Lecture 4

Generics and Traits

Enum sizing

```
struct Node {
    value: u32,
    next: Box<Node>,
}
```

Size of Node? Sum of members.

```
std::mem::size_of::<Node>() == 4 + 8;
```

```
enum Option<u32> {
    Some(u32),
    None,
}
```

Size of Option? Size of largest member, plus up to 8 bytes for tracking which variant is active

```
size_of<Option<u32>>() == 4 + max(4,
0)
Some(9): 0x00000001 0x00000009
None:    0x00000000 0x00000000
```

Enum sizing Optimization

```
enum Option<&u32> {
    Some(&u32),
    None,
}
```

```
Some(0x32ab0012): 0x00000001 0x32ab0012
None:             0x00000000 0x00000000
```

But wait! References are guaranteed to never be null, so we can optimize.

```
Some(0x32ab0012): 0x32ab0012
None:             0x00000000
```

```
enum Option<u32> {
    Some(u32),
    None,
}
```

Size of Option? Size of largest member, plus up to 8 bytes for tracking which variant is active

```
size_of<Option<u32>>() == 4 + max(4,
0)
Some(9): 0x00000001 0x00000009
None:    0x00000000 0x00000000
```

Automatic referencing in method calls

```rust
struct Point(i32, i32);
impl Point {
    fn takes_ref(&self) {}
    fn takes_ownership(self) {}
}

fn main() {
    let as_owned = Point(1, 2);
    let as_ref = &as_owned;
    // auto reference
    as_owned.takes_ref();
    // auto dereference
    as_ref.takes_ownership();
}
```

`self` parameter will be automatically **referenced** and **dereferenced** to match method call

- Only happens to `self` parameter, not other parameters

PLQ Questions

When to use `Option` vs `Result`?

It's a bit of a blurry distinction, but there are some rules of thumb.

- If it's more of an "exists" rather than an "error" relationship, definitely use `Option`
  - E.g. querying an element from a collection
- If you don't care about the "error" value, then you *might* want to use an `Option`
  - Consider future extensibility
- *Exceptional* circumstances (e.g. a server is down, no disk space), use a `Result`

Ultimately it's up to you, no choice is "wrong" *per se*—it's more philosophical in nature.

Common `Result` design patterns

Enums are a great way to package multiple error types into one.

```
enum MyError {
    IoError(IoError),
    ServerError(ServerError),
}
```

Common `Result` design patterns

Sometimes it's convenient to just make your error type `&'static str`.

```
fn fallible() -> Result<T, &'static str> {
    // ...

    if(error_condition) {
        return Err("error description goes here");
    }

    // ...
}
```

How is `drop` implemented?

Normal Rust function or compiler magic?

How would *you* implement it?

```rust
fn my_drop(value: String) {}

fn main() {
    let my_string = "Hello, world!".to_owned();

    my_drop(my_string);
}
```

```rust
fn my_drop(value: String) {}

fn main() {
    let my_string = "Hello, world!".to_owned();

    my_drop(my_string);

    println!("{my_string}");
}
```

```
Diagnostics:
1. borrow of moved value: `my_string`
   value borrowed here after move [E0382]
2. borrow of moved value: `my_string`
   value borrowed here after move [E0382]
```

Works like expected

Type ‘S’ or ‘/’ to search, ‘?’ for more options...

std::mem

Function drop

Since 1.0.0 · Source

Settings Help Summary

```
pub fn drop<T>(_x: T)
```

Disposes of a value.

This does so by calling the argument’s implementation of `Drop`.

This effectively does nothing for types which implement `Copy`, e.g. integers. Such values are copied and *then* moved into the function, so the value persists after this function call.

