Generics or Traits? How about Both?

Andreea Costea

Delft University of Technology

2025-09-16

TU Delft

Types

So far:

Struct: this *and* that.

Enum: this *or* that (with help from **Pattern Matching**).

Types

So far:

Struct: this *and* that.

Enum: this *or* that (with help from **Pattern Matching**).

Today:

Generics: *shared code* for many kinds of types.

Traits: *contract obligation* (interface).

Type Bounds: generics constrained by traits.

Errors: a use case using all of the above.

Generics

A simple example: in-place reversal of an array

```Rust
fn reverse(slice: &mut [i32]) {
  if slice.len() > 0 {
    let mut i = 0;
    let mut j = slice.len() - 1;
    while i < j {
      slice.swap(i, j);
      i += 1;
      j -= 1;
    } } }
```

A simple example: in-place reversal of an array

```Rust
fn reverse(slice: &mut [i32]) {
/*  if slice.len() > 0 {
    let mut i = 0;
    let mut j = slice.len() - 1;
    while i < j {
      slice.swap(i, j);
      i += 1;
      j -= 1;
    } } */ }
```

A simple example: in-place reversal of an array

```rust
fn reverse(slice: &mut [i32]) {
/*  if slice.len() > 0 {
    let mut i = 0;
    let mut j = slice.len() - 1;
    while i < j {
      slice.swap(i, j);
      i += 1;
      j -= 1;
    } } */ }
```

What if we also want a `reverse` for `&mut [char]`?

A simple example: in-place reversal of an array

```Rust
fn reverse_char(slice: &mut [char]) {
/*  if slice.len() > 0 {
    let mut i = 0;
    let mut j = slice.len() - 1;
    while i < j {
      slice.swap(i, j);
      i += 1;
      j -= 1;
    } } */ }
```

A simple example: in-place reversal of an array

```Rust
fn reverse_i32(slice: &mut [i32]) {
  /* some lengthy code */
}

fn reverse_char(slice: &mut [char]) {
  /* the same lengthy code as above */
}
```

A simple example: in-place reversal of an array

```Rust
fn reverse_i32(slice: &mut [i32]) {
  /* some lengthy code */
}

fn reverse_char(slice: &mut [char]) {
  /* the same lengthy code as above */
}
```

That's a lot of duplicated code already!

Can we do better?

Solution: *generic* in-place reversal of an array

```Rust
//          v--- for any type T
fn reverse<T>(slice: &mut [T]) {
//                          ^--- a mut ref to an array of that type T
/*  if slice.len() > 0 {
        let mut i = 0;
        let mut j = slice.len() - 1;
        while i < j {
            slice.swap(i, j);
            i += 1;
            j -= 1;
        } } */ }
```

A simple example: client code

```rust
fn reverse<T>(slice: &mut [T]) { /* some lengthy code */ }

let mut numbers = [1, 2, 3, 4];
let mut letters = ['a', 'b', 'c'];
/*
  reverse(&mut numbers);          // results in: [4, 3, 2, 1];
  reverse(&mut letters);          // ['c', 'b', 'a'];
  reverse(&mut [1.0, 1.5, 2.0]);  // [2.0, 1.5, 1.0]. */
```

A simple example: client code

```rust
fn reverse<T>(slice: &mut [T]) { /* some lengthy code */ }

let mut numbers = [1, 2, 3, 4];
let mut letters = ['a', 'b', 'c'];

reverse(&mut numbers);          // results in: [4, 3, 2, 1];
reverse(&mut letters);          // ['c', 'b', 'a'];
reverse(&mut [1.0, 1.5, 2.0]);  // [2.0, 1.5, 1.0]
```

In C++ this is often called a “template”.

