# Rust programming

## Course – 7

Gavrilut Dragos

rev 10

# Agenda for today

1. Generics
2. Match

# Generics

# Generics

Generics are a way of describing how methods, structs, enums and traits can be built based on a template where the type(s) that is(are) being used in those constructs is/are unknown and will be replace by the compiler at built time.

Generics are very similar to C++ concept (C++20) or somehow similar cu C++ templates, but there are a couple of exceptions (for example, a generic in Rust has the semantic ability to describe some limitations).

Keep in mind that generic work by build code based on a template (this means that using multiple generics will increase the size of your final binary).

# Generics

Generics can be applied for methods/functions or structures/enum and traits. The general format is:

1. Functions

   **fn** name <$Type_1$:*Bounds*, $Type_2$:*Bounds* ,…$Type_n$:*Bounds*> (…) ->

   or

   **fn** name <$Type_1$, $Type_2$ ,…$Type_n$> (…) -> *ReturnType*
   where $Type_1$:*Bounds* ,…$Type_n$:*Bounds* {…}

2. Struct/Enum/Traits

   **struct** name <$Type_1$:*Bounds*, $Type_2$:*Bounds* ,…$Type_n$:*Bounds*> (…) ->

   or

   **struct** name <$Type_1$, $Type_2$ ,…$Type_n$> (…) where $Type_1$:*Bounds* ,…$Type_n$:*Bounds* {…}

**OBS**: *Bounds are a combination of traits and lifetime rules that explain some requirements for a type used in a template.*

# Generics

Generics can be applied for methods/functions or structures/enum and traits. The general format is:

3. Template/Generic methods within the implementation of a type

```
impl TypeName
{
        fn name <Type1:Bounds, Type2:Bounds ,…Typen:Bounds> (…)  {…}
}
```

or

```
impl TypeName
{
        fn name <Type1, Type2 ,…Typen> (…) -> ReturnType
        where Type1:Bounds ,…Typen:Bounds
        {…}
}
```

# Generics

Generics can be applied for methods/functions or structures/enum and traits. The general format is:

4. Traits

```
trait TraitName <Type1:Bounds, Type2:Bounds ,…Typen:Bounds>
{
        fn method1 (…);
        fn method2 (…);
        …..
        fn methodn (…);
}
```

# Generics

## 1. Generic functions

*Rust*

```rust
fn print<T>(v: T)  {
    println!("{:?}",v);
}

fn main() {
    print::<i32>(10);
    print::<f32>(1.5);
    print::<&str>("Hello");
}
```

*Error*

```
error[E0277]: `T` doesn't implement `Debug`
 --> src\main.rs:4:21
  |
4 |     println!("{:?}",v);
  |                     ^ `T` cannot be formatted using `{:?}` because it doesn’t
  |                       implement `Debug`
  |
  = note: this error originates in the macro `$crate::format_args_nl` (in Nightly
builds, run with -Z macro-backtrace for more info)
help: consider restricting type parameter `T`
  |
3 | fn print<T: std::fmt::Debug>(v: T)  {
  |           +++++++++++++++++
```

# Generics

## 1. Generic functions

*Rust*

```
fn print<T>(v: T)  {
    println!("{:?}",v);
```

Notice that creating a stand-alone template is not enough. Rust requires an explicit restriction (in this case since we are using ***println*** macro to output value "v", type "T" must implement **Debug** (due to the use of {:?} ) in order to be printable.

***Error***

```
error[E0277]: `T` doesn't implement `Debug`
 --> src\main.rs:4:21
  |
4 |     println!("{:?}",v);
  |                     ^ `T` cannot be formatted using `{:?}` because it doesn't
  |                       implement `Debug`
  |
  = note: this error originates in the macro `$crate::format_args_nl` (in Nightly
builds, run with -Z macro-backtrace for more info)
help: consider restricting type parameter `T`
  |
3 | fn print<T: std::fmt::Debug>(v: T)  {
  |           +++++++++++++++++
```

# Generics

## 1. Generic functions

*Rust*

```rust
use std::fmt::Debug;

fn print<T: Debug>(v: T)  {
    println!("{:?}",v);
}

fn main() {
    print::<i32>(10);
    print::<f32>(1.5);
    print::<&str>("Hello");
}
```

**Output**

```
10
1.5
"Hello"
```

*Rust*

```rust
use std::fmt::Debug;

fn print<T>(v: T)
where
    T: Debug,
{
    println!("{:?}", v);
}
fn main() {
    print::<i32>(10);
    print::<f32>(1.5);
    print::<&str>("Hello");
}
```

**Output**

```
10
1.5
"Hello"
```

# Generics

In reality, Rust actually builds 3 functions (one for each type T [i32,f32 and &str]  used)

*Rust*

```rust
use std::fmt::Debug;

fn print<T: Debug>(v: T)  {
    println!("{:?}",v);
}

fn main() {
    print::<i32>(10);
    print::<f32>(1.5);
    print::<&str>("Hello");
}
```

↔

*Rust equivalent code*

```rust
fn print_i32(v: i32)  {
    println!("{:?}",v);
}
fn print_f32(v: f32)  {
    println!("{:?}",v);
}
fn print_str(v: &str)  {
    println!("{:?}",v);
}
fn main() {
    print_i32(10);
    print_f32(1.5);
    print_str("Hello");
}
```

**Output**

```
10
1.5
"Hello"
```

# Generics

Generic are identical to using **impl <trait>** (except that **turbofish** can not be used).

*Rust*

```rust
use std::fmt::Debug;

fn print<T: Debug>(v: T)  {
    println!("{:?}",v);
}

fn main() {
    print::<i32>(10);
    print::<f32>(1.5);
    print::<&str>("Hello");
}
```

**Output**

```
10
1.5
“Hello”
```

*Rust*

```rust
use std::fmt::Debug;

fn print(v: impl Debug) {
    println!("{:?}",v);
}

fn main() {
    print::<i32>(10);
    print::<f32>(1.5);
    print::<&str>("Hello");
}
```

***Error***

```
error[E0107]: function takes 0 generic arguments but 1 generic argument was supplied
12 |     print::<f32>(1.5);
   |     ^^^^^------- help: remove these generics
   |     |
   |     expected 0 generic arguments
```

# Generics

Generic are identical to using **impl <trait>** (except that **turbofish** can not be used).

*Rust*

```rust
use std::fmt::Debug;

fn print<T: Debug>(v: T)  {
    println!("{:?}",v);
}

fn main() {
    print::<i32>(10);
    print::<f32>(1.5);
    print::<&str>("Hello");
}
```

**Output**

```
10
1.5
“Hello”
```

*Rust*

```rust
use std::fmt::Debug;

fn print(v: impl Debug) {
    println!("{:?}",v);
}

fn main() {
    print(10);
    print(1.5);
    print("Hello");
}
```

Outside of turbofish usage, the behavior is similar (using **impl <trait>** works like a syntatictic sugar).

# Generics

## 1. Generic functions (a more complex example)

*Rust*

```rust
use std::ops::Add;

fn compute_sum<T: Add>(v1: T, v2: T) -> T
{
    return v1 + v2;
}
fn main() {
    let r1 = compute_sum::<i32>(10,5);
    let r2 = compute_sum::<f32>(10.5,5.5);
    println!("{r1},{r2}");
}
```

***Error***

```
error[E0308]: mismatched types
 --> src\main.rs:4:12
  |
3 | fn compute_sum<T: Add>(v1: T, v2: T) -> T {
  |                - this type parameter    - expected
  |                                         `T` because of return type
4 |     return v1+v2;
  |            ^^^^^ expected type parameter `T`,
  |                  found associated type
  |
```

*So … what happens in this case ?*

# Generics

## 1. Generic functions (a more complex example)

*Rust*

```
use std::ops::Add;

fn compute_sum<T: Add>(v1: T, v
{
    return v1 + v2;
}
fn main() {
    let r1 = compute_sum::<i32>(10,5);
    let r2 = compute_sum::<f32>(10.5,5.5);
    println!("{r1},{r2}");
}
```

We need to specify that T must implement the Add trait in order for the addition of v1+v2 to be possible.

*We have specified that T must support the Add trait as we know that we will add v1 with v2*

# Generics

## 1. Generic functions  (a more complex example)

*Rust*

```
use std::ops::Add;

fn compute_sum<T: Add>(v1: T, v2: T) -> T
{
    return v1 + v2;
}
fn main() {
    let r1 = compute_sum::<i32>(10,5);
    let r2 = compute_su
    println!("{r1},{r2}
}
```

*Rust (from ariths.rs)*

```
pub trait Add<Rhs = Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Notice that in order to use the Add trait we need to specify the **Output** type.

*We have specified that T must support the Add trait as we know that we will add v1 with v2*

# Generics

## 1. Generic functions  (a more complex example)

*Rust*

```rust
use std::ops::Add;

fn compute_sum<T: Add<Output=T>>(v1: T, v2: T) -> T {
    return v1+v2;
}
fn main() {
    let r1 = compute_sum::<i32>(10,5);
    let r2 = compute_sum::<f32>(10.5,5.5);
    println!("{r1},{r2}");
}
```

| **Output** |
| --- |
| 15,16 |

*Notice that in order to properly explain how the template should work, we need to specify that the result of the addition will be of type **T** as well ! We do this by adding the <Output=T> to the specifications of Add trait.*

# Generics

But what if we want to add more restrictions (e.g. we want a specific type to implement multiple traits). The actual format of *Bounds* allows this:

Format: **Type** : *Trait*$_1$ + *Trait*$_2$ + … *Trait*$_n$ + *Lifetime*

Where a trait can be defined in one of the following ways:

- With its name: **Type** : *Trait*
- With its name defined as a template: **Type** : *Trait<T>*
- With its name and one or more inner types defined : **Type** : *Trait<Output=…>*

**OBS**: *Keep in mind that* **Type** *does not necessarily have to be a template type (it could also be an existing type that requires a bound related to the current template: for example:* **i32** : *From<T> means that i32 must implement Trait<T>*

# Generics

Let's consider a slightly more complex function (a function that adds two numbers of potentially different types and return a value of a potentially different type.

*Rust*

```rust
use std::ops::Add;

fn compute_sum<T1, T2, T3>(v1: T1, v2: T2) -> T3
where
    T3: Add<Output = T3> + From<T1> + From<T2>,
{
    return T3::from(v1) + T3::from(v2);
}
fn main() {
    let r1 = compute_sum::<i32, i8, i64>(10, 5);
    let r2 = compute_sum::<f32, i8, f64>(10.5, 5);
    println!("{r1},{r2}");
}
```

| **Output** |
|---|
| 15,15.5 |

# Generics

Let's consider a slightly more complex function (a function that adds two numbers of potentially different types and return a value of a potentially different type.

*Rust*