This function is not magic; it is literally defined as

```
pub fn drop<T>(_x: T) {}
```

Because `_x` is moved into the function, it is automatically dropped before the function returns.

https://doc.rust-lang.org

More ways to use `Option`

`unwrap(self)`	Returns the inner value, otherwise **panics**.
`unwrap_or(self, default: T)`	Returns the inner value, otherwise some “default” value.
`unwrap_or_default(self)`	Same as above, but uses the `Default` trait instead (will discuss this).
`map(self, f: F)`	Uses a closure to *transform* the inner value. ● If `self` is `None`, then the function returns `None` ● If `self` is `Some(x)`, then it calls `f(x)` and returns `Some(f(x))`
`and_then(self, f: F)`	Same as `map`, but “flattens” the returned value. ● If `self` is `None`, then the function still returns `None` ● If `self` is `Some(x)`, then it calls `f(x)` and returns **`f(x)`** ● Requires that `f` returns an `Option` ● Rust’s equivalent of a monadic bind operator

Today: Generics

Generics: Motivating Example

```
enum NumOption {
    Some(u32),
    None
}
```

Boo! Bad!

- Need to duplicate code for every option type you want
- Need to duplicate functions that take `Option` to work on every option type

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

Yes! Better!

- Template for declaring any kind of `Option` you need
- Lets you define functionality for an `Option` of any type

Generics

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

Structs or enums can be generic over one or more types

```
fn first<T,U>(x: T, y: U) -> T {
    x
}
```

Correspondingly, functions can be generic over one or more types

Using Generic Types

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

fn main() {
    // type is inferred, annotation is optional
    let x: Opt<bool> = Opt::Some(true);
}
```

Struct generic types are often inferred

Calling generic functions

```
fn first<T,U>(x: T, y: U) -> T {
    x
}

fn main() {
    // automatically infers the type of T and U
    first(1, "hello");
    // Manually specify type of T and U
    first::<u32,&str>(1, "hello");
}
```

turbofish ::<>

Funky syntax to manually specify function generics. Helps resolve a parsing ambiguity that’s present in C++

How do `impl` and Generics Interact?

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

impl Opt<T> {
    fn unwrap(self) -> T {
        match self {
            Opt::Some(v) => v,
            Opt::None => panic!()
        }
    }

}
```

What's wrong with this?

```
error[E0412]: cannot find type `T` in this scope
  --> scratch.rs:337:10
    |
337 | impl Opt<T> {
    |          ^ not found in this scope
    |
```

How do `impl` and Generics Interact?

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

impl<T> Opt<T> {
    fn unwrap(self) -> T {
        match self {
            Opt::Some(v) => v,
            Opt::None => panic!()
        }
    }

}
```

Generic impl block

"For all T, there is an impl for the type `Opt<T>`"

Why would you ever not `impl<T>`?

```rust
impl Opt<u32> {
    fn unwrap(self) -> u32 {
        match self {
            Opt::Some(v) => v,
            Opt::None => panic!()
        }
    }
}
```

How do `impl` and Generics Interact?

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

impl<T> Opt<T> {
    fn swap<U>(&self, value: U) -> Opt<U> {
        ...
    }




}
```

Different levels of generics

- T is in scope for the entire `impl` block
- U is local to the function scope

More Advanced `impl`s

```
impl<T> Opt<Opt<T>> {
    fn flatten(self) -> Opt<T> {
        match self {
            Opt::Some(Opt::Some(x)) => Opt::Some(x),
            Opt::Some(Opt::None) => Opt::None,
            Opt::None => Opt::None
        }
    }
}
```

Impl blocks can be for arbitrarily complex types, not just simple structs or enums

How are Generics Implemented?

```
enum Opt<T> {
    Some(T),
    None
}
fn main() {
    let x: Opt<bool> = Opt::Some(true);
}
```

Q: What is a generic type?

