

UNDERSTANDING MIR

How rust behaves: source code is compiled into machine code. Pipeline of rust has stages and several translations: " \rightarrow " not necessarily a single step (lex + parse).

Source code \rightarrow AST representation (we remove irrelevant sugarcoated part) \rightarrow High level Intermediate Repres.: it has
Sub to resolve names (at runtime we
solve name) + desugared \rightarrow
Mid Level IR: basically R in terms of
CFG with explicit borrows and lifetimes for
checking \rightarrow
 \rightarrow IR of the LLVM framework
Rust Source Code \rightarrow AST \rightarrow HIR \rightarrow MIR \rightarrow LLVM IR \rightarrow Machine Code

The Compilation Pipeline

Rust Source Code \rightarrow AST \rightarrow HIR \rightarrow MIR \rightarrow LLVM IR \rightarrow Machine Code

where:

- **AST** (Abstract Syntax Tree): Direct representation of source code syntax

- **HIR** (High-level IR): Desugared and name-resolved representation

- **MIR** (Mid-level IR): Control flow graph with explicit borrows and lifetimes

- **LLVM IR**: Low-level representation for the LLVM optimizer and code generator

(meta framework for compilation).

A lot of compilers are built on

LLVM FM. \rightarrow Machine Code

The Mid-level Intermediate Representation (**MIR**) is the part of Rust's compilation pipeline that bridges the gap between Rust's high-level syntax and LLVM IR (the lower-level representation)

What is MIR?

MIR is a CFG representation, but this CFG contains types, it is structured (it contains jumps related to High Level Representation)

↑ all information is where, no inference.

- It represents code in a **Control Flow Graph (CFG)** form.
- It is **typed**, **structured**, and **explicit**, designed for analysis, transformation, and verification.

Designers designed MIR to be the place where optimization and SA occur

Why MIR?

MIR was introduced to **decouple** Rust's **analysis** and **optimizations** from the complexities of its syntax and semantics.

MIR allows the compiler to perform sophisticated safety checks and optimizations at a higher level than LLVM IR.

MIR is well suited to manage optimization and SA specifically for Rust, this is why we work at this level

Why MIR?

MIR serves several key purposes:

1. **Borrow Checking**: MIR makes Rust's ownership and borrowing rules more explicit and easier to verify
2. **Optimization**: It enables Rust-specific optimizations before LLVM
3. **Analysis**: MIR simplifies static analysis for the compiler
4. **Compilation Speed**^①: It improves incremental compilation
5. **Debugging**: MIR provides a more structured representation for diagnostics

Hey Giovanni! So when it says "*MIR improves incremental compilation*", it's talking about how **MIR (Mid-level Intermediate Representation)** helps the Rust compiler get faster at compiling code—especially when you only change small parts of your project.

Let me break it down casually:

When you compile a Rust program, the compiler breaks it down into stages. One of these is MIR, which is a simplified version of your code that's easier for the compiler to analyze and optimize. Think of it as a neat, organized version of your messy room—it's not the final output, but it's way easier to work with.

Now, **incremental compilation** means: "*Don't redo everything from scratch when I only changed a little thing.*" Without MIR, the compiler would often have to reprocess more code than necessary. But thanks to MIR, it can keep track of changes more precisely and avoid wasting time recompiling parts of the code that didn't change.

Explicit Control Flow

Basic blocks: sequence of statements executed in order. At the end you have a terminator: Return, jump or goto.

MIR is structured as a **control flow graph (CFG)** of *basic blocks*:

Each block ends with a *terminator* (like a goto, return, or conditional jump).

This provides a strong support for flow analysis, borrow checking, dead code elimination, etc.

Locals

1st thing: names are slid. We don't have names at runtime, but still we have variables we can access. In IR we need some mech to represent variables. These vars are called locals, they are compiler internal variables that will be mangled to names.