```
use std::ops::Add;

fn compute_sum<T1, T2, T3>(v1: T1, v2: T2) -> T3
where
    T3: Add<Output = T3> + From<T1> + From<T2>,
{
    return T3::from(v1) + T3::from(v2);
}
fn main() {
    let r1 = compute_sum::<i32, i8, i64>(10, 5);
    let r2 = compute_sum::<f32, i8, f64>(10.5, 5);
    println!("{r1},{r2}");
}
```

| Output |
| --- |
| 15,15.5 |

Trait **Add** is needed because we add two values of type T3

# Generics

Let’s consider a slightly more complex function (a function that adds two numbers of potentially different types and return a value of a potentially different type.

*Rust*

```
use std::ops::Add;

fn compute_sum<T1, T2, T3>(v1: T1, v2: T2) -> T3
where
    T3: Add<Output = T3> + From<T1> + From<T2>,
{
    return T3::from(v1) + T3::from(v2);
}
fn main() {
    let r1 = compute_sum::<i32, i8, i64>(10, 5);
    let r2 = compute_sum::<f32, i8, f64>(10.5, 5);
    println!("{r1},{r2}");
}
```

| **Output** |
| --- |
| 15,15.5 |

Trait **From<T1>** is needed because we convert “v1” to  type T3

# Generics

Let’s consider a slightly more complex function (a function that adds two numbers of potentially different types and return a value of a potentially different type.

*Rust*

```
use std::ops::Add;

fn compute_sum<T1, T2, T3>(v1: T1, v2: T2) -> T3
where
    T3: Add<Output = T3> + From<T1> + From<T2>;
{
    return T3::from(v1) + T3::from(v2);
}
fn main() {
    let r1 = compute_sum::<i32, i8, i64>(10, 5);
    let r2 = compute_sum::<f32, i8, f64>(10.5, 5);
    println!("{r1},{r2}");
}
```

| **Output** |
| --- |
| 15,15.5 |

Trait **From<T2>** is needed because we convert “v2” to type T3

# Generics

Let’s analyze another example:

*Rust*

```rust
use std::fmt::Display;
use std::ops::Add;
fn compute_sum<T1, T2, T3>(v1: T1, v2: T2) -> T3
where
    T1: Display + Copy,
    T2: Display + Copy,
    T3: Add<Output = T3> + From<T1> + From<T2> + Display,
{
    let result = T3::from(v1) + T3::from(v2);
    println!("{}+{}={}", v1, v2, result);
    return result;
}
fn main() {
    let r1 = compute_sum::<i32, i8, i64>(10, 5);
    let r2 = compute_sum::<f32, i8, f64>(10.5, 5);
    println!("{r1},{r2}");
}
```

**Output**

```
10+5=15
10.5+5=15.5
15,15.5
```

# Generics

Let’s analyze another example:

*Rust*

```
use std::fmt::Display;
use std::ops::Add;
fn compute_sum<T1, T2, T3>(v1: T1, v2: T2) -> T3
where
    T1: Display + Copy,
    T2: Display + Copy,
    T3: Add<Output = T3> + From<T1> + From<T2> + Display,
{
    let result = T3::from(v1) + T3::from(v2);
    println!("{}+{}={}", v1, v2, result);
    return result;
}
fn main() {
    let r1 = compute_sum::<i32, i8, i64>(10, 5);
    let r2 = compute_sum::<f32, i8, f64>(10.5, 5);
    println!("{r1},{r2}");
}
```

| Output |
| --- |
| 10+5=15 |
| 10.5+5=15.5 |
| 15,15.5 |

**Display** trait is needed for ***println***! macro

# Generics

Let’s analyze another example:

*Rust*

```
use std::fmt::Display;
use std::ops::Add;
fn compute_sum<T1, T2, T3>(v1: T1, v2: T2) -> T3
where
    T1: Display + Copy,
    T2: Display + Copy,
    T3: Add<Output = T3> + From<T1> + From<T2> + Display,
{
    let result = T3::from(v1) + T3::from(v2);
    println!("{}+{}={}", v1, v2, result);
    return result;
}
fn main() {
    let r1 = compute_sum::<i32, i8, i64>(10, 5);
    let r2 = compute_sum::<f32, i8, f64>(10.5, 5);
    println!("{r1},{r2}");
}
```

| **Output** |
|---|
| 10+5=15<br>10.5+5=15.5<br>15,15.5 |

The ownership of “v1” is transferred when T3::from(…) is called. As such, printing v1 is no longer possible, except for the case where “v1” implements the Copy trait.

# Generics

Bounds have another advantage. While in C++, its not that easy to enforce a specific type/list of types for a template variable, in Rust this can be easily obtained with a combination of bounds and traits.

Let's analyze the following problem: we want to create a generic/template where one of the parameters can only be selected from a specific set of types. In C++, we would have to use **static_assert** to achieve this, and even in this case we would still be limited by the fact that everything is written in a header and as such it can be easily modified.

In Rust, we will use a combination of traits and bounds to achieve the same result.

# Generics

Let's consider the following snippet and assume that we would like to make sure that type T is just some signed integer (one of i8, i32 or i128).

*Rust*

```
use std::fmt::Display;
fn print<T>(value: T)
where
    T: Display,
{
    println!("{}", value);
}
fn main() {
    print::<i32>(-5);
    print::<i8>(100);
}
```

**Output**

```
-5
100
```

# Generics

The solution is to create a special trait (that does not have to do anything), implement it for i8, i32 or i128, and finally add a bound for type T.

*Rust*

```
use std::fmt::Display;

trait JustIntegers {}
impl JustIntegers for i8 {}
impl JustIntegers for i32 {}
impl JustIntegers for i128 {}

fn print<T>(value: T)
where
    T: Display + JustIntegers
{
    println!("{}", value);
}
```

```
fn main() {
    print::<i32>(-5);
    print::<i8>(100);
}
```

**Output**

```
-5
100
```

# Generics

The solution is to create a special trait (that does not have to do anything), implement it for i8, i32 or i128, and finally add a bound for type T.

*Rust*

```rust
use std::fmt::Display;

trait JustIntegers {}
impl JustIntegers for i8 {}
impl JustIntegers for i32 {}
impl JustIntegers for i128 {}

fn print<T>(value: T)
where
    T: Display + JustIntegers
{
    println!("{}", value);
}
```

```rust
fn main() {
    print::<i32>(-5);
```

**Output**

```
-5
100
```

Trait **JustIntegers** is implemented for i8, i32 and i128

Type T must have the trait **JustIntegers**

# Generics

The solution is to create a special trait (that does not have to do anything), implement it for i8, i32 or i128, and finally add a bound for type T.

*Rust*

```rust
use std::fmt::Display;
trait JustIntegers {}
impl JustIntegers for i8 {}
impl JustIntegers for i32 {}
impl JustIntegers for i128 {}

fn print<T>(value: T)
where
    T: Display+JustIntegers,
{
    println!("{}", value);
}
fn main() {
    print::<u32>(1);
}
```

*Error*

```
error[E0277]: the trait bound `u32: JustIntegers` is not satisfied
  --> src\main.rs:14:18
   |
14 |     print::<u32>(1);
   |     ------------ ^ the trait `JustIntegers` is not implemented for `u32`
   |     |
   |     required by a bound introduced by this call
   |
   = help: the following implementations were found:
             <i128 as JustIntegers>
             <i32 as JustIntegers>
             <i8 as JustIntegers>
note: required by a bound in `print`
  --> src\main.rs:9:16
   |
7  | fn print<T>(value: T)
   |    ----- required by a bound in this
8  | where
9  |     T: Display+JustIntegers,
   |                ^^^^^^^^^^^^ required by this bound in `print`
```

# Generics

## 2. Structures

*Rust*

```rust
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T
}
fn main() {
    let p1 = Point::<i32>{ x:1,   y:2   };
    let p2 = Point::<f32>{ x:1.2, y:2.3 };
    println!("{:?}",p1);
    println!("{:?}",p2);
}
```

**Output**

```
Point { x: 1, y: 2 }
Point { x: 1.2, y: 2.3 }
```

*In this example we have create two objects: **p1** that has "x" and "y" of type i32, and **p2** where both "x" and "y" are of type f32.*

# Generics

In reality, Rust will construct two completely different structures:

*Rust*

```rust
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T
}
fn main() {
    let p1 = Point::<i32>{ x:1,   y:2   };
    let p2 = Point::<f32>{ x:1.2, y:2.3 };
    println!("{:?}",p1);
    println!("{:?}",p2);
}
```

*Rust (approximation)*

```rust
#[derive(Debug)]
struct Point_i32 { x: i32, y: i32 }
#[derive(Debug)]
struct Point_f32 { x: f32, y: f32 }

fn main() {
    let p1 = Point_i32{ x:1,   y:2   };
    let p2 = Point_f32{ x:1.2, y:2.3 };
    println!("{:?}",p1);
    println!("{:?}",p2);
}
```

# Generics

If we want to implement a method for a generic structure or enum, use the following format: **impl**<$T_1$, $T_2$,... $T_n$> *StructureName*<$T_1$, $T_2$,... $T_n$> { ... }

*Rust*

```rust
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T
}
impl<T> Point<T> {
    fn new(x: T, y: T)->Self { Point {x:x, y:y} }
}
fn main() {
    let p1 = Point::<i32>::new(1,2);
    let p2 = Point::<f32>::new(1.2,2.3);
    println!("{:?}",p1);
    println!("{:?}",p2);
}
```

**Output**

```
Point { x: 1, y: 2 }
Point { x: 1.2, y: 2.3 }
```

# Generics

Types can be inferred from the parameters used. For example, in this case, p1 is of type Point<i32> because “x” is 1 (an i32) and “y” is 2 (an i32).

*Rust*

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x: x, y: y }
    }
}
fn main() {
    let p1 = Point::new(1, 2); // Point<i32>
    let p2 = Point::new(1.2, 2.3); // Point<f64>
    println!("{:?}", p1);
    println!("{:?}", p2);
}
```

**Output**

```
Point { x: 1, y: 2 }
Point { x: 1.2, y: 2.3 }
```

# Generics

If parameters do not match the template, the compiler will throw an error.

