Rust's Generic Programming

Florian Rascoussier

27.06.2023 Passau

Genericity and traits in Rust

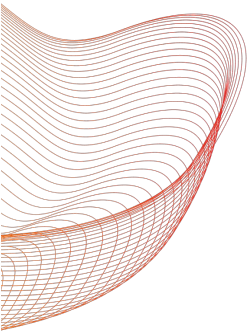
What is Generic Programming ?

Introduction to Generic Programming Paradigm

Definition of Generic Programming

History and evolution of the paradigm

List of programming languages with this feature



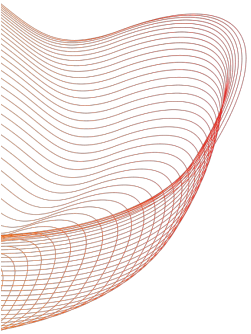
- style of computer programming
- seeks to abstract algorithms for increased reusability
- algorithms are written in terms of types to-be-specified-later
- Instantiation when needed for specific types provided as parameters.

Introduction to Generic Programming Paradigm

Definition of Generic Programming

History and evolution of the paradigm

List of programming languages with this feature



- since the late 1970s and early 1980s.
- ML (1970s) introduced polymorphic (generic) types
- Ada (1980s) added generic modules called "generic packages"
- Popularized by C++ STL developed by Alexander Stepanov in the early to mid-1990s
  - STL: provide generic algorithms and data structures, making extensive use of C++ templates

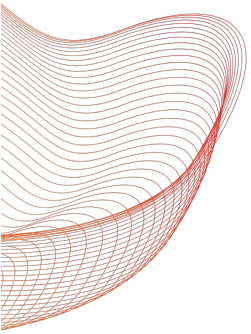
Introduction to Generic Programming Paradigm

Definition of Generic Programming

History and evolution of the paradigm

List of programming languages with this feature

many modern programming languages have incorporated some form of generic programming:



- C++: template & STL
- Java: generics & Interfaces
- C#: generics
- Swift: generics
- Rust: generics & Traits
- Kotlin: generics
- Haskell: generics & Traits
- Scala: generics & Traits

Generic Programming in Rust

Rust

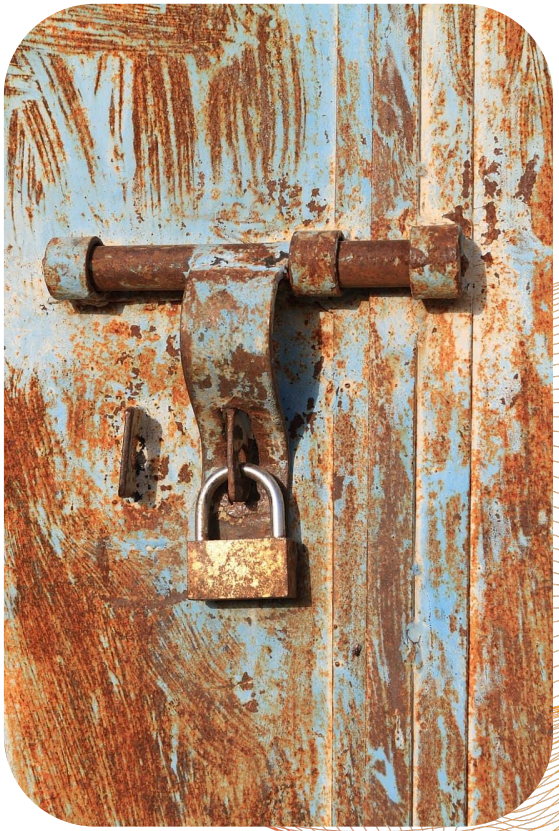
> Multi-paradigm

> 2015, 8 years ago

01 programming language that prioritizes **safety**, **speed**, and **concurrency**, making it well-suited for system-level and performance-critical applications

02 employs a **unique ownership model** with zero-cost abstractions to manage resources, offering a powerful **control over memory** management while avoiding common bugs like null pointer dereferencing and data races

03 expressive **type system** and supports **generic programming and trait-based composition**, facilitating code reusability and maintainability