In Rust MIR, **locals** are compiler-internal variables used to store function arguments, temporaries, intermediate values, and the return value. These are systematically named like _0, _1, _2, etc.

```
fn foo(x: i32, y: i32) -> i32 {  
    let z = x + y;  
    z * 2  
}
```

↑ we can infer type

```
fn foo(x: i32, y: i32) -> i32 {  
    ① let mut _0: i32;           // return value  
    let _1: i32;                 // argument x  
    let _2: i32;                 // argument y  
    let mut _3: i32;             // local variable z  
}
```

Vars represented by -number.

① -0 represents result of a function. In CFG we have special var result, different from all other variables.
Static single Assignment is satisfied.
All different.

② -1, -2 used to represent parameters; -1, -2 are x, y .

• Mutable for optimization

Then a special variable to represent z . -3.

purposes and compiler opt.

z has explicit type. Inference has been already performed

Statements

MIR statements are simple operations:

// Assignment
_2 = const 5_i32; We assign a constant 5 to _2.

// Function call
_4 = function_name(_2, _3);

Very close to rust

// Memory operations

_5 = &mut _2; // Take a mutable reference
• _6 = (*_5); // Dereference

MIR basic blocks

A **basic block** in MIR is a **classical** basic block

- A sequence of **statements** that always executes in order.
- Terminated by a **terminator** (like goto, return, switch, etc.).
- No internal branches; control flow can only enter at the top and exit at the terminator.



Basic Blocks

```
fn foo(x: i32, y: i32) -> i32 {  
    let mut _0: i32;           // return  
    let _1: i32;               // arg x  
    let _2: i32;               // arg y  
    let mut _3: i32;           // local z
```

} Declaration

```
bb0: {  
    _3 = Add(_1, _2);        // z = x + y  
    _0 = Mul(_3, const 2);   // return z * 2  
    return; ①  
}
```

bb0: the basic block:
every function starts with one.
It contains two statements and a terminator.

basic block has a name and label. Thus is associated to body

① Return always provides val stored in _0,

```
fn max(a: i32, b: i32) -> i32 {  
    let mut _0: i32;  
  
    bb0: {  
        switchInt(a > b) -> [false: bb2, otherwise: bb1];  
    }  
  
    bb1: {  
        _0 = a;  
        goto -> bb3;  
    }  
  
    bb2: {  
        _0 = b;  
        goto -> bb3;  
    }  
  
    bb3: {  
        return;  
    }  
}
```

suggested: a should be -1, -2.

basic block is

Terminators

Terminators control flow between basic blocks:

```
// Unconditional jump  
goto -> bb1;  
  
// Conditional branch  
switchInt(_1) -> [0: bb2, otherwise: bb3];  
  
// Return  
return;  
  
// Function call with possible unwinding  
call foo() -> [return: bb1, unwind: bb2];
```



In Rust (and other languages like C++), **unwinding** happens when a function *panics*—like, something went really wrong—and it starts cleaning up the stack to exit the current context.

So when the compiler or documentation says “**function call with possible unwinding**”, it means:

“This function might panic and start unwinding the stack.”

Places and Rvalues

MIR distinguishes between:

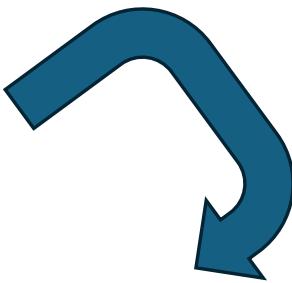
- **Places** (lvalues): Locations where values are stored (variables, dereferenced pointers)
- **Rvalues**: Computations producing values (literals, operations, function calls)

→ What you find on left hand side of assignment:
Some container of data, variables, memory loc.

↳ values, or something that when evaluated gives values.

```
// Place = Rvalue
_1 = Add(_2, _3); // _1 is a place, Add(_2, _3) is an rvalue
```

```
fn add(a: i32, b: i32) -> i32 {  
    let c = 5;  
    a + b + c  
}
```



```
fn add(_1: i32, _2: i32) -> i32 {  
  
    let mut _0: i32;           // Return place  
    let _3: i32;               // c  
    let mut _4: i32;           // Temporary for a + b  
  
    bb0: {  
        _3 = const 5_i32;      // Initialize c  
        _4 = Add(_1, _2);      // Compute a + b  
        _0 = Add(_4, _3);      // Compute (a + b) + c  
        return;                // Return _0  
    }  
}
```

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str { MIR and Lifetime

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
fn main() {
    let string1 = String::from("long string is long");
    let string2 = String::from("xyz");
    let result = longest(&string1, &string2);
    println!("The longest string is {}", result);
```

If we don't insert lifetime as type param. This program won't compile.

Here we say result of longest is a value that must live at least as 'a. And inputs will live at least 'a.