*Rust*

```rust
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x: x, y: y }
    }
}
fn main() {
    let p1 = Point::new(1, 2.5);
    let p2 = Point::new(1.2, 2.3);
    println!("{:?}", p1);
    println!("{:?}", p2);
}
```

***Error***

```
error[E0308]: mismatched types
  --> src\main.rs:12:28
   |
12 |     let p1 = Point::new(1, 2.5);
   |                            ^^^ expected integer, found floating-point number
```

# Generics

You can also use "_" (underline) as a template parameter to ask Rust to infer the type:

*Rust*

```rust
fn f<K: From<T>, T>(x: T) -> K {
    K::from(x)
}
fn main() {
    let x = f::<i64, _>(5); // x is inferred to x64
    println!("{x}");
}
```

**Output**

```
5
```

In this case, we did not provide the type "T", but we've asked Rust to infer the type.

# Generics

Rust does not support specialized templates (like C++ does).

*Rust*

```rust
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x: x, y: y }
    }
}
impl Point<i32> {
    fn new(x: i32, y: i32) -> Self {
        Point { x: x * x, y: y * y }
    }
}
```

***Error***

```
error[E0592]: duplicate definitions with name `new`
  --> src\main.rs:7:5
    |
7   |     fn new(x: T, y: T) -> Self {
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^ duplicate definitions for `new`
...
12  |     fn new(x: i32, y: i32) -> Self {
    |     ------------------------------ other definition for `new`
```

# Generics

However, certain functions can be implemented specifically for a type:

*Rust*

```rust
struct Point<T> { x: T, y: T, }
impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x: x, y: y }
    }
}
impl Point<i32> {
    fn new_i32(x: i32, y: i32) -> Self {
        Point { x: x * x, y: y * y }
    }
}
fn main() {
    let p1 = Point::<i32>::new_i32(1, 2);
    let p2 = Point::<f32>::new(1.2, 2.3);
    println!("{:?}", p1);
    println!("{:?}", p2);
}
```

**Output**

```
Point { x: 1, y: 4 }
Point { x: 1.2, y: 2.3 }
```

# Generics

However, certain functions can be implemented specifically for a type:

*Rust*

```rust
struct Point<T> { x: T, y: T, }
impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x: x, y: y }
    }
}
impl Point<i32> {
    fn new_i32(x: i32, y: i32) -> Self {
        Point { x: x * x, y: y * y }
    }
}
fn main() {
    let p1 = Point::<i32>::new_i32(1, 2);
    let p2 = Point::<f32>::new_i32(1.2, 2.3);
    println!("{:?}", p1);
    println!("{:?}", p2);
}
```

***Error***

```
error[E0599]: no function or associated item named `new_i32` found for struct
`Point<f32>` in the current scope
  --> src\main.rs:18:28
   |
2  | struct Point<T> {
   | --------------- function or associated item `new_i32` not found for this
...
18 |     let p2 = Point::<f32>::new_i32(1.2, 2.3);
   |                            ^^^^^^^ function or associated item not found in
   |                                    `Point<f32>`
   |
   = note: the function or associated item was found for
           - `Point<i32>`------------------------------ other definition for
                                                        `new`
```

new_i32 exists only for templates of Point where T = i32. As such, this line can not compile.

# Generics

3. Generic methods implemented in a structure/enum

*Rust*

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn add<T>(&mut self, value: T)
    where
        i32: From<T>,
        T: Copy
    {
        self.x += i32::from(value);
        self.y += i32::from(value);
    }
}
```

```rust
fn main() {
    let mut p = Point { x: 5, y: 10 };
    p.add::<i32>(10);
    println!("P is ({},{})",p.x,p.y);
    p.add::<i8>(5);
    println!("P is ({},{})",p.x,p.y);
}
```

**Output**

```
P is (15,20)
P is (20,25)
```

# Generics

## 3. Generic methods implemented in a structure/enum

*Rust*

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn add<T>(&mut self, value: T)
    where
        i32: From<T>,
        T: Copy
    {
        self.x += i32::from(value);
        self.y += i32::from(value);
    }
}
```

```rust
fn main() {
    let mut p = Point { x: 5, y: 10 };
    p.add::<i32>(10);
    println!("P is ({},{})",p.x,p.y);
    p.add::<i8>(5);
    println!("P is ({},{})",p.x,p.y);
}
```

"add" is a generic method within the implementation for structure Point

# Generics

## 3. Generic methods implemented in a structure/enum

*Rust*

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn add<T>(&mut self, value: T)
    where
        i32: From<T>,
        T: Copy
    {
        self.x += i32::from(value);
        self.y += i32::from(value);
    }
}
```

```rust
fn main() {
    let mut p = Point { x: 5, y: 10 };
    p.add::<i32>(10);
    println!("P is ({},{})",p.x,p.y);
    p.add::<i8>(5);
    println!("P is ({},{})",p.x,p.y);
}
```

Since we need to convert a value of type T into an i32, then i32 must implement From<T>.

# Generics

## 3. Generic methods implemented in a structure/enum

*Rust*

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn add<T>(&mut self, value: T)
    where
        i32: From<T>,
        T: Copy
    {
        self.x += i32::from(value);
        self.y += i32::from(value);
    }
}
```

```rust
fn main() {
    let mut p = Point { x: 5, y: 10 };
    p.add::<i32>(10);
    println!("P is ({},{})",p.x,p.y);
    p.add::<i8>(5);
    println!("P is ({},{})",p.x,p.y);
}
```

Since we call i32::from twice, and the value’s ownership is transferred the first time, T must implement Copy so that the second i32::from could be valid.

# Generics

## 3. Generic methods implemented in a structure/enum

*Rust*

```rust
struct Point {
    x: i32, y: i32,
}
impl Point {
    fn add<T>(&mut self, value: T)
    where
        i32: From<T>,
        T: Copy
    {
        self.x += i32::from(value);
        self.y += i32::from(value);
    }
}
fn main() {
    let mut p = Point { x: 5, y: 10 };
    p.add::<&str>("test");
}
```

*Error*

```
error[E0277]: the trait bound `i32: From<&str>` is not satisfied
  --> src\main.rs:17:7
   |
17 |     p.add::<&str>("test");
   |       ^^^ the trait `From<&str>` is not implemented for `i32`
   |
   = help: the following implementations were found:
             <i32 as From<NonZeroI32>>
             <i32 as From<bool>>
             <i32 as From<i16>>
             <i32 as From<i8>>
           and 71 others
note: required by a bound in `Point::add`
  --> src\main.rs:8:14
   |
6  |     fn add<T>(&mut self, value: T)
   |        --- required by a bound in this
7  |     where
8  |         i32: From<T>,
   |              ^^^^^^^ required by this bound in `Point::add`
```

# Generics

## 4. Generic traits

*Rust*

```rust
trait ConvertorToType<T> {
    fn convert_to(self) -> T;
}
impl ConvertorToType<i32> for i8 {
    fn convert_to(self) -> i32 { (self as i32) * 2 }
}
impl ConvertorToType<f32> for i8 {
    fn convert_to(self) -> f32 { (self as f32) / 2.0 }
}

fn main() {
    let x: f32 = 123i8.convert_to();
    let y: i32 = 123i8.convert_to();
    println!("{x},{y}");
}
```

**Output**

```
61.5,246
```

# Generics

## 4. Generic traits

*Rust*

```
trait ConvertorToType<T> {
    fn convert_to(self) -> T;
}
impl ConvertorToType<i32> for i8 {
    fn convert_to(self) -> i32 { (self as i32) * 2 }
}
impl ConvertorToType<f32> for i8 {
    fn convert_to(self) -> f32 { (self as f32) / 2.0 }
}

fn main() {
    let x: f32 = 123i8.convert_to();
    let y: i32 = 123i8.convert_to();
    println!("{x},{y}");
}
```

This behavior is very similar to a specialized template from C++.

# Generics

## 4. Generic traits

*Rust*

```
trait ConvertorToType<T> {
    fn convert_to(self) -> T;
}
impl ConvertorToType<i32> for i8 {
    fn convert_to(self) -> i32 { (self as i32) * 2 }
}
impl ConvertorToType<f32> for i8 {
    fn convert_to(self) -> f32 { (self as f32) / 2.0 }
}

fn main() {
    let x: f32 = 123i8.convert_to();
    let y: i32 = 123i8.convert_to();
    println!("{x},{y}");
}
```

| Output |
|---|
| 61.5,246 |

Another similar way of writing the same thing is:

```
let x = ConvertorToType::<f32>::convert_to(123i8);
```

# Generics

At the same time, default implementation for a trait can be used as well.

*Rust*

```
trait ConvertorToType<T>
where
    Self: Sized,
    T: From<Self>
{
    fn convert_to(self) -> T {
        T::from(self)
    }
}
impl ConvertorToType<i32> for i8 {}
impl ConvertorToType<f32> for i8 {}

fn main() {
    let x = ConvertorToType::<f32>::convert_to(123i8);
    let y: i32 = 123i8.convert_to();
    println!("{x},{y}");
}
```

**Output**

```
123,123
```

# Generics

It is also possible to overwrite the original implementation:

*Rust*

```
trait ConvertorToType<T>
where
    Self: Sized,
    T: From<Self>
{
    fn convert_to(self) -> T { T::from(self) }
}
impl ConvertorToType<i32> for i8 {}
impl ConvertorToType<f32> for i8 {
    fn convert_to(self) -> f32 {
        1.2345
    }
}

fn main() {
    let x = ConvertorToType::<f32>::convert_to(123i8);
    let y: i32 = 123i8.convert_to();
    println!("{x},{y}");
}
```

| Output |
|---|
| 1.2345,123 |

This behavior is very similar to a specialized template from C++.

# Generics

It is also possible to overwrite the original implementation:

*Rust*

```
trait ConvertorToType<T>
where
    Self: Sized,
    T: From<Self>
{
    fn convert_to(self) -> T { T::fro   lf) }
}
impl ConvertorToType<i32> for i8 {}
imp

}

fn main() {
    let x = ConvertorToType::<f32>::convert_to(123i8);
    let y: i32 = 123i8.convert_to();
    println!("{x},{y}");
}
```

Why do we need this **Sized** trait here ?

When compiling the code, the compiler needs to know the size of a type in order to implement operation over it that might imply copy-ing an object. Since we already require for T to implement From<Self> and this implies ownership transfer of Self, Self must have a known size.

# Generics

It is also possible to overwrite the original implementation:

*Rust*

```
trait SomeValue {
    fn get_some_value()->Self;
}
impl SomeValue for i32 {
    fn get_some_value()->i32 { 12345 }
}

trait Initialize<T> where T: SomeValue
{
    fn convert_to(&self) -> T {
        T::get_some_value()
    }
}
impl Initialize<i32> for i8 {}

fn main() {
    let x = Initialize::<i32>::convert_to(&123i8);
    println!("{x}");
}
```

| **Output** |
| --- |
| 12345 |

Notice that we don't need to use **Sized** anymore in this case:

- method ***convert_to*** receives a reference
- default implementation of ***convert_to*** does not require any ownership transfer (assignments or From methods called)

# Generics

When using generics (methods, functions, structures, etc) we might need to use turbofish (::<…>) notation to refer to a specific implementation of a generic.