A: Instructions for how to generate code for a specific `Opt` like `Opt<u32>`. No code generated

Code generated only for case `Opt<bool>`. Exact same code as if you had written `BoolOpt` manually

- No runtime cost
- Some compile time cost
- Some binary size cost
- Same as C++
- “Monomorphization”

Comparing `Option`s

How to write a function that takes two `Options` and

1. If both are `Some`, returns true if the first is greater than the second
2. Otherwise, returns `None`

Comparing Opts

```
fn opt_gr<T>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

Comparing Opts

```rust
fn opt_gr<T>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

```
error[E0369]: binary operation `>` cannot be applied to type `T`
--> scratch.rs:3:53
  |
3 |         (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
  |                                                   - ^ - T
  |                                                   |
  |                                                   T
```

Comparing `Opt`s

```
                          HashMap          HashMap
fn opt_gr<T>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
                                                      ???
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

```
                                                            HashMap
error[E0369]: binary operation `>` cannot be applied to type `T`
--> scratch.rs:3:53
  |
3 |         (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
  |                                                   - ^ - T
  |                                                   |
  |                                                   T
```

Takeaway

`impl<T>` and `fn foo<T>` aren't very useful :(

You can't do much with `T`:

1. Return it
2. Wrap it in a struct/enum
3. Pass it to another function
4. Not much else

What might we want to do with `T`?

- Addition/subtraction
- Printing
- Copying
- Checking equality
- Dereference

Bringing `Ord`er

```
fn opt_gr<T: Ord>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

Can't call `opt_gr` with just any `T`

- `T` has to have an order

`Ord` is a "trait"

Traits

Traits

```
trait ToInt {
    fn to_int(&self) -> u32;
}

impl ToInt for f32 {
    fn to_int(&self) -> u32 {
        self.to_bits()
    }
}

fn add<T: ToInt>(a: T, b: T) -> u32 {
    a.to_int() + b.to_int()
}
```

Traits define a set of methods without implementations

Types can provide implementations to "implement" that trait

"`f32` implements `ToInt`"

Generic functions can restrict inputs to only types that implement a certain trait

Yes, they're similar to interfaces

Traits

```
fn foo<T: ToInt + Ord>(a: T, b: T) -> u32 {
    if a > b {
        a.to_int()
    } else {
        b.to_int()
    }
}
```

Without **types,** function **arguments** could be anything. **Types** restrict the domain of **arguments**

Without **traits,** function **generics** could be anything. **Traits** restrict the domain of **generics**

In an alternate universe...

```
fn opt_gr<T>(a: Opt<T>, b: Opt<T>) -> Opt<bool> {
    match (a, b) {
        (Opt::Some(x), Opt::Some(y)) => Opt::Some(x > y),
        _ => Opt::None
    }
}
```

What if this compiled?

```
fn main() {
    opt_gr(
        Opt::Some(1),
        Opt::Some(2));
}
```

And this didn't?

```
fn main() {
    opt_gr(
        Opt::Some(HashMap::new()),
        Opt::Some(HashMap::new()));
}
```

Is this better or worse than the way Rust does things?

Common Trait Speedrun!

Traits are great for abstracting your own code by organizing shared behavior, but there's also many useful traits built in to the standard library that the language treats specially.

Let's explore some...

Common Trait Speedrun: `Default`

```rust
trait Default {
    fn default() -> Self;
}

fn main() {
    let x: String = Default::default();
}
```

```rust
fn main() {
    let x: u32 = Default::default();
    let o: Option<String> =
Default::default();
}
```

Almost like overloading!

`Default` is implemented by types that have a default value

- Requires a zero-argument function that returns the type.

What's this?

`Self` refers to the type that the current impl block is for

Common Trait Speedrun: `Clone`

```rust
trait Clone {
    fn clone(&self) -> Self;
}

fn main() {
    let s1 = String::new("Hello");
    let s2 = s1.clone()
}
```

`Clone` is implemented by types that can be deep-copied

Common Trait Speedrun: `Display`/`Debug`

```rust
trait Display {
    fn fmt(&self, f: &mut Formatter)
        -> Result<(), Error>;
}
trait Debug {
    fn fmt(&self, f: &mut Formatter)
        -> Result<(), Error>;
}
```

```rust
fn main() {
    let s = String::new("Test\n");
    // Uses Display
    println!("{}", s);
    // Uses Debug
    println!("{:?}", s);
}
```

`Display` and `Debug` define how a value can be converted to a string for printing

- `Display` for a user-facing representation
- `Debug` for a developer-facing representation

Instead of returning a string directly, work with a `Formatter` object to allow additional configuration (e.g. padding)

Quiz Time

```rust
fn f<T: /* ??? */>(t: &T) {
    let t2 = t.clone();
    println!("{}", t2);
}
```

What is the smallest set of trait bounds on `T` needed to make this function type-check?