```
// Simplified MIR for the longest function
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    let _0: &'a str;                      // Return place
    let mut _1: bool;                     // Temporary for condition
    let mut _2: usize;                   // Temporary for x.len()
    let mut _3: usize;                   // Temporary for y.len()
```

• Simplified version

```
// Basic block 0
_2 = len(x);
_3 = len(y);
_1 = gt(_2, _3); // Compute x.len() > y.len()
switchInt(_1) -> [false: bb2, otherwise: bb1]; // Branch based on condition
```

check whether $\text{len}(x) > \text{len}(y)$

// Compute $x.\text{len}() > y.\text{len}()$

Depending on the branch value we jump to bb2/bb1.

```
// Basic block 1 (true branch)
_0 = x;                                // Set return value to x
goto -> bb3;                            // Jump to exit block

// Basic block 2 (false branch)
_0 = y;                                // Set return value to y
goto -> bb3;                            // Jump to exit block
```

```
// Basic block 3 (exit)
return; // Return control to caller with value in _0
}
```

How does the borrow checker behave here? What happens in MIR?

- Looks at function signature and understand scope and lifetime of parameters.

Borrow Checking with MIR (step 1)

Building the borrows in scope

The borrow checker first identifies all borrows in scope at each point in the function:

In the function signature, it registers that:

- Parameter x is borrowed with lifetime 'a' *x is borrowed with lifetime of at least 'a'*
- Parameter y is borrowed with lifetime 'a' *Same for y. So we have constraints to x, y*
- The return value must live for at least lifetime 'a' *based on lifetime*.

↳ *For the return value life time is the exact same. Another constraint. 3 constraints here.*

Actual lifetime will be greater or equal to 'a'. So set of constraints to be solved. The approach of constraint solver based on monotone FWR.

More about each basic block: we are performing Dark Flow analysis:
we analyze blocks and add constraints.

Borrow Checking with MIR (step 2)

The borrow checker analyzes the control flow graph to ensure that all possible execution paths respect these lifetimes:

- Basic block 1: It verifies that returning x is valid because x has lifetime ' a ' which matches the return type
- Basic block 2: It verifies that returning y is valid because y has lifetime ' a ' which matches the return type

Borrow Checking (step 3)

For the `main` function, the borrow checker would create MIR that includes lifetime information for the local variables and function call:

```
// Simplified MIR for main

fn main() {
    let _0: ();                                // Return place
    let _1: String;                            // string1
    let _2: String;                            // string2
    let _3: &'1 str,                          // result, where '1 is a fresh
                                                // lifetime variable
    let mut _4: &'2 str;                      // Reference to string1
    let mut _5: &'3 str;                      // Reference to string2
```

*place to store references
for borrowing*

• At the end we know body of a function satisfies constraints.
But this is not enough. The function ~~largest~~ satisfies constraints, but real lifetime is lifetime of actual parameters. We plug function in Ctx where function is called. We perform analysis also in basic block associated to main.

Borrow Checking (step 3 cont)

```
// Basic block 0
x _1 = String::from("long string is long");
y _2 = String::from("xyz");

// Create borrows with explicit lifetimes
_4 = &'2 _1; } Same lifetime of x and y.      // Borrow of string1
_5 = &'3 _2; } Same lifetime of x and y.      // Borrow of string2
①