In many cases, Rust is able to infer the type (based on parameters) but sometimes (if several matches for the same generic exists) you might be required to use either this notation or other forms.

As a generic observation, it is preferred **NOT TO** use turbofish notation (except for cases where there is no other way around).

***Let's see some scenarios where this notation can be used. Pay close attention for alternative situations where a different syntax can be used.***

# Generics

Case 1:

*Rust*

```rust
struct MyNumber<T> {
    value: T
}
impl<T> MyNumber<T> {
    fn set(&mut self,x: T) {
        self.value = x;
    }
}
fn main() {
    let mut x = MyNumber::<i32>{value:0};
    x.set(123);
    println!("{}",x.value);
}
```

One case where turbofish can be used is if the structure is a template/generic and upon creation you need to explain to Rust the generic parameter(s).

In this particular case, we need to explain what is "T" when initializing a MyNumber object.

**Alternatives:**

```rust
let mut x = MyNumber{ value:0 };
```

```rust
let mut x:MyNumber<i32> = MyNumber{ value:0 };
```

**OBS**: *Notice that "x.set(123)" does not need to use turbofish notation as we already know*

# Generics

Case 2:

*Rust*

```rust
struct MyNumber {
    value: i32
}
impl MyNumber {
    fn set<T>(&mut self,x: T) where i32: From<T>{
        self.value = i32::from(x);
    }
}
fn main() {
    let mut x = MyNumber{value:10};
    x.set::<i8>(100i8);
    println!("{}",x.value);
}
```

In this case, method ***set*** from the implementation of MyNumber is generic and as such type “T” must be specified or inferred (if possible).

**Alternatives:**

```rust
x.set(100i8);
```

# Generics

Case 3:

*Rust*

```
struct MyNumber {
    value: i32,
}
trait ValueSetter<T> {
    fn set(value: T) -> Self;
}
impl ValueSetter<i8> for MyNumber {
    fn set(x: i8)->Self
    {
        MyNumber {
            value: i32::from(x),
        }
    }
}
fn main() {
    let x = MyNumber::set(12i8);
    println!("{}", x.value);
}
```

Notice that we did not use turbofish notation. This is because method set is not generic/template and as such using something like set::<i8>:: (…) is not a valid semantic expression.

# Generics

Case 4:

*Rust*

```rust
struct MyNumber<T> {
    value: T,
}
trait ValueSetter<V> {
    fn set(value: V) -> Self;
}
impl<T> ValueSetter<i8> for MyNumber<T>
where
    T: From<i8>,
{
    fn set(x: i8) -> Self {
        MyNumber {
            value: T::from(x),
        }
    }
}
fn main() {
    let x = MyNumber::<i8>::set(123i8);
    println!("{}", x.value);
}
```

In this case, we need turbofish to specify the template for MyNumber. Type “V” will be inferred by Rust from the argument value of set method.

**Alternatives:**

```rust
let x: MyNumber::<i8> = MyNumber::set(123i8);
```

# Generics

Case 5:

*Rust*

```rust
struct MyNumber<T> { value: T }
trait ValueSetter<V> { fn set(value: V) -> Self;}
impl<T> ValueSetter<i8> for MyNumber<T>
Where T: From<i8>,
{
    fn set(x: i8) -> Self {
        MyNumber { value: T::from(x) }
    }
}
impl<T> ValueSetter<i16> for MyNumber<T>
Where T: From<i16>,
{
    fn set(x: i16) -> Self {
        MyNumber { value: T::from(x) }
    }
}
fn main() {
    let x = MyNumber::<i32>::set(123 as i16);
    println!("{}", x.value);
}
```

In this case, we need to specify the type of MyNumber and we need to make sure that we specifically explain the parameter type of the method set.
Keep in mind that **::set::<i16>(…)** is not valid as method set is not generic.

**Alternatives:**

```rust
let x: MyNumber::<i32> = MyNumber::set(123 as i16);
```

# Generics

Case 6:

*Rust*

```rust
struct MyNumber<T> {
    value: T,
}
trait ValueSetter<V> {
    fn set(value: V) -> Self;
}
impl<T,V> ValueSetter<V> for MyNumber<T>
where
    T: From<V>,
{
    fn set(x: V) -> Self {
        MyNumber {
            value: T::from(x),
        }
    }
}
fn main() {
    let x = MyNumber::<i32>::set(123 as i16);
    println!("{}", x.value);
}
```

In this case, both “V” and “T” template parameters must be deducted. For “T” we can use turbofish notation, “V” will be obtained from the parameter of **set** method.

**Alternatives:**

```rust
let x: MyNumber::<i32> = MyNumber::set(123 as i16);
```

# Generics

Finally, let's discuss a little bit what is the advantage of requiring a strict list of traits within the definition of a Generic. Let's analyze the following two cases (Rust and C++)

*Rust*

```rust
struct Test {
    x: i32,
}

fn add_values<T>(v1: &T, v2: &T) -> T {
    return v1 + v2;
}

fn main() {
    let t1 = Test { x: 10 };
    let t2 = Test { x: 20 };
    let t3 = Test { x: 0 };
    t3 = add_values(&t1, &t2);
}
```

*C++*

```cpp
struct Test {
    int x;
};

template <typename T>
T add_values(T& v1, T& v2) {
    return v1 + v2;
}

void main() {
    Test t1 = { 10 };
    Test t2 = { 20 };
    Test t3;
    t3 = add_values(t1, t2);
}
```

# Generics

Now → let's look at how errors are presented in both cases (notice that structure Test has no add operator in both cases, and as such "v1+v2" where v1,v2 is T is not possible.

*Rust*

```
struct Test {
    x: i32,
}

fn add_values<T>(v1: &T, v2: &T) -> T {
    return v1 + v2;
```

***Error***

```
error[E0369]: cannot add `&T` to `&T`
 --> src\main.rs:5:15
  |
5 |     return v1 + v2;
  |            -- ^ -- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
4 | fn add_values<T: std::ops::Add<Output = &T>>(v1: &T, v2: &T) -> T {
  |              ++++++++++++++++++++++++++++
```

*C++*

```
struct Test {
    int x;
};

template <typename T>
T add_values(T& v1, T& v2) {
    return v1 + v2;
}

void main() {
    Test t1 = { 10 };
    Test t2 = { 20 };
    Test t3;

}
```

***Error***

```
error C2676: binary '+': 'T' does not
define this operator or a conversion to a
type acceptable to the predefined operator
```

# Generics

Rust also support constants as a generic parameter (similar to the ones from C++). A constant parameter must de defined in the template declaration with the keyword const followed by the generic parameter name and its type (**const** N: *type*). Type N can be one of the following: u8, u16, u32, u64, u128, i8, i16, i32, i64, i128, usize , isize, char and bool.

*Rust*

```
#[derive(Debug)]
struct FixArray <T,const N: usize> {
    elements: [T;N]
}
fn main() {
    let a:FixArray<i32,5> = FixArray {elements: [0;5]};
    let b:FixArray<char,9> = FixArray {elements: ['A';9]};
    println!("{:?}",a);
    println!("{:?}",b);
}
```

**Output**

```
FixArray { elements: [0, 0, 0, 0, 0] }
FixArray { elements: ['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A'] }
```

# Generics

Let's see a more complex example:

| *Rust (Generic structure declaration)* | *(Generic structure implementation)* |
|---|---|

```rust
#[derive(Debug)]
struct FixArray<T, const N: usize>
where
    T: std::ops::AddAssign,
    T: From<u8>,
    T: Copy
{
    elements: [T; N],
}
```

```rust
impl<T, const N: usize> FixArray<T, N>
where
    T: std::ops::AddAssign,
    T: From<u8>,
    T: Copy
{
    fn new(value: T) -> Self {
        FixArray::<T, N> {
            elements: [value; N],
        }
    }
    fn consecutive(start: T) -> Self {
        let mut x: FixArray<T, N> = FixArray::new(start);
        let mut temp = start;
        for i in &mut x.elements {
            *i = temp;
            temp += T::from(1u8);
        }
        x
    }
}
```

# Generics

Let's see a more complex example:

*Rust*

```rust
#[derive(Debug)]
struct FixArray<T, const N: usize>
where
    T: std::ops::AddAssign,
    T: From<u8>,
    T: Copy
{

}
```

```rust
impl<T, const N: usize> FixArray<T, N>
where
    T: std::ops::AddAssign,
    T: From<u8>,
    T: Copy
{
    fn new(value: T) -> Self {
ray::<T, N> {
lements: [value; N],
utive(start: T) -> Self {
ut x: FixArray<T, N> = FixArray::new(start);
ut temp = start;
 in &mut x.elements {
i = temp;
```

*Rust*

```rust
fn main() {
    let a: FixArray<i32, 5> = FixArray::new(1);
    let b: FixArray<u8, 7> = FixArray::consecutive(10);
    println!("{:?}", a);
    println!("{:?}", b);
}
```

**Output**

```
FixArray { elements: [1, 1, 1, 1, 1] }
FixArray { elements: [10, 11, 12, 13, 14, 15, 16] }
```

# Generics

Rust generics can **conditionally** implement some traits using the **where** keyword:

*Rust*

```rust
struct MyStruct<T> {
    data: T,
}
trait OddNumber {
    fn is_odd(&self) -> bool;
}
impl OddNumber for i32 {
    fn is_odd(&self) -> bool { (*self % 2) == 1 }
}
impl<T> OddNumber for MyStruct<T>
where
    T: OddNumber,
{
    fn is_odd(&self) -> bool { self.data.is_odd() }
}
fn main() {
    let x: MyStruct<i32> = MyStruct { data: 5 };
    println!("x is odd ? => {}", x.is_odd());
}
```

**Output**

x is odd ? => true

This tells the compiler to implement the trait OddNumber over **MyStruct<T>** only if T also implements ***OddNumber*** trait.

# Generics

Rust generics can **conditionally** implement some traits using the **where** keyword:

*Rust*

```rust
struct MyStruct<T> {
    data: T,
}
trait OddNumber {
    fn is_odd(&self) -> bool;
}
impl OddNumber for i32 {
    fn is_odd(&self) -> bool { (*self % 2) == 1 }
}
impl<T> OddNumber for MyStruct<T>
where
    T: OddNumber,
{
    fn is_odd(&self) -> bool { self.data.is_odd() }
}
fn main() {
    let x: MyStruct<i32> = MyStruct { data: 5 };
    println!("x is odd ? => {}", x.is_odd());
}
```

**Output**

```
x is odd ? => true
```

Since **i32** implements OddNumber, so will ***MyStruct<i32>*** implement OddNumber as well. As a result, we can call x.is_odd().