Genericity and traits in programming languages

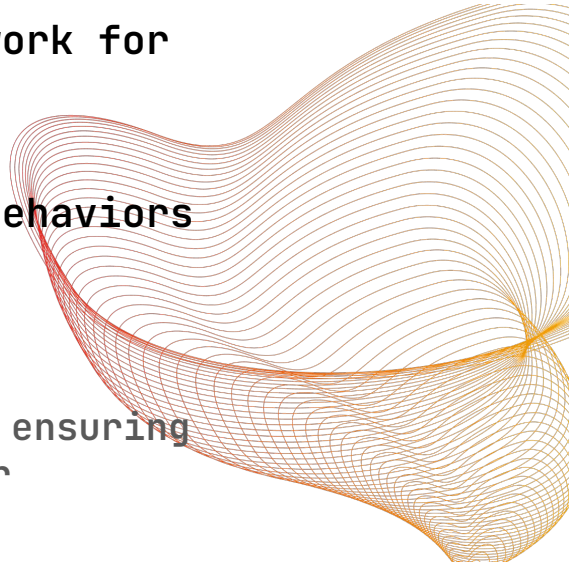
Genericity:

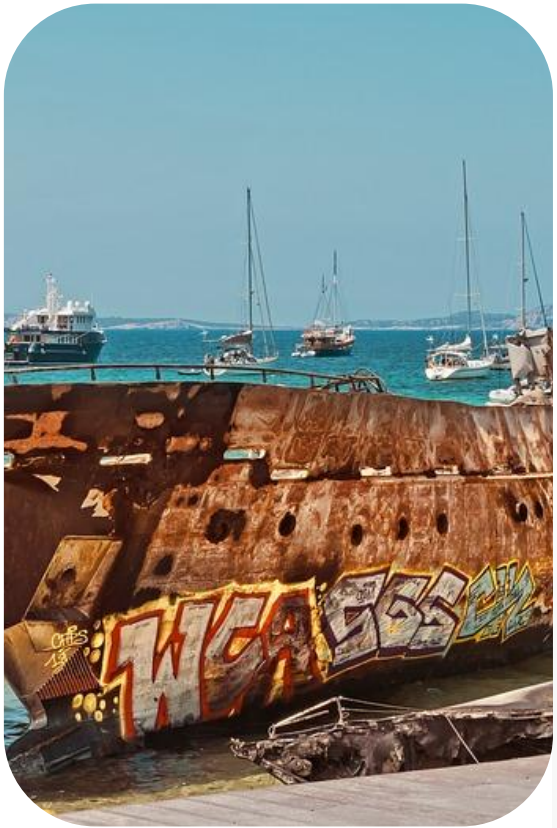
- Allows to write reusable code.
- Build data structures and functions that can work for multiple types of data

Traits:

- guarantee that a particular type has certain behaviors
- Specify and regroup common behaviors

> Using traits as bounds on generics allows to write safe-reusable code that works on multiple types while ensuring those types conform to a certain interface of behavior





Simple example of Generic Programming in Rust

> define a data type or function that doesn't specify the exact type it operates on

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

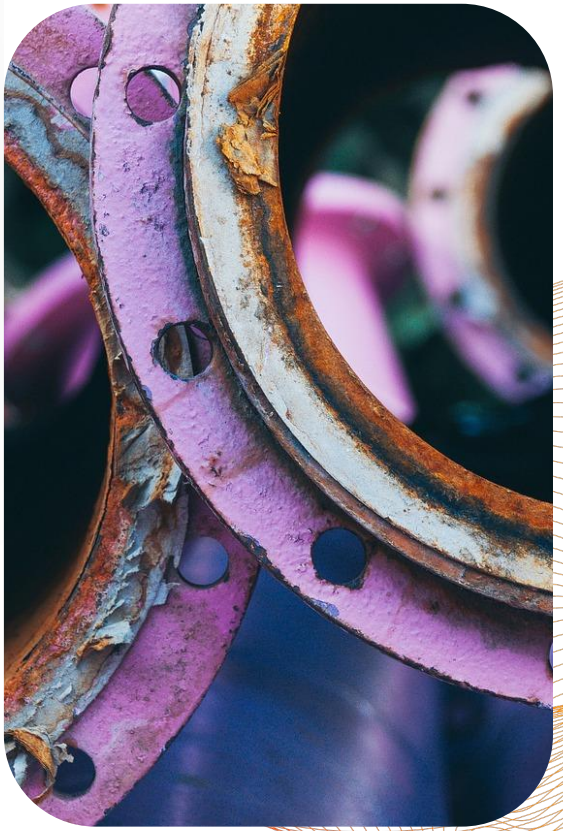
> define a data type or function that doesn't specify the exact type it operates on

More complex example of Generic Programming

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
struct Position {
    x: f32,
    y: f32,
}
```