// Call longest with inferred lifetime constraints
// This causes the borrow checker to unify lifetimes '2 and '3 into '1 ②
_3 = longest::<'1>(_4, _5);    <'1> is lifetime of returned lifetime.
// ... rest of the function ...
```

① Imagine we say lifetime one and lifetime 2. Longest has lifetime '1.

② Create borrow with explicit lifetime.

② Umphy: algorithm provides satisfaction of constraints (when you have stuff that is more complex)

- What does lifetime ' $'_1$, ' $'_2$, ' $'_3$ ' mean? Those are symbolic lifetimes. Like x_1, x_2, x_3 . And we check that $x_1 = x_2 = x_3$.
lifetime of
 - ↳ constraints are satisfied

Lifetime Constraint Solving

The borrow checker then solves the constraints to determine concrete lifetimes:

1. It determines that '2 (the lifetime of the reference to string1) must be at least as long as '1 (the lifetime used in the call to longest)
2. It determines that '3 (the lifetime of the reference to string2) must be at least as long as '1 as well
3. It verifies that both string1 and string2 live long enough to satisfy these constraints

A Rust code

```
fn main() {  
    let mut x = 5;  
    let y: &mut i32 = &mut x; // '2: mutable borrow  
    let z: &i32 = &*y; // '1: immutable reborrow  
    let _result = *z; // read value  
    // return result (implicitly)  
}
```

Implicit result vs the value Z. How do we manage this: any

basic block has special place to store result.

- We provide result by returning a value. We have this assignment as implicit return. Imagine you write `result = *z`. You will have a warning

Alternative formulation

```
fn main() {  
    let mut x = 5;  
    let y: &mut i32 = &mut x; // '2: mutable borrow  
    let z: &i32 = &*y;           // '1: immutable reborrow  
    let result = *z;           // read value  
    println!("{}", result);  
}
```

This is something like MIR representation of Prog.

MIR Regions

```
// This code shows explicit lifetime regions
let mut _0: i32;
let _1: &'1 i32; Z = &y // Reference with explicit region '1
let _2: i32;
let _3: &'2 mut i32; // Mutable reference with region '2
bb0: {
    _2 = const 5_i32; X = 5
    _3 = &'2 mut _2; Y = &mut X // Begin borrow with region '2
    _1 = &'1 (*_3); Z = &y // Immutable reborrow with region '1
    _0 = (*_1); all uses of Z // Use of immutable borrow
    return; // End of all regions
}
```

L_y = &mut X

Two regions, 1, and 2 associated to the variables.

The borrow checker ensures that:

1. Region ' \cdot_2 ' includes all uses of $_3 = y$ *Region '2' (the one of y's mutable borrow) includes all uses of y)*
2. Region ' \cdot_1 ' includes all uses of $_1 = z$
3. *Mutable and immutable borrows don't overlap improperly* *Region '1' (the one of z's immutable borrow) includes all uses of z).*

rust

Copia Modifica

```
let mut _0: i32;           // return value
let _1: &'1 i32;           // immutable ref with lifetime '1
let _2: i32;               // local variable holding 5
let _3: &'2 mut i32;       // mutable ref with lifetime '2

bb0: {
    _2 = const 5_i32;       // let mut x = 5;
    _3 = &'2 mut _2;        // y = &mut x;
    _1 = &'1 (*_3);         // z = &y; (reborrow immutably)
    _0 = (*_1);             // let _result = *z;
    return;                 // End of function
}
```

1. Region '`'2` includes all uses of `_3` (the mutable reference)

💡 Why does this matter?

Because if `_3` (your `&mut x`) is used *outside* of `'2`, that means you're using a reference after its lifetime ended.

That's **use-after-free** territory — the worst kind of bug.

💡 By making `'2` include *all uses* of `_3`, the compiler guarantees:

"Hey, whenever you use `_3`, I'll make sure the thing it points to is still valid and no one else is using it in a way that would break stuff."

2. Region '`'1` includes all uses of `_1` (the immutable reference)

Same logic, but now for `_1`, the immutable reference.

💡 Why is this needed?

Imagine this:

rust

Copia Modifica