A: `Clone`

B: `Clone + Display`

C: `Clone + Display + Debug`

D: <none>

Common Trait Speedrun: `PartialEq`

```
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;



}
impl PartialEq for u32 { ... }

fn main() {
    let x: u32 = 1;
    let y: u32 = 2;
    x == y;

}
```

`PartialEq` is for types that can be compared for equality

symmetric: a == b -> b == a

transitive: a == b && b == c -> a == c

Compiler uses `PartialEq` impl for `==` operator

Common Trait Speedrun: `PartialEq`

```
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;

    fn ne(&self, other: &Self) -> bool {
        !self.eq(other)
    }
}
impl PartialEq for u32 { ... }

fn main() {
    let x: u32 = 1;
    let y: u32 = 2;
    x == y;
    x != y;
}
```

`PartialEq` is for types that can be compared for equality

symmetric: a == b -> b == a

transitive: a == b && b == c -> a == c

What's this?

"Provided method"—implemented for free once you implement the required methods

Compiler uses `PartialEq` impl for `==` operator

Common Trait Speedrun: PartialEq

Ok, but where's plain old Eq?

```
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;

    fn ne(&self, other: &Self) -> bool {
        !self.eq(other)
    }
}
impl PartialEq for u32 { ... }

fn main() {
    let x: u32 = 1;
    let y: u32 = 2;
    x == y;
    x != y;
}
```

PartialEq is for types that can be compared for equality

symmetric: a == b -> b == a

transitive: a == b && b == c -> a == c

What's missing here?

"Provided method"—implemented for free once you implement the required methods

Compiler uses PartialEq impl for == operator

Common Trait Speedrun: Eq

```
trait Eq: PartialEq { }

impl Eq for u32 {}
```

Eq is for equality relations

symmetric: a == b -> b == a

transitive: a == b && b == c -> a == c

reflexive: a == a

What's this?

"Trait inheritance"—*a type can only be* `Eq` *if it is also* `PartialEq`.

What's this?

"Marker trait"—*note to the compiler that says this type has the properties associated with* `Eq`

Common Trait Speedrun: Copy

```rust
trait Copy: Clone { }

impl Copy for u32 {}

fn main() {
    let x: u32 = 1;
    let y = x;
    println!("{} and {}", x, y);
}
```

Copy is for types that can be trivially copied

Common Trait Speedrun: `ToString`

```
trait ToString {
    fn to_string(&self) -> String;
}
```

`ToString` types can be converted to a string

```
impl<T: fmt::Display> ToString for T {
   fn to_string(&self) -> String {
       ...
   }
}
```

What's this?

"Blanket Implementation"—*if T implements `Display` then T implements `ToString`*

Takeaways so far

We can use built-in operators on custom types
by implementing certain built-in traits

Next up: using built-in arithmetic operators

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);

fn main() {
    let a = Vec3(1.0, 2.0, 3.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use `*` operator on custom `Vec3` type for element-wise product.
Challenge: how to define `Mul` trait?

```
trait Mul {

    fn mul(self, other: &Self) -> Self;
}
impl Mul for Vec3 {

    fn mul(self, other: &Self) -> Self {
        Vec3(self.0 * other.0,
             self.1 * other.1,
             self.2 * other.2)
    }
}
```

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use `*` operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul {

    fn mul(self, other: &Self) -> Self;
}
impl Mul for Vec3 {

    fn mul(self, other: &Self) -> Self {
        Vec3(self.0 * other.0,
             self.1 * other.1,
             self.2 * other.2)
    }
}
```

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use `*` operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul {

    fn mul(self, other: &Self) -> Self;
}
impl Mul for Scalar {

    fn mul(self, other: &Vec3) -> Self {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
```

```
error[E0053]: method `mul` has an incompatible type for trait
 --> scratch.rs:24:25
   |
24 |     fn mul(self, other: &Vec3) -> Self {
   |                         ^^^^^ expected `Scalar`, found `Vec3`
   |
```

```
}
```

⚠️Not generic enough!⚠️

```
trait Mul {
    fn mul(self, other: &Self) -> Self;
}

impl Mul for Scalar {
    fn mul(self, other: &Vec3) -> Self {
        Vec3(self * other.0,
             self * other.1,
             self * other.2)
    }
}
```

Goal: use `*` operator on `Scalar` and `Vec3`.

