Rust programming

Course – 6

Gavrilut Dragos

rev 9

Agenda for today

1. OOP concepts in Rust
2. Traits
3. Super traits / inheritance
4. Special traits
5. Operators

OOP

OOP

Rust structures have both a role of a structure and C++ class. However, there are several differences between how a class in C++ and its equivalent in Rust are designed.

Maybe one of the most important one, is that methods for every object are implemented separately (and not as part of that object definition). This techniques allows rust to define traits (characteristics) that can be define for every object (including the one that are already part of the standard library and basic types).

To add a method to a class, use the **impl** keyword, follow by the name of the class.

Rust

```rust
struct MyClass {
    // data members
}
impl MyClass {
    // methods of the class
}
```

Methods

Methods are defined with the impl construct with the following format:

- *[visibility]* **fn** method_name (*[params]*) -> *<return_type>* {…}
- *[visibility]* **fn** method_name (**self**, *[params]*) -> *<return_type>* {…}
- *[visibility]* **fn** method_name (**&self**, *[params]*) -> *<return_type>* {…}
- *[visibility]* **fn** method_name (**&mut self**, *[params]*) -> *<return_type>* {…}

Where:

- *[params]* → is a list of parameters (similar to the one that can be added to a regular Rust function)
- *[visibility]* → a set of keywords that explain the visibility of that method
- *<return_type>* → the return type of that method
- **self, &self, &mut self** → used if the method is applying to the object. If not prezent, the method is considered static.

Methods

Let's see a very simple example:

Rust

```
struct MyClass {
    value: i32
}
impl MyClass {
    fn inc(&mut self) { self.value += 1; }
    fn get(&self) -> i32 { return self.value; }
}

fn main() {
    let mut obj = MyClass{value:0};
    obj.inc();
    println!("{}",obj.get());
}
```

Output

```
1
```

Methods

Let’s see a very simple example:

Rust

```
struct MyClass {
    value: i32
}
impl MyClass {
    fn inc(&mut self) { self.value += 1; }
    fn get(&self) -> i32 { return self.value; }
}

f

    obj.inc();
    println!("{}",obj.get());
}
```

“self” is somehow similar to “this” pointer from C++. For this method a mutable reference to the object is required in order to be able to change its data members.

Output

```
1
```

Methods

Let's see a very simple example:

Rust

```
struct MyClass {
    value: i32
}
impl MyClass {
    fn inc(&mut self) { self.value += 1; }
    fn get(&self) -> i32 { return self.value; }
}

f
    println!("{}",obj.get());
}
```

In this case we only need an immutable reference towards the object as we don't need to modify its content.

Output

```
1
```

Methods

Let's see a very simple example:

Rust

```rust
struct MyClass {
    value: i32
}
impl MyClass {
    fn inc(&mut self) { self.value += 1; }
    fn get(&self) -> i32 { return self.value; }
}
fn main() {
    let mut obj = MyClass{value:0};
    obj.value = 5;
    obj.inc();
    println!("{}",obj.get());
}
```

C++

```cpp
class MyClass {
    public:
        int value;
        void inc() { value+=1; }
        int get() const { return value; }
};
void main() {
    MyClass obj;
    obj.value = 5;
    obj.inc();
    printf("%d\n",obj.get());
}
```

Output

```
6
```

OBJ: *Notice that methods in Rust that receive a &self are translated as const methods in C++ (see method get)*

Static Methods

If the &self / &mut self or self are omitted when defining an object method, that method is considered to be static. In this example, method print_name is static and can only be access via the class/struct name specifier.

Rust

```rust
struct MyClass {
    value: i32
}
impl MyClass {
    fn print_name() {
        println!("MyClass");
    }
}
fn main() {
    MyClass::print_name();
}
```

Output

```
MyClass
```

C++

```cpp
class MyClass {
    public:
        int value;
        static void print_name() {
            printf("MyClass");
        }
};
void main() {
    MyClass::print_name();
}
```

Static Methods

If you want to call a static method from a regular method you can use either "Self" (with capital "S") as a type, or the name of the type you are implementing a method for.