```
let x = 5;
let r = &x; // '1 starts
... // r used here
// what if '1 ended before last use of r?
```

If `'1` didn't include *all uses* of `r`, you could use `r` after it expired — again, **use-after-free** or undefined behavior.

Oh! So we are just saying: the region `'2` we are defining must contain all usages of `y` that has mutable borrow. While `'1` contains all usages of `z` that does immutable borrow. We are just laying down the rules under which we will define the regions in the analysis. Correct?

Exactly, Giovanni — yes, that's it! 🎉

You've nailed the key idea:

We're not saying *how long* `'1` and `'2` should be at the start — we're saying:

"Let's define regions `'1` and `'2` such that they include all the uses of their respective borrows — and then we'll check that those regions don't overlap in a way that breaks Rust's rules."

It's like setting the ground rules for a puzzle:

Optimizations

- MIR enables Rust-specific optimizations like:
 - Dead code elimination
 - Copy propagation
 - Loop simplification
 - Inlining
 - Removing unnecessary borrows

You can do a lot of things

Inspecting MIR

- You can view the MIR for your code by using the -Z dump-mir flag when compiling:

```
rustc -Z dump-mir=all my_code.rs
```

- Alternatively, you can use the Rust Playground with the "MIR" button to see the MIR for your code online.

MIR in the Rust Ecosystem

MIR serves as a foundation for many Rust tools and features:

1. **Miri**: The MIR interpreter for constant evaluation and undefined behavior detection
2. **Polonius**: The next-generation borrow checker Borrow checker, called Polonius
3. **Non-Lexical Lifetimes (NLL)**: More precise lifetime analysis
4. **Optimization passes**: Rust-specific optimizations before LLVM
→ Dataflow FW for static analysis
5. **rustc_mir_dataflow**: Framework for dataflow analysis

A COMPLETE EXAMPLE

The code

```
fn main() {  
    let mut x = 5;  
    let y = &x;  
    x = 10;  
    println!("{} {}", y);  
}
```

The screenshot shows the Rust playground interface. At the top, there's a toolbar with buttons for RUN, DEBUG, STABLE, SHARE, TOOLS, and CONFIG. The code editor window contains the following Rust code:

```
1 fn main() {
2     let mut x = 5;
3     let y = &x;
4     x = 10;
5     println!("{}", y);
6 }
7
8 }
```

Below the code editor is an execution output panel titled "Execution". It displays the error message and the corresponding stack trace:

```
error[E0506]: cannot assign to `x` because it is borrowed
--> src/main.rs:5:5
   |
4 |     let y = &x;
   |             -- `x` is borrowed here
5 |     x = 10;
   |     ^^^^^^ `x` is assigned to here but it was already borrowed
6 |     println!("{}", y);
   |             - borrow later used here
```

At the bottom of the execution panel, there's a note and a warning message:

For more information about this error, try `rustc --explain E0506`.
warning: `playground` (bin "playground") generated 1 warning

MIR Translation

```
fn main() {  
    let mut _0: () ; // return value (unit)  
    let mut _1: i32; // x  
    let _2: &i32; // y: reference to x  
    let mut _3: () ; // temp for println  
  
    bb0: {  
        _1 = const 5i32; // let mut x = 5;  
        _2 = &_1; // let y = &x;  
        _1 = const 10i32; // x = 10;  
        // Error: _1 is still borrowed by _2  
        _3 = call println(_2); // println!("{}", y);  
        return;  
    }  
}
```

• This is the translation

✓

Borrow Checking

MIR checks CFG:

MIR does **lifetime checking** by doing **dataflow analysis** over the **Control Flow Graph (CFG)** to track:

- ① 1. When borrows start and end
- 2. Which variables are live at each program point
- 3. Whether a variable is accessed while it's still borrowed

Borrow checker
does this basically

Back to our example