Challenge: how to define `Mul` trait?

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use `*` operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul {

    fn mul(self, other: &Self) -> Self;
}
impl Mul for Scalar {

    fn mul(self, other: &Vec3) -> Self {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

Goal: use `*` operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul<Rhs> {

    fn mul(self, other: &Rhs) -> Self;
}
impl Mul<Vec3> for Scalar {

    fn mul(self, other: &Vec3) -> Self {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Case Study: Multiplication

⚠️Not always true!⚠️

```rust
struct Vec3(f32, f32, f32);
struct Scalar(f32);
fn main() {
    let a = Scalar(2.0);
    let b = Vec3(4.0, 5.0, 6.0);
    let c = a * b;
}
```

```rust
trait Mul<Rhs> {

    fn mul(self, other: &Rhs) -> Self;
}
impl Mul<Vec3> for Scalar {

    fn mul(self, other: &Vec3) -> Self {
                * other.0,
                * other.1,
                * other.2)
```

```
error[E0308]: mismatched types
--> scratch.rs:3:9
  |
2 |         fn mul(self, other: &Vec3) -> Self {
  |                                       ---- expected `Scalar`
  |
3 | /           Vec3(self   * other.0,
4 | |                self   * other.1,
5 | |                self   * other.2)
  | |_______________________________^ expected `Scalar`, found `Vec3`
```

Case Study: Multiplication

```
struct Vec3(f32, f32, f32);
s
f



}
```

“Associated Type”—type associated with a specific trait impl

Goal: use `*` operator on `Scalar` and `Vec3`.
Challenge: how to define `Mul` trait?

```
trait Mul<Rhs> {
    type Output;
    fn mul(self, other: &Rhs) -> Output;
}

impl Mul<Vec3> for Scalar {
    type Output = Vec3;
    fn mul(self, other: &Vec3) -> Output {
        Vec3(self   * other.0,
             self   * other.1,
             self   * other.2)
    }
}
```

Case Study: Multiplication

Final Implementation—fully generic

- Arbitrary left type, right type, and output type

```
trait Mul<Rhs> {
    type Output;
    fn mul(self, other: &Rhs) -> Output;
}
impl Mul<Vec3> for Scalar {
    type Output = Vec3;
    fn mul(self, other: &Vec3) -> Output {
        Vec3(self  * other.0,
             self  * other.1,
             self  * other.2)
    }
}
```

Common Trait Speedrun: `From`

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

`From` is for type to type conversions

```rust
impl From<A> for B {
    ...
}
```

If you see an impl like this, read it as “B can be made from A”

```rust
impl From<&String> for String {
    fn from(value: &String) -> String {
        ...
    }
}
```

An example implementation. How can we make a `String` given an `&String`?

Common Trait Speedrun: `Into`

```
trait Into<T> {
    fn into(self) -> T;
}

impl Into<String> for &String {
    fn into(self) -> String {
        ...
    }
}
// In standard library:
impl<T, U: From<T>> Into<U> for T {
    fn into(self) -> U {
        U::from(self)
    }
}
```

`Into` is the opposite of `From`

Can implement very similarly to `From`

- Seems sort of redundant...

Never need to implement `Into`, just implement `From` and `Into` will be implemented automatically. `Into` is more of a convenience.