# Generics

Rust generics can **conditionally** implement some traits using the **where** keyword:

*Rust*

```
struct MyStruct<T> {
    data: T,
}
trait OddNumber {
    fn is_odd(&self) -> bool;
}
impl OddNumber for i32 {
    fn is_odd(&self) -> bool { (*self % 2) == 1 }
}
impl<T> OddNumber for MyStruct<T>
where
    T: OddNumber,
{
    fn is_odd(&self) -> bool { self.data.is_odd() }
}
fn main() {
    let x: MyStruct<u32> = MyStruct { data: 5 };
    println!("x is odd ? => {}", x.is_odd());
}
```

***Error***

```
error[E0599]: the method `is_odd` exists for struct `MyStruct<u32>`, but
its trait bounds were not satisfied
  --> src\main.rs:22:36
   |
1  | struct MyStruct<T> {
   | ------------------
   | |
   | method `is_odd` not found for this struct
   | doesn't satisfy `MyStruct<u32>: OddNumber`
...
22 |     println!("x is odd ? => {}", x.is_odd());
   |                                    ^^^^^^ method cannot be called on
`MyStruct<u32>` due to unsatisfied trait bounds
   |
```

In this case, **we can not call** x.is_odd() as the trait ***OddNumber*** was not implemented over MyStruct<u32> due to the fact that **u32** does not implement ***OddNumber***.

# Generics

Rust generics can **conditionally** implement some traits using the **where** keyword:

*Rust*

```rust
struct MyStruct<T> {
    data: T,
}
trait OddNumber {
    fn is_odd(&self) -> bool;
}
impl OddNumber for i32 {
    fn is_odd(&self) -> bool { (*self % 2) == 1 }
}
impl<T> OddNumber for MyStruct<T>
where
    T: OddNumber,
{
    fn is_odd(&self) -> bool { self.data.is_odd() }
}
fn main() {
    let x: MyStruct<u32> = MyStruct { data: 5 };
    println!("x = {}",x.data);
}
```

**Output**

```
5
```

Notice that this code compiles correctly. This is because ***OddNumber*** is only implemented for MyStruct<T> if only if T implements ***OddNumber***. Otherwise, the trait is is not implemented and if is_odd method is not called the code compiles.

# Generics

This conditional implementation of traits for generics allows Rust to add a specific behavior whenever #[derive] is being used over a generic. Let's analyze the following case:

*Rust*

```rust
fn main() {
    let x: Option<i32> = Some(1);
    let y = x;
    println!("x = {:?}, y = {:?}",x,y);
}
```

**Output**

x = Some(1), y = Some(1)

*Rust*

```rust
fn main() {
    let x: Option<String> = Some("123".to_string());
    let y = x;
    println!("x = {:?}, y = {:?}",x,y);
}
```

***Error***

```
error[E0382]: borrow of moved value: `x`
 --> src\main.rs:4:35
  |
2 |     let x: Option<String> = Some("123".to_string());
  |         - move occurs because `x` has type `Option<String>`, which
           does not implement the `Copy` trait
3 |     let y = x;
  |             - value moved here
4 |     println!("x = {:?}, y = {:?}",x,y);
  |                                   ^ value borrowed here after move
```

# Generics

So … why the case with Option<i32> works, but the case with Option<String> does not ?

Let's see how Option is defined …

*Rust (option.rs)*

```
#[derive(Copy, PartialOrd, Eq, Ord, Debug, Hash)]
#[rustc_diagnostic_item = "Option"]
#[stable(feature = "rust1", since = "1.0.0")]
pub enum Option<T> {
    /// No value.
    #[lang = "None"]
    #[stable(feature = "rust1", since = "1.0.0")]
    None,
    /// Some value of type `T`.
    #[lang = "Some"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

# Generics

So … why the case with Option<i32> works, but the case with Option<String> does not ?

Let's see how O

*Rust (option.rs)*

```
#[derive(Copy,
#[rustc_diagnosti
#[stable(feature =
pub enum Option<T>
    /// No value.
    #[lang = "None"]
    #[stable(feature = "rust1", since = "1.0.0")]
    None,
    /// Some value of type `T`.
    #[lang = "Some"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

As you can see, Option generic implements the Copy trait. The #derive will add generate a code in the following format:

```
impl<T> Copy for Option<T> Where T: Copy {…}
```

Since i32 implements **Copy**, so does Option<i32>. Similarly, since String **does not** implement **Copy**, Option<String> will not implement **Copy**, thus explaining the different behavior in the the two presented cases.

# Match

# Match

Rust has a specific keyword (match) designed for complex and efficient value matching against various patterns. This is similar to the switch keyword from C++, however it is more complex and can perform more complex matches.

The general format for a match is:

```
match value {
    rule1,
    rule2,
    ….
    rulen,
}
```

where ***rule***$_i$ has the following format:

- pattern => code
- pattern => { code }
- pattern **if** condition => code
- pattern **if** condition => { code }

"***pattern if condition***" is also called a match guard

A ***rule*** in a match is often called an **arm** of the pattern matching !

# Match

There are a couple of constraints that need to be followed for a match to be correct:

1. **At least one rule** must be provided to a match construct
2. **All possible values must be covered** by the existing rules provided to a match construct
3. **No overlapping rules**. There can not be two rules in the same match construct that match the same value.

Character underline ('_') used as a pattern has a special meaning : ***everything else***. It is similar to the usage of default keyword in a C++ switch statement.

# Match

The pattern used in a match constructs also has multiple forms:

1. A single constant value (e.g. a number, a string, etc)
2. Multiple constant values, separated by '|' operator → (e.g. 1 | 2 | 3)
3. A range → (e.g. 1..=5)
4. An enum
5. An array
6. A slice
7. A tuple
8. A struct
9. A pointer or a reference

# Match

Let’s see a very simple example:

*Rust*

```
fn main() {
    for x in 1..10 {
        match x {
            1 => println!("one"),
            2 => println!("two"),
            3 => println!("three"),
            4 => println!("four"),
            5 => println!("five"),
            _ => println!("Another number"),
        }
    }
}
```

**Output**

```
one
two
three
four
five
Another number
Another number
Another number
Another number
```

# Match

Let's see a very simple example:

*Rust*

```
fn main() {
    for x in 1..10 {
        match x {
            1 => println!("one"),
            2 => println!("two"),
            3 => println!("three"),
            4 => println!("four"),
            5 => println!("five"),
        }
    }
}
```

***Error***

```
error[E0004]: non-exhaustive patterns: `i32::MIN..=0_i32` and `6_i32..=i32::MAX`
not covered
 --> src\main.rs:3:15
  |
3 |         match x {
  |               ^ patterns `i32::MIN..=0_i32` and `6_i32..=i32::MAX` not covered
  |
```

Notice that we have removed the arm/rule "`_ => println!("Another number")`". As a result, not all possible cases are covered, and a compile error is thrown. The error also provides a list of values that were missed (values from i32::MIN to 0 and values from 6 to i32::MAX)

# Match

Let’s see a very simple example:

*Rust*

```rust
fn main() {
    let text = "three";
    let value: i32;
    match text {
        "one" => value = 1,
        "two" => value = 2,
        "three" => value = 3,
        "four" => value = 4,
        _ => value = 0,
    }
    println!("value = {value}");
}
```

**Output**

```
value = 3
```

Notice that we can also match strings.

# Match

As a general rule, in case of strings, you have to check every one of them – so the usual complexity is O(n). However, in some cases, Rust can make some optimization:

*Rust (1.73.0 – optimized)*

```
fn foo(x: &str) -> u32 {
    match x {
        "one" => 1,
        "two" => 2,
        "three" => 3,
        "four" => 4,
        _ => 0,
    }
}
```

In this case Rust first **checks the length** and then the values. For example, if the string has a length of 4 bytes, it only needs to check it against the string "four". Furthermore, if the string does not have the length 3,4 or 5 than it return 0 (the default case).

```
foo:
        cmp     rsi, 5 // rsi = length of x
        je      has_length_of_5
        cmp     rsi, 4
        je      has_length_of_4
        xor     eax, eax
        cmp     rsi, 3
        jne     default case
```

# Match

As a general rule, in case of strings, you have to check every one of them – so the usual complexity is O(n). However, in some cases, Rust can make some optimization:

*Rust (1.73.0 – optimized)*

```rust
fn foo(x: &str) -> u32 {
    match x {
        "abcd" => 5,
        "ghij" => 15,
        "klmn" => 25,
        _ => 100,
    }
}
```

```asm
foo:
        mov     eax, 100
        cmp     rsi, 4
        jne     SIZE_IS_NOT_4
        cmp     dword ptr [rdi], 1684234849
        je      TEXT_IS_abcd
        cmp     dword ptr [rdi], 1785292903
        je      TEXT_IS_ghij
        cmp     dword ptr [rdi], 1852664939
        setne   al
        test    al, 1
        mov     ecx, 100
        mov     eax, 25
        cmovne  eax, ecx
SIZE_IS_NOT_4:
        ret
TEXT_IS_abcd:
        mov     eax, 5
        ret
TEXT_IS_ghij:
        mov     eax, 15
        ret
```

In this case, since all strings have size 4, and 4 bytes actually represent an u32, the solution is to check the value directly. Simply put, the u32 with value 1684234849 is in fact the u32 value of a memory with 4 bytes that are “abcd”

# Match

You can initialize any kind of variable in a match construct:

*Rust*

```
struct Point {x: i32, y: i32}
fn main() {
    let text = "three";
    let p: Point;
    match text {
        "one" => p = Point{x:1,y:1},
        "two" => p = Point{x:2,y:2},
        "three" => p = Point{x:3,y:3},
        "four" => p = Point{x:4,y:14},
        _ => p = Point{x:0,y:0},
    }
    println!("P = ({},{})",p.x, p.y);
}
```

**Output**

```
P = (3,3)
```

In this we initialize variable “p” that is of type Point.

# Match

When initializing a variable through a match construct, that variable **MUST** be initialized in all rules/match arms !