Generics

- A generic is a placeholder for a type.
- Functions with a *generic type parameter* are *polymorphic*.
- **Monomorphization**: at compile time, each concrete type generates its own concrete function.

https://godbolt.org/z/Mz71shEr4 https://godbolt.org/z/e1fK3a3hf

```Rust
fn reverse<T>(slice: &mut [T]) { /* some lengthy code */ }

// monomorphizes once for `&mut [i32]`
reverse(&mut numbers);
// and once for `&mut [char]`
reverse(&mut letters);
```

Is this how we design our types in real-world code?

```Rust
fn reverse<T>(slice: &mut [T]) { /* some lengthy code */ }

let mut numbers = [1, 2, 3, 4];
let mut letters = ['a', 'b', 'c'];

reverse(&mut numbers);
reverse(&mut letters);
```

Is this how we design our types in real-world code?

```rust
fn reverse<T>(slice: &mut [T]) { /* some lengthy code */ }

let mut numbers = [1, 2, 3, 4];
let mut letters = ['a', 'b', 'c'];

reverse(&mut numbers);
reverse(&mut letters);
```

In real-world code these simple arrays are generally part of more complex types.

Is this how we design our types in real-world code?

We normally wrap our simple types in **structs**:

```Rust
struct Packet{
    /* other fields */
    data: [i32; 4],
}
```

Is this how we design our types in real-world code?

We normally wrap our simple types in **structs**:

```rust
struct Packet{
  /* other fields */
  data: [i32; 4],
}
```

Generic types work for structs too:

```rust
struct Packet<T>{
  /* other fields */
  data: [T; 4],          /* The struct holds a data array of type T */
}
```

Is this how we design our types in real-world code?

We normally wrap our simple types in **structs**:

```rust
struct Packet{
  /* other fields */
  data: [i32; 4],
}
```

Generic types and **generic consts** work for structs too:

```rust
struct Packet<T, const N: usize>{
  /* other fields */
  data: [T; N],                /* Struct holds N elements of type T. */
}
```

Generic types and consts

```rust
struct Packet<T, const N: usize>{
  /* other fields */
  data: [T; N],
}
```

Create type aliases for the kind of `Packet` structs I may need:

```rust
type NumPacket  = Packet<i32, 4>;
type CharPacket = Packet<char, 3>;

let mut numbers = NumPacket::new([1, 2, 3, 4]);
let mut letters = CharPacket::new(['a', 'b', 'c']);
```

Generic types and consts

```Rust
struct Packet<T, const N: usize>{
  /* other fields */
  data: [T; N],
}
```

Implement reverse for all `Packet` structs:

```Rust
impl <T, const N: usize> Packet<T,N> {

   fn reverse(slice: &mut Self) { /* some lengthy code */  }
}
```

Generic types and consts

Implement reverse for all `Packet` structs:

```rust
impl <T, const N: usize> Packet<T,N> {

    fn reverse(slice: &mut Self) {
        if slice.data.len() > 0 {
            /* some lenghty code */
        } } }
```

Generic types and consts

Implement reverse for all `Packet` structs:

```rust
impl <T, const N: usize> Packet<T,N> {

    fn reverse(slice: &mut Self) {
        if N > 0 {                          // the length is actually a const
            /* some lenghty code */
        } } }
```

Generic types and consts

Implement reverse for all `Packet` structs:

```Rust
impl <T, const N: usize> Packet<T,N> {

    fn reverse(slice: &mut Self) {
        if N > 0 {                          // the length is actually a const
            /* some lenghty code */
        } } }
```

Delegate part of your program logic to the type system (**zero cost abstraction**):

- during monomorphization `N` will be replaced by an actual constant, thus
- the `if` statement dissapears completely in the concrete `reverse` functions.

Generic types and consts

Use a **const generic** when:

- The value is known at compile time.
- You want zero runtime overhead (the value affects the type itself).