Common Trait Speedrun: Using `Into`

```rust
trait Into<T> {
    fn into(self) -> T;
}
```

`Into` is the opposite of `From`

Sometimes type mismatches can be frustrating

```rust
fn write_to_file(data: Vec<u8>) {
    ...
}


fn main() {
    write_to_file("Hello");
}
```

```
error[E0308]: mismatched types
--> scratch.rs:6:19
    |
6   | write to file("Hello");
    | ------------- ^^^^^^^ expected `Vec<u8>`, found `&str`
    | |
    | arguments to this function are incorrect
```

Common Trait Speedrun: Using `Into`

```
trait Into<T> {
    fn into(self) -> T;
}
```

```
fn write_to_file(data: Vec<u8>) {
    ...
}

fn main() {
    write_to_file("Hello".into());
}
```

`Into` is the opposite of `From`

Sometimes type mismatches can be frustrating

✅

Common Trait Speedrun: Using `Into`

```
trait Into<T> {
    fn into(self) -> T;
}
```

`Into` is the opposite of `From`

```
fn write_to_file<T: Into<Vec<u8>>>(data: T) {
  let data = data.into();
  ...
}

fn main() {
   write_to_file("Hello");
   write_to_file(vec![3, 4, 5, 6, 7]);
}
```

If you control the API, you can also use this neat little trick.

Rust doesn't support overloaded functions, so this is as close as you can get.

Probably best to use this sparingly.

✅

Common Trait Speedrun: Fn

```
pub fn map<T, U, F: Fn(T) -> U>(list: &[T], f: F) -> &[U];
```

Fn: trait and special syntax for declaring function types

Common Trait Reference

`Default` types have a default value

`Clone` types can be deep copied

`Copy` types can be cloned by a bit-wise copy

`PartialEq (PartialOrd)` types can be compared with a partial equality (order) relation

`Eq (Ord)` types can be compared with an equality (total order) relation

`ToString` types can be converted to a string

`Debug` types can be converted to a developer-facing string representation

`Display` types can be converted to a user-facing string representation

`Add<T>`, `Mul<T>`, `Sub<T>`, `Div<T>` types can be summed/producted/differenced/divided with a value of type `T`

`From<T>` types can be created from a value of type `T`

`Into<T>` types can be converted to a value of type `T`

`Fn(T, U,...) -> V` types can be called with the corresponding parameters and return type

e.g. https://doc.rust-lang.org/std/convert/trait.Into.html

Deriving Traits

```rust
#[derive(Debug)]
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{:?}", id);
}
```

Recall from last time... but now we understand!

`#[derive(...)]` syntax invokes a *macro*, a function that takes the code for your struct (or enum) as input and produces more code as output. In this case, a trait implementation.

Deriving Traits

```
#[derive(Debug, Clone, Copy,
PartialEq)]
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{:?}", id);
}
```

Recall from last time… but now we understand!

`#[derive(...)]` syntax invokes a *macro*, a function that takes the code for your struct (or enum) as input and produces more code as output. In this case, a trait implementation.

How are Traits Implemented?

```
fn write_to_file(data: Vec<u8>) {
    ...
}

fn main() {
    write_to_file("Hello".into());
}
```

Just like generics, no runtime cost

- compiler statically determines which function to call based on type inference

Dowsides of traits: hard to read docs?

Function std::fs::write 1.26.0 · source · [-]

```
pub fn write<P: AsRef<Path>, C: AsRef<[u8]>>(path: P, contents: C) -> Result<()>
```

Upside of traits: high level reasoning

```
pub fn sort_by_key<K, F>(&mut self, f: F)
where
    F: FnMut(&T) -> K,
    K: Ord,
```

Determine the types before writing the implementation

Overview of traits

Functions, structs, and enums can be generic.

To restrict the types that the generic can be instantiated with, use traits

Traits define a set of methods a type must implement

Trait Inheritance: some traits can only be implemented if another trait is implemented as well

Marker traits: traits with no methods

Provided methods: methods that are automatically defined in terms of other trait methods

Generic traits: makes a trait generic over a type

Associated types: types that implement this trait must also specify what the associated type is

Further reading

Common Rust Traits

The Rust Book 10.2

Rust By Example

CS242 Notes