```
struct Boxer {
    weight: f32,
    name: String,
}
```



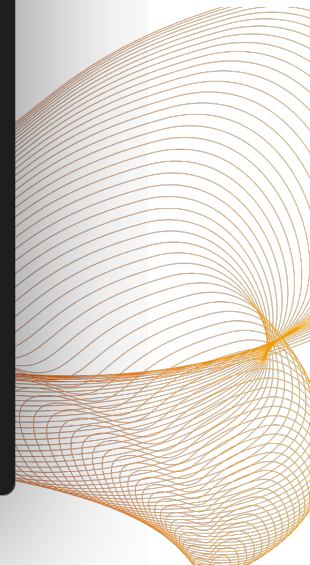
A simple example: Comparing things

```rust
use std::cmp::Ordering;

impl PartialOrd for Position {
    fn partial_cmp(&self, other: &Position) -> Option<Ordering> {
        if self.x == other.x && self.y == other.y {
            Some(Ordering::Equal)
        } else if self.x <= other.x && self.y <= other.y {
            Some(Ordering::Less)
        } else {
            Some(Ordering::Greater)
        }
    }
}
```

A simple example: Comparing things

```rust
use std::cmp::Ordering;

impl PartialEq for Position {
    fn eq(&self, other: &Position) -> bool {
        self.x == other.x && self.y == other.y
    }
}
```

```rust
impl PartialOrd for Boxer {
    fn partial_cmp(&self, other: &Boxer) -> Option<Ordering> {
        if self.weight == other.weight {
            Some(Ordering::Equal)
        } else if self.weight <= other.weight {
            Some(Ordering::Less)
        } else {
            Some(Ordering::Greater)
        }
    }
}

impl PartialEq for Boxer {
    fn eq(&self, other: &Boxer) -> bool {
        self.weight == other.weight
    }
}
```



A simple example: Comparing things

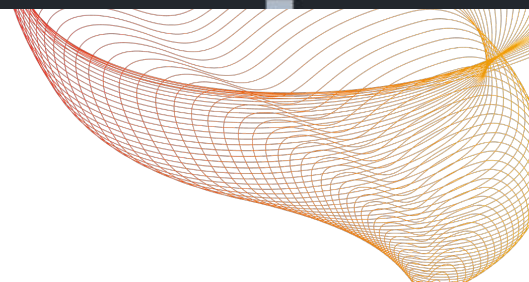
```
fn main() {

      // test largest function on Position
    let position1 = Position { x: 1.0, y: 2.0 };
    let position2 = Position { x: 2.0, y: 10.0 };
    let position3 = Position { x: 100.0, y: 200.0 };

    let positions = vec![position1, position2, position3];

    let largest_position = &largest(&positions);
    println!("largest position {{ x:{}, y:{} }}", largest_position.x,
largest_position.y);

  ...
```

A simple example: Comparing things

```
    // test largest function on boxer
    let boxer1 = Boxer { weight: 110.0, name: String::from("Mike Tyson")
};
    let boxer2 = Boxer { weight: 80.0, name: String::from("Muhammad Ali")
};
    let boxer3 = Boxer { weight: 100.0, name: String::from("George
Foreman") };
    let boxers = vec![boxer1, boxer2, boxer3];

    let largest_boxer = &largest(&boxers);
    println!("largest boxer {{ weight:{}, name:{} }}",
largest_boxer.weight, largest_boxer.name);

}
```

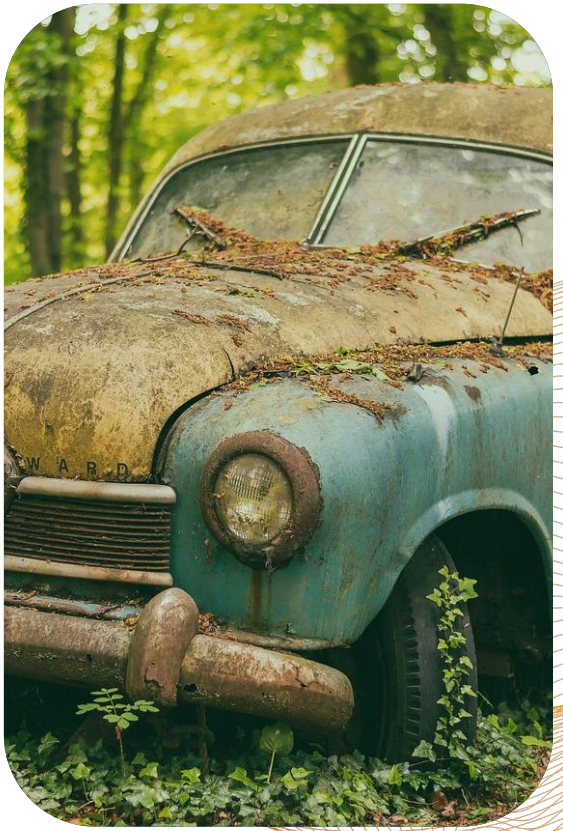
A simple example: Comparing things