Generic types and consts

Use a **const generic** when:

- The value is known at compile time.
- You want zero runtime overhead (the value affects the type itself).

```Rust
struct Packet<T, const N: usize, const DEDUPLICATE: bool>{
  /* other fields */
  data: [T; N],
}
```

Generic types and consts

Use a **const generic** when:

- The value is known at compile time.
- You want zero runtime overhead (the value affects the type itself).

```rust
struct Packet<T, const N: usize, const DEDUPLICATE: bool>{
    /* other fields */
    data: [T; N],
}
```

```rust
struct Packet<T, const N: usize, const MAX_SIZE: usize>{
    /* other fields */
    data: [T; N],
}
```

Generic types and consts

Use a **const generic** when:

- The value is known at compile time.
- You want zero runtime overhead (the value affects the type itself).

Use struct **fields as flags** when:

- The value may vary at runtime between instances.
- You want dynamic flexibility without making the type dependent on it.

Generics

- **Polymorphic functions**: make a function work for any type.

```Rust
fn reverse<T>(slice: &mut [T]);
```

```Rust
fn get_first<'a>(v: &'a Vec<i32>, x:  &'a i32) -> &'a i32
```

Generics

- **Polymorphic functions**: make a function work for any type.
- **Generic structs**: parameterize the type of fields.

```Rust
struct Packet<T, const N: usize>{
    /* other fields */
    data: [T; N],
}
```

Generics

- **Polymorphic functions**: make a function work for any type.
- **Generic structs**: parameterize the type of fields.
- **Generic enums**: can hold different types generically.

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

Generics

- **Polymorphic functions**: make a function work for any type.
- **Generic structs**: parameterize the type of fields.
- **Generic enums**: can hold different types generically.
- **Impl blocks**: implement methods for a generic struct or enum.

```rust
impl <T, const N: usize> Packet<T,N> { /*  */ }
```

Traits

Traits

A trait defines a behavior that selected types must implement (like interfaces).

- *type contracts*—obligation to implement a behaviour (properties of types).

Traits

A trait defines a behavior that selected types must implement (like interfaces).

- *type contracts*—obligation to implement a behaviour (properties of types).
- A trait is a behavior that a group of types *share*:
  - `Copy`: for scalar types;
  - `Clone`: deep copy;
  - `Drop`: to free memory.

  ...

Traits

A trait defines a behavior that selected types must implement (like interfaces).

- *type contracts*—obligation to implement a behaviour (properties of types).
- A trait is a behavior that a group of types *share*:
  - `Copy`: for scalar types;
  - `Clone`: deep copy;
  - `Drop`: to free memory.

  ...

```rust
trait Clone {
    fn clone(&self) -> Self;  /* a type will implement this behaviour */
}
```

A Simple Trait Example

```Rust
struct Id(u32);
struct Point { x: f64, y: f64, }
```

A Simple Trait Example

```Rust
struct Id(u32);
struct Point { x: f64, y: f64, }


// a function that prints any data structure in my project
fn print_string(item: & ???) {

    println!("Item: {}", item ??? );

}
```

A Simple Trait Example: add a "stringable" behaviour

```rust
struct Id(u32);
struct Point { x: f64, y: f64, }

// A trait the imposes the `to_string` behavior
trait ToString { fn to_string(&self) -> String; }

// a function that prints any data structure in my project
fn print_string(item: & ???) {

    println!("Item: {}", item ??? );

}
```

Solution: add a "stringable" behaviour (part-1)

```rust
struct Id(u32);
struct Point { x: f64, y: f64, }

// A trait the imposes the `to_string` behavior
trait ToString { fn to_string(&self) -> String; }

// a function that prints any data structure in my project
fn print_string(item: &impl ToString) {
//                        ^--- a type implementing the ToString trait
    println!("Item: {}", item.to_string() );
//                             ^--- invoking the method on any item
}
```

Solution: implement the trait (part-2)

```Rust
// A trait the imposes the `to_string` behavior
trait ToString { fn to_string(&self) -> String; }

// Implement the trait
impl ToString for ID {
    fn to_string(&self) -> String {
        format!("id_x{}", self.0)                    // id_x0
    } }
impl ToString for Point {
    fn to_string(&self) -> String {
        format!("Point({}, {})", self.x, self.y)  // Point(1.0, 2.0)
    } }
```