*Rust*

```
struct Point {x: i32, y: i32}
fn main() {
    let text = "three";
    let p: Point;
    match text {
        "one" => p = Point{x:1,y:1},
        "two" => p = Point{x:2,y:2},
        "three" => p = Point{x:3,y:3},
        "four" => p = Point{x:4,y:14},
        _ => {},
    }
    println!("P = ({},{})",p.x, p.y);
}
```

*Error*

```
error[E0381]: used binding `p` is possibly-uninitialized
  --> src\main.rs:12:28
   |
4  |     let p: Point;
   |         - binding declared here but left uninitialized
5  |     match text {
6  |         "one" => p = Point{x:1,y:1},
   |                  ------------------ binding initialized here in some conditions
7  |         "two" => p = Point{x:2,y:2},
   |                  ------------------ binding initialized here in some conditions
8  |         "three" => p = Point{x:3,y:3},
   |                    ------------------ binding initialized here in some conditions
9  |         "four" => p = Point{x:4,y:14},
   |                   ------------------- binding initialized here in some conditions
...
12 |     println!("P = ({},{})",p.x, p.y);
   |                            ^^^ `p.x` used here but it is possibly-uninitialized
   |
```

Notice that the default case '_' does not initialize variable "p"

# Match

Let's see a match construct that uses the ***bool*** type.

*Rust*

```
fn main() {
    let v = true;
    match v {
        true => println!("Value is true"),
        false => println!("Value is false")
    }
}
```

**Output**

Value is true

Notice that there is no need for the rule/arm "_" (the default case) if all possible cases are already covered ! (in case of a bool variable this means the case of value ***true*** and the case of value ***false***).

# Match

However, it is important to know that using the default case "_" when all possible cases are already covered will not trigger an error but a **warning**.

*Rust*

```
fn main() {
    let v = true;
    match v {
        true => println!("Value is true"),
        false => println!("Value is false"),
        _ => println!("something else")
    }
}
```

**Output**

```
Value is true
```

***Warning***

```
warning: unreachable pattern
 --> src\main.rs:6:9
  |
6 |         _ => println!("something else")
  |         ^
  |
  = note: `#[warn(unreachable_patterns)]` on by default
```

# Match

Another particular case are floating point values. While right now they are allowed in a match construct, they are going to be forbidden in future releases due to high complexity around comparing floating point values (e.g. including NaN values).

*Rust*

```rust
fn main() {
    let v = 1.2;
    match v {
        1.2 => println!("Found 1.2"),
        1.1 => println!("Found 1.1"),
        0.0 => println!("Found 0"),
        f64::NAN => println!("found NAN"),
        _ => println!("another value")
    }
}
```

**Output**

```
Found 1.2
```

***Warning***

```
warning: floating-point types cannot be used in patterns
 --> src\main.rs:4:9
  |
4 |         1.2 => println!("Found 1.2"),
  |         ^^^
  |
  = note: `#[warn(illegal_floating_point_literal_pattern)]` on by default
  = warning: this was previously accepted by the compiler but is being phased
out; it will become a hard error in a future release!
```

More info on this topic on: [https://github.com/rust-lang/rust/issues/41620](https://github.com/rust-lang/rust/issues/41620)

# Match

To match multiple values, use the `|` operator like in the following example:

*Rust*

```rust
fn main() {
    let v = 19;
    match v {
        2            => println!("An odd prime number"),
        3|5|7        => println!("Prime numbers under 10"),
        11|13|17|19 => println!("Prime numbers under 20"),
        _            => println!("Another value")
    }
}
```

**Output**

```
Prime numbers under 20
```

The same logic where every possible value has to be matched by one of the rules has to be present in this case as well (this is why we need the final '_' (default) rule).

# Match

However, it is possible to duplicate a value when using the OR ('|') operator. The code will compile and will use the match rule that first uses that value. At the same time, a warning will be thrown to explain that the second value is unreachable.

*Rust*

```rust
fn main() {
    let v = 19;
    match v {
        2           => println!("An odd prime number"),
        3|5|7|19    => println!("Prime numbers under 10"),
        11|13|17|19 => println!("Prime numbers under 20"),
        _           => println!("Another value")
    }
}
```

**Output**

Prime numbers under 10

***Warning***

```
warning: unreachable pattern
 --> src\main.rs:6:18
  |
6 |         11|13|17|19=> println!("Prime numbers under 20"),
  |                  ^^
  |
  = note: `#[warn(unreachable_patterns)]` on by default
```

In this case, the second 19 value is considered an unreachable pattern. Notice that the output now is "***Prime numbers under 10***" as the match is done for the rule that first uses 19 (rule with 3|5|7|19)

# Match

It is also possible to match entire intervals (by using the operator ..=). However, keep in mind that matching an entire interval (or several of them) is not always a simple job (if you want to improve matching performance).

*Rust*

```rust
fn main() {
    let grade = 5;
    match grade {
        1..=4 => println!("Class failed"),
        5..=10 => println!("Class passed"),
        _ => println!("Invalid grade"),
    }
}
```

**Output**

```
Class passed
```

As such, it is preferred to use inclusive intervals (a..=b) rather than exclusive ones.

# Match

Partial intervals can be used (notice the 5.. usage in the next code).

*Rust*

```
fn main() {
    let grade = 5;
    match grade {
        1..=4 => println!("Class failed"),
        5.. => println!("Class passed"),
        _ => println!("Invalid grade"),
    }
}
```

**Output**

```
Class passed
```

Keep in mind that default value "_" has to be used in this case to cover all possible cases (for example value 0 or negative values).

# Match

Overlapping intervals are also possible !

*Rust*

```
fn main() {
    let grade = 5;
    match grade {
        1..=4  => println!("Class failed"),
        0..=10 => println!("Class passed"),
        _      => println!("Invalid grade"),
    }
}
```

**Output**

```
Class passed
```

**OBS**: *Keep in mind that intervals are hard to optimize and that the goal of a matcher is to obtain an O(1) access/check time.*

# Match

For range/interval-based rules only numerical and char values and patterns can be used. As such constructs like the next one that attempt to match intervals based on strings are not possible !

*Rust*

```rust
fn main() {
    let name = "John";
    match name {
        "abc"..="zzz" => println!("Interval one"),
        " "..="aaa" => println!("Interval two"),
        _ => println!("Another interval"),
    }
}
```

*Error*

```
error[E0029]: only `char` and numeric types are allowed in range patterns
 --> src\main.rs:4:9
  |
4 |         "abc"..="zzz" => println!("Class failed"),
  |         -----^^^-----
  |         |       |
  |         |       this is of type `&'static str` but it should be `char` or numeric
  |         this is of type `&'static str` but it should be `char` or numeric
```

# Match

Match constructs are often used with **enums**. Notice that since an **enum** has a finite set of possible values, the default "_" case is not needed.

*Rust*

```
enum Color {
    Red,
    Green,
    Blue,
    Black,
}
fn main() {
    let c = Color::Red;
    match c {
        Color::Red   => println!("Red"),
        Color::Green => println!("Green"),
        Color::Blue  => println!("Blue"),
        Color::Black => println!("Black"),
    }
}
```

**Output**

```
Red
```

# Match

If a specific case from an **enum** is not covered, the Rust compiler can also provide insights into what is missing and what needs to be added.

*Rust*

```rust
enum Color {
    Red,
    Green,
    Blue,
    Black,
}
fn main() {
    let c = Color::Red;
    match c {
        Color::Red   => println!("Red"),
        Color::Green => println!("Green"),
        Color::Blue  => println!("Blue"),
    }
}
```

*Error*

```
error[E0004]: non-exhaustive patterns: `Color::Black` not covered
 --> src\main.rs:9:11
  |
9 |     match c {
  |           ^ pattern `Color::Black` not covered
  |
note: `Color` defined here
 --> src\main.rs:5:5
  |
1 | enum Color {
  |      -----
...
5 |     Black,
  |     ^^^^^ not covered
  = note: the matched value is of type `Color`
help: ensure that all possible cases are being handled by adding a
match arm with a wildcard pattern or an explicit pattern as shown
  |
12~         Color::Blue => println!("Blue"),
13~         Color::Black => todo!(),
  |
```

# Match

Let’s discuss another case that involves a match construct and an enum.

Let’s consider library “A” that exports an enum with a list of possible errors defined like in the following way:

```
enum Error { Format, IO, Parameters }
```

Let’s also consider application “B” that uses library “A” as a dependency and has a code that matches the errors from library (crate) “A”:

```
match error {
      Error::Format      => {…},
      Error::IO          => {…},
      Error::Parameters  => {…},
}
```

# Match

What happens if library "A" decides to add a new value in the error list ?

```
enum Error { Format, IO, Parameters, Critical }
```

The immediate result will be that application "B" will not compile anymore:

```
match error {
      Error::Format      => {…},
      Error::IO          => {…},
      Error::Parameters  => {…},
}
```

Rust will show an error because this match construct from application "B" does not match all possible values of Error.

So … what are the solutions in this case ?

# Match

**So … what are the solutions in this case ?**

1. Application "B" has to refactor its code to match the new constraints from library "A". This is possible, but if library "A" changes its enum often, this might be an issue for the developers of application "A"

2. Library "A" uses the non-exhaustive attribute for their **enum**:

```
#[non_exhaustive]
enum Error { Format, IO, Parameters }
```

This flag will force the compiler to explicitly request that application "A" adds the default case (_) on every match even if all cases are already treated. This will however make sure that if newer versions of library "A" adds new variants to the enum, they will be treated application "B".

# Match

**So … what are the solutions in this case ?**

2. Library “A” uses the non-exhaustive attribute for their **enum**:

```
#[non_exhaustive]
enum Error { Format, IO, Parameters }
```

This flag will force the compiler to explicitly request that application “A” adds the default case (_) on every match even if all cases are already treated.

As such, the code from application “B” will be changed into something like this:

```
match error {
      Error::Format      => {…},
      Error::IO          => {…},
      Error::Parameters  => {…},
      _                  => { <default processing for future errors> }
}
```

# Match

Let's try a more complex **enum** (one that contains specific values as well). You will notice that this code does not compile as it requires as to specify the value as well.