```
fn main() {  
    let mut _0: () ; // return value (unit)  
    let mut _1: i32; // x  
    let _2: &i32; // y: reference to x  
    let mut _3: () ; // temp for println  
  
    bb0: {  
        _1 = const 5i32; // let mut x = 5;  
        _2 = &_1; // let y = &x; → creates the borrowing; borrow checker knows w/ starts here  
        _1 = const 10i32; // x = 10;  
        // Error: _1 is still borrowed by _2  
        _3 = call println(_2); // println!("{}",  
y);  
        return;  
    }  
}
```

The statement `_2 = &_1;` creates a shared borrow of `_1`. The lifetime of this borrow starts here.

→ creates the borrowing; borrow checker knows w/ starts here

Back to our example

```
fn main() {  
    let mut _0: ();                      // return value (unit)  
    let mut _1: i32;                     // x  
    let _2: &i32;                        // y: reference to x  
    let mut _3: ();                      // temp for println  
  
    bb0: {  
        _1 = const 5i32;                // let mut x = 5;  
        _2 = &_1;                      // let y = &x;  
        _1 = const 10i32;               // x = 10;  
        // Error: _1 is still borrowed by _2  
        _3 = call println(_2);         // println!("{}",  
y);  
        return;  
    }  
}
```

The statement `_2 = &_1;` creates a **shared borrow of `_1`**. The lifetime of this borrow starts **here**.

The compiler sees that `_1` is mutated (`_1 = const 10i32`) while it is borrowed.

This violates Rust's rule: you can't mutate something while it's borrowed.

↑
You have a mutation of x where borrow of _1 is still active

Back to our example

```
fn main() {  
    let mut _0: ();                      // return value (unit)  
    let mut _1: i32;                     // x  
    let _2: &i32;                        // y: reference to x  
    let mut _3: ();                      // temp for println  
  
    bb0: {  
        _1 = const 5i32;                // let mut x = 5;  
        _2 = &_1;                      // let y = &x;  
        _1 = const 10i32;               // x = 10;  
        // Error: _1 is still borrowed by _2  
        _3 = call println(_2);         // println!("{}",  
                                         y);  
        return;  
    }  
}
```

The statement `_2 = &_1;` creates a **shared borrow of _1**. The lifetime of this borrow starts **here**.

The compiler sees that `_1` is **mutated** (`_1 = const 10i32`) while it is borrowed.

This violates Rust's rule: **you can't mutate something while it's borrowed**.

Even though lexically the borrow of `y` ends at the end of the block, **NLL (Non-Lexical Lifetimes)** tries to narrow the lifetime to the actual last use.

In this case, `println!("{}", y);` occurs after the mutation. Overall the borrow is still live.

Not correct!

Fix the code

Easy: we fix the code by moving the use of **y** before modifying **x**.

```
fn main() {  
    let mut x = 5;  
    let y = &x;  
    println!("{} {}", y); // use the borrow  
    x = 10; // now it's safe to modify  
}
```

What about MIR?

```
bb0: {  
    _1 = const 5i32;  
    _2 = &_1;  
    _3 = call println(_2);      // last use of _2  
    _1 = const 10i32;          // now borrow is over, mutation ok  
    return;  
}
```



Tracking borrow corresponds to liveness analysis

- Borrow checkers identifies borrow regions where a borrow is valid

Borrowing Summary

| Concept | Explanation |
|-----------------------------|--|
| Liveness Analysis | Tracks which variables are used at each program point. |
| Borrow Regions | MIR borrows have explicit start/end points for better precision than lexical scopes. |
| Place and Projection | Borrows of nested fields (e.g., x.y.z) are tracked precisely. |
| Two-phase borrows | Allows patterns like <code>vec.push(vec.len())</code> via MIR analysis. |
| NLL (Non-Lexical Lifetimes) | MIR enables finer lifetime granularity than the surface syntax. |

Path solution for assignment
Collection of constraints + Solution

- BONUS

TRAITS

Traits in Rust

A **trait** is the language construct that defines **shared behavior**. It's similar to **interfaces** in languages (e.g Java), but more expressive and tightly integrated with Rust's type system.

Key Features:

- A **trait** defines a set of method signatures.
- Types **implement** traits to declare that they support those methods.
- Traits enable **static (compile-time) polymorphism**.

Imagine you have class A with method m, A(m) and class A₁ that overrides m. If I do a.A and call a.m(). I execute method of A. Because among metadata in heap for A, I have a ph to method table (VTABLE for C++ people) and you will find code of M. In Java you can do A a₁=new A₁(). I type a₁ A,

In Java you have dynamic dispatch and it will execute in association
to A1 because I do this dynamically (dynamic dispatch). For static
dispatch you execute m of A.

An Example