A Simple Trait Example: client code

```Rust
fn print_string(item: &impl ToString) {
    println!("Item: {}", item.to_string());
}

let n = ID(0);
let p = Point { x: 1.0, y: 2.0 };

print_string(&n);           // prints: id_x0
print_string(&p);           // prints: Point(1.0, 2.0)
```

Traits vs Generics

- think of **traits** as an ***obligation to implement*** *a type behaviour*:
  - you know that any type implementing a certain trait will have this behaviour,
  - but, the developer implements the behaviour for each type (**specialisation**).
  - the behaviour could implement a *different logic* for each type.

- think of **generics** as types sharing the *same behaviour* ***implementation***:
  - implemented behaviour shared across multiple similar types.
  - specialisation happens at compile time (monomorphization).
  - the behaviour implements *the same logic* for each type.

Traits vs Generics

- think of **traits** as an ***obligation to implement*** *a type behaviour*:
  - you know that any type implementing a certain trait will have this behaviour,
  - but, the developer implements the behaviour for each type (**specialisation**).
  - the behaviour could implement a *different logic* for each type.

- think of **generics** as types sharing the *same behaviour* ***implementation***:
  - implemented behaviour shared across multiple similar types.
  - specialisation happens at compile time (monomorphization).
  - the behaviour implements *the same logic* for each type.

Traits vs Generics

- think of **traits** as an ***obligation to implement*** *a type behaviour*:
  - you know that any type implementing a certain trait will have this behaviour,
  - but, the developer implements the behaviour for each type (**specialisation**).
  - the behaviour could implement a *different logic* for each type.

- think of **generics** as types sharing the *same behaviour* ***implementation***:
  - implemented behaviour shared across multiple similar types.
  - specialisation happens at compile time (monomorphization).
  - the behaviour implements *the same logic* for each type.

Type Bounds

```rust
struct Packet<T, const N: usize>{
  data: [T; N],
}
```

```rust
struct Packet<T, const N: usize>{
  data: [T; N],
}

//       v--- the generic type T must implement ToString
impl <T: ToString, const N: usize> Packet<T,N> {...}
```

```rust
struct Packet<T, const N: usize>{
  data: [T; N],
}

//       v--- the generic type T must implement ToString
impl <T: ToString, const N: usize> Packet<T,N> {

 fn print(&self) {
     for item in &self.data {
       print!("{} ", item.to_string());
       //                ^--- rely on the fact that T implements it
     }
 }
}
```

```rust
struct Packet<T, const N: usize>{
  data: [T; N],
}

impl <T: ToString, const N: usize> Packet<T,N> {
 fn print(&self) { /* ... */ }
}

let mut numbers = NumPacket::new([1, 2, 3, 4]);
let mut letters = CharPacket::new(['a', 'b', 'c']);

numbers.print();
letters.print();
```

Type Bounds

So far, it feels like all we've done is find another way to do something we already knew: print!

Well, yes, because whenever we did print there was a combo of pre-defined generics and traits working out the print for us.

Type Bounds

So far, it feels like all we've done is find another way to do something we already knew: print!

Well, yes, because whenever we did print there was a combo of pre-defined generics and traits working out the print for us.

```
1 println!("{}", 3) (generic macro)
2 └─ format_args!("{}", 3)
3       └─ trait bound : Display (implements ToString trait)
4             └─ <i32 as Display>::fmt(&3, f)
5                   └─ write!(f, "3") (monomorphized)
```

Type Bounds

So far, it feels like all we've done is find another way to do something we already knew: print!

Well, yes, because whenever we did print there was a combo of pre-defined generics and traits working out the print for us.

```
1  println!("{?:}", [1,2,3]]);
```

Courtesy of `#[derive(Debug)]`

- automatically implements the `Debug` trait—similar to `ToString`.

