memory usage. See Approaches to Memory Management.

method:
A function associated with an object or a type in Rust. See Methods.

module:
A namespace that contains definitions, such as functions, types, or traits, to organize code in Rust. See Modules.

move:
The transfer of ownership of a value from one variable to another in Rust. See Move Semantics.

mutable:
A property in Rust that allows variables to be modified after they have been declared.

ownership:

panic:
An unrecoverable error condition in Rust that results in the termination of the program. See Panics.

pattern:
A combination of values, literals, or structures that can be matched against an expression in Rust. See Pattern Matching.

payload:
The data or information carried by a message, event, or data structure.

receiver:
The first parameter in a Rust method that represents the instance on which the method is called.

reference:
A non-owning pointer to a value that borrows it without transferring ownership. References can be shared (immutable) or exclusive (mutable).

reference counting:
A memory management technique in which the number of references to an object is tracked, and the object is deallocated when the count reaches zero. See Rc.

Rust:
A systems programming language that focuses on safety, performance, and concurrency. See What is Rust?.

safe:
Refers to code that adheres to Rust's ownership and borrowing rules, preventing memory-related errors. See Unsafe Rust.

slice:
A dynamically-sized view into a contiguous sequence, such as an array or vector. Unlike arrays, slices have a size determined at runtime. See Slices.

scope:
The region of a program where a variable is valid and can be used. See Blocks and Scopes.

standard library:
A collection of modules providing essential functionality in Rust. See Standard Library.

static:
A keyword in Rust used to define static variables or items with a `'static` lifetime. See static.

string:
A data type storing textual data. See Strings.

struct:
A composite data type in Rust that groups together variables of different types under a single name. See Structs.

test:
A function that tests the correctness of other code. Rust has a built-in test runner. See Testing.

thread safety:
The property of a program that ensures correct behavior in a multithreaded environment. See Send and Sync.

trait:
A collection of methods defined for an unknown type, providing a way to achieve polymorphism in Rust. See Traits.

trait bound:
An abstraction where you can require types to implement some traits of your interest. See Trait Bounds.

tuple:
A composite data type that contains variables of different types. Tuple fields have no names, and are accessed by their ordinal numbers. See Tuples.

type:
A classification that specifies which operations can be performed on values of a particular kind in Rust. See Types and Values.

type inference:
The ability of the Rust compiler to deduce the type of a variable or expression. See Type Inference.

undefined behavior:
Actions or conditions in Rust that have no specified result, often leading to unpredictable program behavior. See Unsafe Rust.

union:
A data type that can hold values of different types but only one at a time. See Unions.

unit test:
Rust comes with built-in support for running small unit tests and larger integration tests. See Unit Tests.

unit type:
Type that holds no data, written as a tuple with no members. See speaker notes on Functions.

unsafe:
The subset of Rust which allows you to trigger *undefined behavior*. See Unsafe Rust.

variable:
A memory location storing data. Variables are valid in a *scope*. See Variables.

Other Rust Resources

The Rust community has created a wealth of high-quality and free resources online.

Official Documentation

The Rust project hosts many resources. These cover Rust in general:

- **The Rust Programming Language: the canonical free book about Rust. Covers the** language in detail and includes a few projects for people to build.
- **Rust By Example: covers the Rust syntax via a series of examples which showcase** different constructs. Sometimes includes small exercises where you are asked to expand on the code in the examples.
- **Rust Standard Library: full documentation of the standard library for Rust.**
- **The Rust Reference: an incomplete book which describes the Rust grammar and** memory model.
- **Rust API Guidelines: recommendations on how to design APIs.**

More specialized guides hosted on the official Rust site:

- **The Rustonomicon: covers unsafe Rust, including working with raw pointers and** interfacing with other languages (FFI).
- **Asynchronous Programming in Rust: covers the new asynchronous programming** model which was introduced after the Rust Book was written.
- **The Embedded Rust Book: an introduction to using Rust on embedded devices** without an operating system.

Unofficial Learning Material

A small selection of other guides and tutorial for Rust:

- **Learn Rust the Dangerous Way: covers Rust from the perspective of low-level C** programmers.
- **Rust for Embedded C Programmers: covers Rust from the perspective of developers** who write firmware in C.
- **Rust for professionals: covers the syntax of Rust using side-by-side comparisons with** other languages such as C, C++, Java, JavaScript, and Python.
- **Rust on Exercism: 100+ exercises to help you learn Rust.**

async/await are also covered.

Advanced testing for Rust applications: a self-paced workshop that goes beyond Rust's built-in testing framework. It covers `googletest`, snapshot testing, mocking as well as how to write your own custom test harness.

Beginner's Series to Rust and Take your first steps with Rust: two Rust guides aimed at new developers. The first is a set of 35 videos and the second is a set of 11 modules which covers Rust syntax and basic constructs.

Learn Rust With Entirely Too Many Linked Lists: in-depth exploration of Rust's memory management rules, through implementing a few different types of list structures.

The Little Book of Rust Macros: covers many details on Rust macros with practical examples.

Please see the Little Book of Rust Books for even more Rust books.

Credits

The material here builds on top of the many great sources of Rust documentation. See the page on other resources for a full list of useful resources.

The material of Comprehensive Rust is licensed under the terms of the Apache 2.0 license, please see `LICENSE` for details.

Rust by Example

Some examples and exercises have been copied and adapted from Rust by Example. Please see the `third_party/rust-by-example/` directory for details, including the license terms.

Rust on Exercism

Some exercises have been copied and adapted from Rust on Exercism. Please see the `third_party/rust-on-exercism/` directory for details, including the license terms.

CXX

The Interoperability with C++ section uses an image from CXX. Please see the