*Rust*

```
enum Distance {
    Inch(i32),
    Cm(i32),
    Km(i32)
}
fn main() {
    let d = Distance::Km(100);
    match d {
        Distance::Inch => println!("Distance is in inch"),
        Distance::Cm => println!("Distance is in cm"),
        Distance::Km => println!("Distance is in km"),
    }
}
```

*Error*

```
error[E0532]: expected unit struct, unit variant or constant, found tuple variant `Distance::Inch`
 --> src\main.rs:9:9
  |
2 |     Inch(i32),
  |     --------- `Distance::Inch` defined here
...
9 |         Distance::Inch => println!("Distance is in inch"),
  |         ^^^^^^^^^^^^^^ help: use the tuple variant pattern syntax instead: `Distance::Inch(_)`
```

# Match

In this context, the underline character '_' means to match any value as long as the variant type is the one specified. While this code is correct, we might want to get the actual value as well (e.g. in out case value 100)

*Rust*

```
enum Distance {
    Inch(i32),
    Cm(i32),
    Km(i32)
}
fn main() {
    let d = Distance::Km(100);
    match d {
        Distance::Inch(_) => println!("Distance is in inch"),
        Distance::Cm(_)   => println!("Distance is in cm"),
        Distance::Km(_)   => println!("Distance is in km"),
    }
}
```

**Output**

Distance is in km

# Match

The general format to get the actual value associated with an enum variant is:

enum::variant(variable_name) => {code}

*Rust*

```
enum Distance {
    Inch(i32),
    Cm(i32),
    Km(i32)
}
fn main() {
    let d = Distance::Km(100);
    match d {
        Distance::Inch(value) => println!("Distance is {} inch", value),
        Distance::Cm(value)   => println!("Distance is {} cm", value),
        Distance::Km(value)   => println!("Distance is {} km", value),
    }
}
```

**Output**

Distance is 100 km

# Match

We can also match an exact value associated with a variant (in this case for variant Km we have two rules/match-arms:

- **Km(90)** that will match cases where Distance type is Km and its value is 90
- **Km(value)** that will match the rest of the cases where Distance type is Km

*Rust*

```rust
enum Distance {
    Inch(i32),
    Cm(i32),
    Km(i32)
}
fn main() {
    let d = Distance::Km(100);
    match d {
        Distance::Inch(value) => println!("Distance is {} inch", value),
        Distance::Cm(value)   => println!("Distance is {} cm", value),
        Distance::Km(90)      => println!("Particular case (90 Km)"),
        Distance::Km(value)   => println!("Distance is {} km", value),
    }
}
```

**Output**

```
Distance is 100 km
```

# Match

Match constructs are often used with Option<…> and Result<…> enums to match the two cases those two enums can provide.

*Rust*

```
fn get_odd_number(value: u32)->Option<u32> {
    if value % 2 != 0 {
        Some(value)
    } else {
        None
    }
}
fn main() {
    let d = get_odd_number(11);
    match d {
        None => println!("Not an odd number"),
        Some(x) => {
            println!("this is an odd number {}",x);
        }
    }
}
```

**Output**

this is an odd number 11

# Match

Similar to how enums matching works, a structure and its value can be matched in a match construct. In this case , the underline character "_" has a special meaning (it implies that a specific field from a structure can have whatever value we want, and we don't need to know its value).

Let's consider that we have a 2-dimensional screen of size 100 x 100 where points can be written.

Each point is represented by a structure with two coordonates (X and Y) and we are interested in knowing if a point is:

- The origin (0,0)
- On one of the margins (X = 0 or X = 100 or Y = 0 or Y = 100)
- The center (50,50)

# Match

Let’s see how we can use a match construct to build this rules:

*Rust*

```rust
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let p = Point { x: 0, y: 0 };
    match p {
        Point{x:0,y:0}   => println!("Origin"),
        Point{x:50,y:50} => println!("Center"),
        Point{x:0,y:_}   => println!("Bottom margin"),
        Point{x:100,y:_} => println!("Top margin"),
        Point{x:_,y:0}   => println!("Left margin"),
        Point{x:_,y:100} => println!("Right margin"),
        _                => println!("Other point")
    }
}
```

**Output**

Origin

# Match

Let’s see how we can use a match construct to build this rules:

*Rust*

```
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let p = Point { x: 0, y: 0 };
    match p {
        Point{x:0,y:0}   => println!("Origin"),
        Point{x:50,y:50} => println!("Center"),
        Point{x:0,y:_}   => println!("Bottom margin"),
        Point{x:100,y:_} =>
        Point{x:_,y:0}
        Point{x:_,y:100} =>
        _                => println!("Other point")
    }
}
```

**Output**

```
Origin
```

This matches any Point that has int Y-axes set up to value 0

# Match

It is also possible to associate the name of the field to its value by NOT specifying the expected value (e.g instead of "***x:<value>***" or "***x:_***" just use "x")

*Rust*

```rust
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let p = Point { x: 0, y: 23 };
    match p {
        Point{x:0,y:0}   => println!("Origin"),
        Point{x:50,y:50} => println!("Center"),
        Point{x:0,y}     => println!("Bottom margin (0,{})",y),
        Point{x:100,y:_} => println!("Top margin"),
        Point{x:_,y:0}   => println!("Left margin"),
        Point{x:_,y:100} => println!("Right margin"),
        _                => println!("Other point")
    }
}
```

**Output**

```
Bottom margin (0,23)
```

# Match

It is also possible to associate the name of the field to its value by NOT specifying the expected value (e.g instead of “***x:<value>***” or “***x:_***” just use “x”)

*Rust*

```
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let p = Point { x: 0, y: 23 };
    match p {
        Point{x:0,y:0}   => println!("Origin"),
        Point{x:50,y:50} => println!("Center"),
        Point{x:0,y}     => println!("Bottom margin (0,{})",y),
        Point{x:100,y:_} => println!("Top margin"),
        Point{x:_,y:0}   => println!("Left margin"),
        Point{x:_,y:100} => println!("Right margin"),
                         => println!("Other point"),
```

**Output**

Bottom margin (0,23)

Notice that we have not specified a value for the field “y” (in the form of y:value). As such, it is considered that any value can match “Point::y” and that value is linked/bounded to the variable “y” that can further be used in the rule code.

# Match

It is also possible to associate the name of the field to its value by NOT specifying the expected value (e.g instead of “***x:<value>***” or “***x:_***” just use “x”)

*Rust*

```
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let p = Point { x: 0, y: 23 };
    match p {
        Point{x:0,y:0}    => println!("Origin"),
        Point{x:50,y:50} => println!("Center"),
        Point{x:0,y}      => println!("Bottom margin (0,{})",y),
        Point{x:100,y:_} => println!("Top margin"),
        Point{x:_,y:0}    => println!("Left margin"),
```

**Output**

Bottom margin (0,23)

A similar result can be obtained if we use an alias:

```
Point{x:0,y:my_var} => println!("Bottom margin (0,{})",my_var),
```

# Match

If a structure has multiple parameters and you are interested in matching only one of them, you can use “..” to specify that the rest of them should be ignored.

*Rust*

```
struct Point4D {
    x: i32,
    y: i32,
    z: i32,
    t: i32
}
fn main() {
    let p = Point4D { x: 5, y: 23, z:15, t:30 };
    match p {
        Point4D{x:0,y:0,z:0,t:0}     => println!("Origin"),
        Point4D{x:50,y:50,z:50,t:50} => println!("Center"),
        Point4D{x:5, ..}             => println!("Some point with x=5"),
        _                            => println!("Other point")
    }
}
```

**Output**

Some point with x = 5

# Match

If a structure has multiple parameters and you are interested in matching only one of them, you can use ".." to specify that the rest of them should be ignored.

*Rust*

```
struct Point4D {
    x: i32,
    y: i32,
    z: i32,
```

A similar result can be obtained if we use an alias:

```
Point4D{x:5,y:_,z:_,t:_} => println!("Some point with x=5"),
```

or

```
Point4D{x:5,y,z,t} => println!("Some point with x=5"),
```

```
        Point4D{x:0,y:0,z:0,t:0}     => println!("Origin"),
        Point4D{x:50,y:50,z:50,t:50} => println!("Center"),
        Point4D{x:5, ..}             => println!("Some point with x=5"),
        _                            => println!("Other point")
    }
}
```

**Output**

Some point with x = 5

# Match

A similar logic to what happens in case of a structure can be applied any tuple. Let's consider the previous example with Point4D defined as a tuple:

*Rust*

```
fn main() {
    // A Point4D with tuples (X,Y,Z,T)
    let p = (0,0,0,0);
    match p {
        (0,0,0,0)  => println!("Origin"),
        (0,..)     => println!("X-axis is 0 "),
        (1,2,..)   => println!("X axis is 0 and Y axis is 2"),
        (_,_,_,10) => println!("T axis is 10"),
        (9,y,z,_)  => println!("X is 0, Y = {}, z = {}, t is ignored",y,z),
        (7,..,t)   => println!("X is 7, X and Y are ignored, T is {}",t),
        _          => println!("Other point")
    }
}
```

**Output**

```
Origin
```

# Match

A similar logic to what happens in case of a structure can be applied any tuple. Let's consider the previous example with Point4D defined as a tuple:

*Rust*

```
fn main() {
    // A Point4D with tuples (X,Y,Z,T)
    let p = (7,6,5,4);
    match p {
        (0,0,0,0)  => println!("Origin"),
        (0,..)     => println!("X-axis is 0 "),
        (1,2,..)   => println!("X axis is 0 and Y axis is 2"),
        (_,_,_,10) => println!("T axis is 10"),
        (9,y,z,_)  => println!("X is 0, Y = {}, z = {}, t is ignored",y,z),
        (7,..,t)   => println!("X is 7, X and Y are ignored, T is {}",t),
        _          => println!("Other point")
    }
}
```

**Output**

X is 7, X and Y are ignored, T is 4

You can use ".." or "_" to ignore one or multiple values. At the same time, using a variable name will assign the value from the tuple to that variable for usage in the rule code

# Match

A similar logic to what happens in case of a structure can be applied any tuple. Let's consider the previous example with Point4D defined as a tuple:

*Rust*

```rust
fn main() {
    // A Point4D with tuples (X,Y,Z,T)
    let p = (2,0,0,10);
    match p {
        (0,0,0,0)  => println!("Origin"),
        (0,..)     => println!("X-axis is 0 "),
        (1,2,..)   => println!("X axis is 0 and Y axis is 2"),
        (_,_,_,10) => println!("T axis is 10"),
        (9,y,z,_)  => println!("X is 0, Y = {}, z = {}, t is ignored",y,z),
        (7,..,t)   => println!("X is 7, X and Y are ignored, T is {}",t),
```

**Output**

T axis is 10

A similar result can be obtained if we use the ".." operator:

```rust
(..,10) => println!("T axis is 10"),
```

# Match

Keep in mind that the order of the rules matters. If two rules match the same case, the first one (in terms of the definition order) will be used. In our case, (0,0,0,10) matches both (0,..) and (_,_,_,10). However, the result will be the first one that its being matched → (0,..)