Deriving

```Rust
#[derive(
    PartialEq, Eq,      // equality checking
    PartialOrd, Ord,    // ordering (a > b)
    Copy, Clone,        // copying, cloning
    Hash,               // hashing for `HashMap`
    Debug,              // debug printing
    Default             // automatically provides a “default value”
)]
pub struct Vector {
  x: f64,
  y: f64,
}
```

Survey

Rust Topics

So far:

- **Struct**: this *and* that.
- **Enum**: this *or* that (with help from **Pattern Matching**).
- **Generics**: *shared code* for many kinds of types.
- **Traits**: *shared contract obligation* (interface).
- **Type Bounds**: generics constrained by traits.
- (*today*) **Errors**: a use case using all of the above.

Rust Topics

So far:

- **Struct**: this *and* that.
- **Enum**: this *or* that (with help from **Pattern Matching**).
- **Generics**: *shared code* for many kinds of types.
- **Traits**: *shared contract obligation* (interface).
- **Type Bounds**: generics constrained by traits.
- (*today*) **Errors**: a use case using all of the above.

Next Lectures:

- **Iterators**: tools for processing sequences of values.
- **Box**: smart pointer for heap-allocating data.
- **Testing**: how to check that your code behaves as expected?
- **Crate Engineering**: how to think about and design your software.

Is there any Rust topic you would like us to revisit or is there a topic we do not cover but you are interested in?

Important Traits: Conversion

```Rust
pub trait Into<T>: Sized {
    fn into(self) -> T;
}

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

The `Sized` trait indicates that the size of a type is known at compile time.

https://doc.rust-lang.org/stable/std/convert/trait.From.html

Important Traits: Dereferencing

```rust
pub trait Deref {
    type Target: ?Sized;

    fn deref(&self) -> &Self::Target;
}
```

- `Box` is just a struct.
- But because it implements `Deref` it can be used like a pointer!

https://doc.rust-lang.org/stable/std/ops/trait.Deref.html

Important Traits: Drop

- Determines what to do when a value goes out of scope.
- Recursively called.
- This is how `Vec` frees its elements.

```rust
pub trait Drop {
    fn drop(&mut self);
}
```

https://doc.rust-lang.org/stable/std/ops/trait.Drop.html

Vec:

- https://doc.rust-lang.org/1.81.0/src/alloc/vec/mod.rs.html#398
- https://doc.rust-lang.org/1.81.0/src/alloc/raw_vec.rs.html#596
- https://doc.rust-lang.org/1.81.0/src/alloc/vec/mod.rs.html#3302

Important Traits: Formatting

```Rust
pub trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

https://doc.rust-lang.org/stable/std/fmt/trait.Display.html

Errors

Error Handling

Another `enum` in the standard library:

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

It's a bit like `Option` if `None` were to wrap a value.

Error Handling

Another `enum` in the standard library:

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

You'll often see functions like this:

```rust
fn do_thing(some_input: X) -> Result<Y, Error> {
  // if it goes well: Ok
  // if it goes bad: Err
}
```

Common Error Handling Pattern

```Rust
fn do_big_thing() -> Result<Output, Error> {
  let outcome1 = match do_small_thing1() {
    Ok(i) => i,
    Err(e) => return Err(e),
  };
  let outcome2 = match do_small_thing2(outcome1) {
    Ok(i) => i,
    Err(e) => return Err(e),
  };
  Ok(outcome2)
}
```

Common Error Handling Pattern

```Rust
fn do_big_thing() -> Result<Output, Error> {
    let outcome1 = match do_small_thing1() { Ok(i) => i, Err(e) => return Err(e)};
    let outcome2 = match do_small_thing2(outcome1) { Ok(i) => i, Err(e) => return Err(e)};
    let outcome3 = match do_small_thing3(outcome2) { Ok(i) => i, Err(e) => return Err(e)};
    let outcome4 = match do_small_thing4(outcome3) { Ok(i) => i, Err(e) => return Err(e)};
    let outcome5 = match do_small_thing5(outcome4) { Ok(i) => i, Err(e) => return Err(e)};
    Ok(outcome5) }
```

Common Error Handling Pattern

So much boilerplate!