```
trait Speak {  
    fn speak(&self);  
}  
struct Dog;  
struct Cat;  
  
impl Speak for Dog {  
    fn speak(&self) {  
        println!("Woof!");  
    }  
}  
  
impl Speak for Cat {  
    fn speak(&self) {  
        println!("Meow!");  
    }  
}
```

↑ self param of obj calling it.

Usign the Trait

```
fn make_speak<T: Speak>(animal: T) {  
    animal.speak();  
}  
  
fn main() {  
    let d = Dog;  
    let c = Cat;  
    make_speak(d);  
    make_speak(c);  
}
```

Using the Trait

```
fn make_speak<T: Speak>(animal: T) {  
    animal.speak();  
}
```

```
fn main() {  
    let d = Dog;  
    let c = Cat;  
    make_speak(d);  
    make_speak(c);  
}
```

This is **static dispatch**: the compiler generates specialized code for each concrete type.

↑
Thus is done statically, there is static dispatch to solve code for function

Default behaviour

```
trait Greet {  
    fn greet(&self) {  
        println!("Hello!");  
    }  
}
```

Default implementation is thus, but
we can refine it if we override
default implementation

Any type implementing Greet will inherit greet() unless
overridden.

Dynamic dispatch

```
fn say_something(speaker: &dyn Speak) {  
    speaker.speak(); // dynamic dispatch  
}
```

↑ You call method speak by using dynamic dispatch: you look at the actual object to solve further

```
trait Area {
    fn area(&self) -> f64;
}

struct Circle { radius: f64 }
struct Rectangle { width: f64, height: f64 }

impl Area for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius * self.radius
    }
}

impl Area for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}
```

Memory unsafe in Rust

- Yes!!! You understand correctly
- Even though Rust has been designed for **guaranteeing memory safety**, there are still scenarios where you can write **memory-unsafe code**, but only when you **explicitly opt into unsafe**.

Rust Safety

- Rust's safety guarantees apply only to "**safe Rust**".
- When you write unsafe code, you're telling the compiler:
"Trust me, I know what I'm doing."
- This allows you to perform operations the compiler normally disallows, but it also opens the door to **undefined behavior**, just like in C or C++.

Deferencing a dangling pointer

```
fn dangling_pointer() {  
    let x = Box::new(42);  
    let ptr: *const i32 = &*x;  
  
    drop(x); // deallocate the memory  
    unsafe {  
        println!("{} {}", *ptr); // use-after-free  
    }  
}
```

x is dropped, so the heap memory is freed.
ptr still points to that memory.
Dereferencing ptr causes **undefined behavior**.

Mutable aliases

```
fn mutable_aliasing() {  
    let mut x = 42;  
    let r1: *mut i32 = &mut x;  
    let r2: *mut i32 = &mut x;  
  
    unsafe {  
        *r1 = 1;  
        *r2 = 2; // simultaneous mutable access  
    }  
  
    println!("x = {}", x);  
}
```

Rust strictly forbids aliasing of mutable references.
In safe code, the borrow checker would prevent this.
In unsafe, you can break this rule, causing data races or logic errors.

Buffer overflow

```
fn buffer_overflow() {  
    let mut arr = [8; 5];  
    let ptr = arr.as_mut_ptr();  
  
    unsafe {  
        *ptr.add(10) = 1; // write beyond bounds  
    }  
}
```

Best practices

Adopt **safe Rust** wherever possible.

Use **unsafe** in **small, well-audited sections**.

Wrap **unsafe code in safe abstractions** (e.g., types or functions).

Document invariants that unsafe code relies on. *Document why you did unsafe things*

Use **tools like miri** to test unsafe logic. *and why you did them*