```
〈onyr ★ kenzael〉〈rust_programming_paradigm_presentation〉〉 cargo run
   Compiling rust_programming_paradigm_presentation v0.1.0 (/home/onyr
esentation)
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/rust_programming_paradigm_presentation`
Hello, world!
largest position { x:100, y:200 }
largest boxer { weight:110, name:Mike Tyson }
```



Pros of Generic Programming in Rust

1. **Code Reusability**
2. **Type Safety**: generics checked at comp. time
3. **Performance: No runtime overhead.** At compile time, Rust uses a process called **monomorphization** where it generates non-generic versions of your generic code for each specific instance they are used, which makes the code as fast as writing it for a specific type.
4. **Better Abstraction**: better abstraction, shared behavior across different types. Easier encapsulation and reasoning



Cons of Generic Programming in Rust

1. **Increased Compile Times**: can increase compile times
2. **Complexity**: can make the code more complex and harder to read, especially for those new to these concepts.
3. **Error Messages**: error messages can sometimes become cryptic and hard to decipher
4. **Limitations of the Trait System**: Certain functionalities are difficult to implement using Rust's trait system.

Examples in other languages

C++ | Java | Haskell

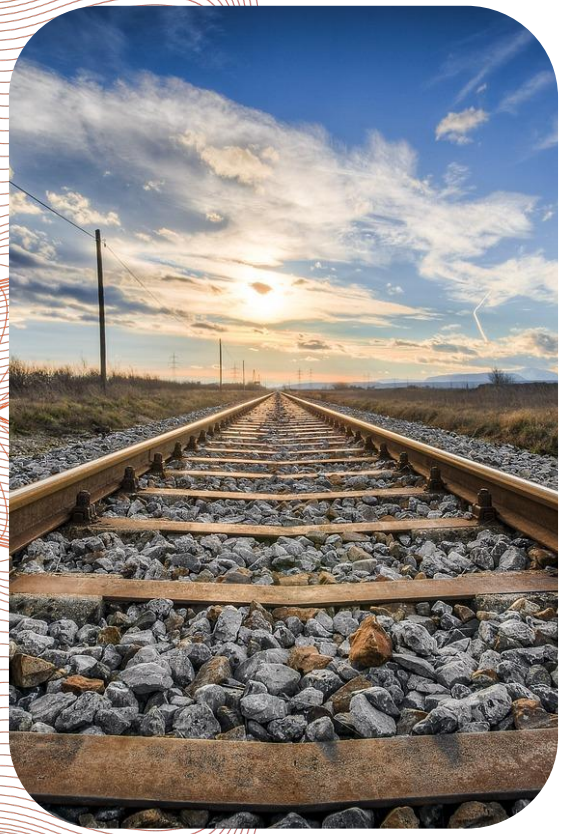
```cpp
#include <iostream>
#include <string>

// Define the interface
class Summarizable {
public:
    virtual std::string summarize() = 0; // pure virtual function
};

// Function that accepts a Summarizable object
template <typename T>
void notify(T& item) {
    static_assert(std::is_base_of<Summarizable, T>::value, "T must inherit from Summarizable");
    std::cout << "Breaking news! " << item.summarize() << std::endl;
}
```

C++ | Java | Haskell

```java
public interface Summarizable {
    String summarize();
}

public static <T extends Summarizable> void notify(T item) {
    System.out.println("Breaking news! " + item.summarize());
}
```

C++ Java Haskell

```haskell
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}

class Summarizable a where
    summarize :: a -> String

notify :: (Summarizable a) => a -> IO ()
notify item = putStrLn ("Breaking news! " ++ summarize item)
```



Conclusion

01 Further reading and resources

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer, “Safe systems programming in Rust” Communications of the ACM, Volume 64, Number 4 (2021), Pages 144-152

02 Summary of key takeaways

- **Genericity**: use different types for same thing
- **Traits**: define an expected behavior

> Increase in safe-reusable code
> No runtime overhead