Common Error Handling Pattern

Use the `?` macro to do exactly the same:

```Rust
let outcome1 = do_small_thing1()?;
```

means

```Rust
let outcome1 = match do_small_thing1() {
    Ok(i) => i,
    Err(e) => return Err(e)
};
```

Common Error Handling Pattern

So the big example from before becomes:

```rust
fn do_big_thing() -> Result<Output, Error> {
    let outcome1 = do_small_thing1()?;
    let outcome2 = do_small_thing2(outcome1)?;
    let outcome3 = do_small_thing3(outcome2)?;
    let outcome4 = do_small_thing4(outcome3)?;
    let outcome5 = do_small_thing5(outcome4)?;

    Ok(outcome5)
}
```

Assignment: 10 minutes

Implement a function with the following signature:

Template:

```rust
fn from_file(filename: &Path) -> Result<i32, MyError>;
```

Define `MyError` to account for at least 2 possible errors that `from_file` should handle.

Use `std::path::Path` for access to `Path`.

Error handling in real life:

```Rust
pub enum MyError {
    Io(io::Error),
    Json(serde_json::Error),
    Yaml(serde_yaml::Error),
    Plist(plist::Error),
}

fn from_file(path: &Path) -> Result<Something, MyError>;
```

- This is an error for a single function `from_file`.
- It can fail in 4 different major ways.
- A different function might have different ways to fail.

Error handling in real life:

```Rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum MyError {
    Io(io::Error), // ...
}
```

Error handling in real life:

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum MyError {
    Io(io::Error), // ...
}
```

`#[derive(Error, Debug)]`

- Error: generates an implementation of `std::error::Error` for your enum.
- Debug: lets you print the enum with the `{:?}` formatter for debugging.

Together, these two macros save you from manually implementing the boilerplate needed to make your enum a proper error type.

Error handling in real life: (cont.)

```Rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, MyError> {
  let contents = read_file()?; // does this work? No!


}
```

Error handling in real life:

```rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, MyError> {
    let contents = match read_file() {
        Ok(i) => i,
        //  v--- io::Error
        Err(e) => return Err(e),
        //                  ^--- expects MyError
    }
}
```

Error handling in real life:

```rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, MyError> {
    let contents = match read_file() {
        Ok(i) => i,
        //  v-- io::Error
        Err(e) => return Err(MyError::Io(e));    // wrap it into MyError
        //  but then we cannot use the `?` macro!
    }
}
```

Error handling in real life:

```rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, MyError> {
    let contents = match read_file() {
        Ok(i) => i,
        //  v-- io::Error      vvvv-- convert io::Error to MyError
        Err(e) => return Err(e.into()),        // Implement `From` trait!
        //           MyError--^^^^^^^^
    }
}
```

Error handling in real life:

```Rust
#[derive(Error, Debug)]
pub enum MyError {
    Io(io::Error), // ...
}

impl From<io::Error> for MyError {
  fn from(e: io::Error) -> Self { MyError::Io(e) }
}
```

Implement the `From` trait (the `Into` trait is derived from the implementation of the `From` trait).

Error handling in real life:

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum MyError {
    Io(#[from] io::Error), // ...
}
```

Or, use the macro `#[from]` to automatically implement `From <io::Error>` for `MyError`.

Error handling in real life:

```Rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, MyError> {
  // this works! automatic conversion from io::Error to MyError
  let contents = read_file()?;



}
```

Assignment: 10 minutes

Modify your earlier code such that it makes at least one call to a pre-defined method which returns `Result` with a pre-defined error type.