*Rust*

```
fn main() {
    // A Point4D with tuples (X,Y,Z,T)
    let p = (0,0,0,10);
    match p {
        (0,0,0,0)   => println!("Origin"),
        (0,..)      => println!("X-axis is 0 "),
        (1,2,..)    => println!("X axis is 0 and Y axis is 2"),
        (_,_,_,10) => println!("T axis is 10"),
        (9,y,z,_)   => println!("X is 0, Y = {}, Z = {}, t is ignored",y,z),
        (7,..,t)    => println!("X is 7, X and Y are ignored, T is {}",t),
        _           => println!("Other point")
    }
}
```

**Output**

X-axis is 0

# Match

Just like in the case of tuples, a similar logic can be applied for array as well. Lets consider the following example:

*Rust*

```
fn main() {
    let a = [1,2,3,4];
    match a {
        [0,0,0,0]  => println!("A vector with Zeros"),
        [0,..]     => println!("A vector with first element Zero"),
        [..,5]     => println!("A vector with last element 5"),
        [2,..,3]   => println!("A vector with first element 2 and last one 3"),
        [_,2,_,4]  => println!("A vector with second element 2 and last one 4"),
        _          => println!("Something else")
    }
}
```

**Output**

A vector with second element 2 and last one 4

**OBS**: *Just like in the previous cases, .. can be used to match multiple consecutive elements.*

# Match

You can also capture the value of an element from the array, and/or combine this method with the usage of _ and .. to ignore one or multiple values.

*Rust*

```
fn main() {
    let a = [0,2,3,4];
    match a {
        [0,0,0,0]  => println!("A vector with Zeros"),
        [0,x,y,z]  => println!("A vector with [0,{},{},{}]",x,y,z),
        [..,l,5]   => println!("A vector with last elements: {} and 5",l),
        [_,m,..]   => println!("A vector with the second element {}",m)
    }
}
```

**Output**

A vector with [0,2,3,4]

OBS: Notice that we don't need the final rule/match-arm for the default value (_). This is because **[_,m,..]** will match everything else and will provide the value of the second parameter in variable "m"

# Match

Keep in mind that in case of arrays, the number of elements described in each rule must match the number of elements in the array. In this next example, the second rule/match-arm has 3 elements instead of 4 (the number of elements from "a")

*Rust*

```
fn main() {
    let a = [0,2,3,4];
    match a {
        [0,0,0,0]  => println!("A vector with Zeros"),
        [0,x,y]    => println!("A vector with [0,{},{}]",x,y),
        _          => println!("Something else")
    }
}
```

***Error***

```
error[E0527]: pattern requires 3 elements but array has 4
 --> src\main.rs:5:9
  |
5 |         [0,x,y]    => println!("A vector with [0,{},{}]",x,y),
  |         ^^^^^^^ expected 4 elements
```

# Match

While in case of arrays, the number of elements must be matched by all patterns, in case of slices, there is no such rule. Let's analyze the next example:

*Rust*

```
fn check_slice(slice: &[u8]) {
    println!("Testing: {:?}",slice);
    match slice {
        [_] => { println!("  Match: a slice with one element "); },
        [a,0] => { println!("  Match: two elements (first is {a}, last is 0)"); },
        [a,b] => { println!("  Match: two elements ({a} and {b})"); },
        [a,_,b] => { println!("  Match: three elements: first:{a}, last:{b}"); },
        [1,..,5] => { println!("  Match: starts with 1 and ends with 5"); }
        _ => { println!("  Match: other cases "); }
    }
}
fn main() {
    let x = [1u8,0,5];
    check_slice(&x);
    check_slice(&x[..1]);
    check_slice(&x[..2]);
}
```

Notice that we have called function check_slice with slices of various size (3,1 and 2 elements)

**Output**

```
Testing: [1, 0, 5]
  Match: three elements: first:1, last:5
Testing: [1]
  Match: a slice with one element
Testing: [1, 0]
  Match: two elements (first is 1, last is 0)
```

# Match

While in case of arrays, the number of elements must be matched by all patterns, in case of slices, there is no such rule. Let's analyze the next example:

*Rust*

```
fn check_slice(slice: &[u8]) {
    println!("Testing: {:?}",slice);
    match slice {
        [_] => { println!("  Match: a slice with one element "); },
        [a,0] => { println!("  Match: two elements (first is {a}, last is 0)"); },
        [a,b] => { println!("  Match: two elements ({a} and {b})"); },
        [a,_,b] => { println!("  Match: three elements: first:{a}, last:{b}"); },
        [1,..,5] => { println!("  Match: starts with 1 and ends with 5"); }
        _ => { println!("  Match: other cases "); }
    }
}
fn main() {
    let x = [1u8,0,5];
    check_slice(&x);
    check_slice(&x[..1]);
    check_slice(&x[..2]);
}
```

Notice that there are two arms that match the slice [1,0,5]. Rust will stop at the first match (so the order of the rules matters).

# Match

It is also possible to capture a slice from the array by using the sigil character (@) to bind a part of the array into a new slice that can be used in the rule/arm code.

In case of arrays, the general format for this type of binding is:

<variable_name> @ ..

Keep in mind that <variable_name> @ _ (to bind a variable name to a single position) is also possible but unnecessary as you can use the variable name directly.

*Rust*

```
fn main() {
    let a = [0,1,2,3,4];
    match a {
        [0,0,0,0,0]          => println!("A vector with Zeros"),
        [0,middle @..,0]     => println!("Middle of the vector is: {:?}",middle),
        [first_3 @ ..,3,4]  => println!("First three elements: {:?}", first_3),
        _                    => println!("Something else")
    }
}
```

**Output**

First three elements: [0, 1, 2]

# Match

However, binding a value with a sigil character (@) is useful for cases where that value is not stored in a local variable (but it is the result of an expression). Let's consider the following case:

*Rust*

```
fn get_a_random_value() -> u8 { rand::random::<u8>() % 101u8 }
fn main() {
    match get_a_random_value() {
        0       => println!("Zero"),
        1..=49  => println!("Less than half"),
        50..=99 => println!("Better than half"),
        100     => println!("100"),
        _       => println!("Impossible value")
    }
}
```

**Output (possible)**

```
Better than half
```

Notice that the result of get_a_random_value() function is not stored in function main. As such if we want to use the actual value in one of the match construct arms, we can't.

# Match

However, binding a value with a sigil character (@) is useful for cases where that value is not stored in a local variable (but it is the result of an expression). Let's consider the following case:

*Rust*

```rust
fn get_a_random_value() -> u8 { rand::random::<u8>() % 101u8 }
fn main() {
    match get_a_random_value() {
        0             => println!("Zero"),
        n @ 1..=49    => println!("Less than half with value: {}",n),
        n @ 50..=99   => println!("Better than half with value: {}",n),
        100           => println!("100"),
        _             => println!("Impossible value")
    }
}
```

| **Output (possible)** |
| --- |
| Better than half with value: 57 |

Notice that the result of get_a_random_value() function is not stored in function main. As such if we want to use the actual value in one of the match construct arms, we can't.

# Match

You can also bind with **enums** variants values. Let's change the previous example to use an Option instead of an u8 value.

*Rust*

```
fn get_a_random_value() -> Option<u8> {
    let x = rand::random::<u8>();
    if x<101u8 { Some(x) } else { None }
}
fn main() {
    match get_a_random_value() {
        Some(0)           => println!("Zero"),
        Some(n @ 1..=49)  => println!("Less than half with value: {}",n),
        Some(n @ 50..=99) => println!("Better than half with value: {}",n),
        Some(100)         => println!("100"),
        Some(_)           => println!("Other cases"),
        None              => println!("Higher than 100 value")
    }
}
```

**Output (possible)**

Better than half with value: 57

# Match

You can also bind with **enums** variants values. Let's change the previous example to use an Option instead of an u8 value.

*Rust*

```
fn get_a_random_value() -> Option<u8> {
    let x = rand::random::<u8>();
    if x<101u8 { Some(x) } else { None }
}
fn main() {
    match get_a_random_value() {
        Some(0)           => println!("Zero"),
        Some(n @ 1..=49)  => println!("Less than half with
        Some(n @ 50..=99) => println!("Better than half with v
        Some(100)         => println!("100"),
        Some(_)           => println!("Other cases"),
        None              => println!("Higher than 100 value")
    }
}
```

**Output (possible)**

Better than half with value: 57

Notice that in reality, Some(_) case is not needed as all possible values are already covered. However, as all possible u8 values for the Some case must be covered we have to add it otherwise we will not be able to compile the code.

# Match

Rust also support guards for a rule/match-arm. This allows using a more complex checks than the one than the ones allowed by the pattern rule/matching arm.

*Rust*

```
fn get_a_random_value() -> u8 {
    rand::random::<u8>() % 101u8
}
fn main() {
    match get_a_random_value() {
        v if v == 0   => println!("Zero"),
        v if v < 49   => println!("Less than half with value: {}",v),
        v if v < 99   => println!("Better than half with value: {}",v),
        v if v == 100 => println!("100"),
        _             => println!("Impossible value")
    }
}
```

**Output (possible)**

```
Better than half with value: 95
```

**OBS**: *Keep in mind that guards might have a performance impact (use them carefully !)*

# Match

Let's consider an even more complex example where we use the match guard to filter out prime numbers.

*Rust*

```rust
fn get_a_random_value() -> u8 {
    rand::random::<u8>() % 20u8
}
fn is_prime(value: u8) -> bool {
    if value <  2 { return false; }
    if value == 2 { return true; }
    for i in 2..=(value/2) {
        if (value % i) == 0 { return false; }
    }
    return true;
}
fn main() {
    match get_a_random_value() {
        v if v == 0       => println!("Zero"),
        v if is_prime(v) => println!("A prime number: {}",v),
        v                 => println!("Other values: {}",v)
    }
}
```

**Output (possible)**

```
A prime number: 5
```

# Match (C++ vs Rust)

| Option/Feature | Rust | C++ |
|---|---|---|
| Match numerical value (e.g. integers) | YES | YES |
| Match strings constants (literals) → "…" | YES | - |
| Match enums (classical) | YES | YES |
| Match enums (variant style) | YES | - |
| Match structs | YES | - |
| Match tuples | YES | - |
| Match arrays | YES | - |
| Match multiple values | YES | YES |
| Match intervals | YES | - |
| Guards | YES | - |
| Variable binding | YES | - |
| Continue to next rule/match arm | - | YES |

# Match

Keep in mind that a match construct is desired to be fast (ideally with O(1) access time), but these optimization will not always be possible. Some of the limitations of C++ are because they can not obtain a better performance with a switch for some cases (other than a linear O(n) one).

Examples where using a **match** should provide **best performance**:

- Constant numbers (ideally consecutive numbers: 0,1,2, ….)
- Enum values (but not variants)

Examples where using a match will probably translate into a chained **if…else** constructs

- Strings → it is more efficient to use an automata
- Multiple numeric intervals (especially if they have gaps)
- Guards

Q & A