For example, you could use `read_to_string` from `std::fs`, and the `io::Error`:

Template:

```Rust
use std::fs;
/* signature */
fn read_to_string<P:AsRef<Path>>(path:P)->Result<String,io::Error>;
/* function invocation example ( filename has type `Path`): */
let res = fs::read_to_string(&filename);
```

Dynamic Dispatch

What if we could pass a function pointer that we just call at the right time?

```Rust
fn print_thing(print_fn: fn()) {
  print!("the thing is");
  print_fn(); println!()
}
fn print_3() { print!("3") }
// ...
print_thing(print_3);
print_thing(print_3u64);
print_thing(print_3f64);
print_thing(print_hi);
```

Dynamic Dispatch

Uses a vtable (runtime type description) to create a “trait object”

```rust
fn print_thing(x: &dyn Display) {
  println!("the thing is {i}");
}

print_thing(&3);
print_thing(&3u64);
print_thing(&3.0);
print_thing(&"hi");
```

https://godbolt.org/z/hcqGdW8z8

Dynamic Dispatch

Advatages

- *Reduced compiled code size*: One vtable per trait object type, instead of generating a copy of the function for every concrete type (no code bloat).
- *Runtime flexibility*: can store heterogeneous types that implement the same trait (e.g. `dyn Display`).

Disadvatages

- *Runtime overhead*: each method call goes through a vtable lookup (extra indirection).
- *No inlining across the boundary*: The compiler can't inline or optimize based on the concrete type, losing some performance opportunities.

Monomorphization vs Dynamic Dispatch

Monomorphization:

- faster, more optimized, but bigger binaries.

Dynamic dispatch:

- smaller binaries, more flexible, but slower calls.

Casting - revisit

- No numeric conversions;
- No user-defined implicit casts: traits like `From` and `Into` exist, but they are explicit (`.into()`, `T::from(x)`), not automatic.

Casting - revisit

- No numeric conversions;
- No user-defined implicit casts: traits like `From` and `Into` exist, but they are explicit (`.into()`, `T::from(x)`), not automatic.

Automatic casting:

- Reference adjustments:
  - `&mut T -> &T`
  - `&T -> *const T`
  - `&mut T -> *mut T`

Casting - revisit

- No numeric conversions;
- No user-defined implicit casts: traits like `From` and `Into` exist, but they are explicit (`.into()`, `T::from(x)`), not automatic.

Automatic casting:

- Reference adjustments:
- Unsized coercions:
  - `&[T; N]` -> `&[T]` (array to slice)
  - `&str` -> `&dyn AsRef<str>`
  - `Box<[T; N]>` -> `Box<[T]`

Casting - revisit

- No numeric conversions;
- No user-defined implicit casts: traits like `From` and `Into` exist, but they are explicit (`.into()`, `T::from(x)`), not automatic.

Automatic casting:

- Reference adjustments:
- Unsized coercions:
- Trait object coercions
  - `&T -> &dyn Trait` if `T: Trait`
  - `Box<T> -> Box<dyn Trait>`
  - `Rc<T> -> Rc<dyn Trait>`

Casting - revisit

- No numeric conversions;
- No user-defined implicit casts: traits like `From` and `Into` exist, but they are explicit (`.into()`, `T::from(x)`), not automatic.

Automatic casting:

- Reference adjustments
- Unsized coercions
- Trait object coercions
- Function pointer adjustments
  - `` `fn(T) -> U` –> `fn(T) -> U `` with fewer lifetime or mutability requirements`.
  - Example: `&'static str` can be coerced to `&str` in some contexts.

Casting - revisit

- No numeric conversions;
- No user-defined implicit casts: traits like `From` and `Into` exist, but they are explicit (`.into()`, `T::from(x)`), not automatic.

Automatic casting:

- Reference adjustments
- Unsized coercions
- Trait object coercions
- Function pointer adjustments
- Closure to function pointer
  - A non-capturing closure like `|x| x + 1` can be coerced to a function pointer `fn(i32) -> i32`.

Summary

- Generics create patterns/templates of items.
- They can be instantiated with concrete types.
- (Generic) Types used can be constrained by traits.

Next week: the `Iterator` trait!

Repo with the exercises we tried in the class

Clone this repo for access to the exercises in this slide:

git@github.com:andrecostea/CESE4000.git