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn print_name() {
        print!("MyClass");
    }
    fn print_me(&self) {
        Self::print_name();
        println!(" -> value: {}",self.value);
    }
}
fn main() {
    let x = MyClass{value:10};
    x.print_me();
}
```

Output

MyClass -> value: 10

Static Methods

If you want to call a static method from a regular method you can use either "Self" (with capital "S") as a type, or the name of the type you are implementing a method for.

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn print_name() {
        print!("MyClass");
    }
    fn print_me(&self) {
        Self::print_name();
        println!(" -> value: {}",self.value);
    }
}
fn
}
```

Output

MyClass -> value: 10

Alternatively, you can use "`MyClass::print_name()`" to obtain the same result.

Static data members

There are also no static data members in Rust. However, we can use global variable to achieve a similar result as a static data member in C++. When we are going to talk about visibility, we will show how this global variables can be hidden.

Rust

```rust
struct MyClass {
    value: i32,
}
static mut my_class_x: i32 = 10;
impl MyClass {
    fn inc() { unsafe { my_class_x += 1; } }
    fn get()->i32 {
        unsafe { return my_class_x; }
    }
}
fn main() {
    MyClass::inc();
    println!("{}",MyClass::get());
}
```

Output

```
11
```

C++

```cpp
class MyClass {
    public:
        int value;
        static int x;
        static void inc() { x++; }
        static int get() { return x; }
};
int MyClass::x = 10;
void main() {
    MyClass::inc();
    printf("%d\n",MyClass::get());
}
```

Static data members

There are also no static data members in Rust. However, we can use global variable to achieve a similar result as a static data member in C++. When we are going to talk about visibility, we will show how this global variables can be hidden.

Rust

```rust
struct MyClass {
    value: i32,
}
static mut my_class_x: i32 = 10;
impl MyClass {
    fn inc() { unsafe { my_class_x += 1; } }
    fn get()->i32 {
        unsafe { return my_class_x; }
    }
}
                                        ());
```

Notice the usage of a special keyword -> unsafe. Global variables can be modified by multiple threads and as such their usage may lead to undefined behavior.

C++

```cpp
class MyClass {
    public:
        int value;
        static int x;
        static void inc() { x++; }
        static int get() { return x; }
};
int MyClass::x = 10;
void main() {
    MyClass::inc();
    printf("%d\n",MyClass::get());
}
```

Static data members

As a general rule, it is not recommended to create a global (mutable) variable to be used as a static field for an object. However, since some designed patters (such as **Singleton**) might require such an approach this is allowed but must be done in such a way that access to that variable is limited (so that we reduce the chance of an undefined behavior).

If such a construct is used without the unsafe keyword, the code will not compile.

Rust

```rust
struct MyClass {
    value: i32,
}
static mut my_class_x: i32 = 10;
impl MyClass {
    fn inc() {  my_class_x += 1;  }
    fn get()->i32 { return my_class_x;  }
```

Error

```
error[E0133]: use of mutable static is unsafe and requires
unsafe function or block
 --> src\main.rs:6:17
  |
6 |     fn inc() {  my_class_x += 1;  }
  |                 ^^^^^^^^^^^^^^^ use of mutable static
  |
  = note: mutable statics can be mutated by multiple threads:
          aliasing violations or data races will cause undefined
          behavior
```

Calling methods

Another interesting thing is that (semantically) Rust has "self" (in different forms) as the first parameter for methods that are associated/implemented for a struct.

This implies that a method is a little bit different than what we know from C++. In C++ a method can only be called by the actual object, in Rust a method can be called in two different ways:

- **object**.*method* ($Param_1$, $Param_1$,.. $Param_n$), where object is of type ObjectType

or

- **ObjectType**::*method* (*[reference]***object**, $Param_1$, $Param_1$,.. $Param_n$)

Where *[reference]* refers to the fact that the first parameter should reflect its definition (self , &self or &mut self)

Calling methods

Let’s see an example:

Rust

```
struct A { value: u32 }
impl A {
    fn print_a(&self) {
        println!("value = {}",self.value);
    }
}

fn main() {
    let a1 = A{value:10};
    let a2 = A{value:20};
    a1.print_a();
    A::print_a(&a2);
}
```

Output

```
value = 10
value = 20
```

Notice that we have called **print_a** method in two different ways !

Calling methods

Also, there is no difference between a regular function that is designed to take the first parameter a reference or an object of type "A", or a similar method implemented for type "A". In this example, we showcase this behavior. Method **call** receives a pointer to a function that has a first parameter of type &mut A and the second parameter of type u32. Both "g" and "A::f" qualify for this type of functions.

Rust

```rust
struct A {}
impl A {
    fn f(&mut self, x: u32) { println!("{}", x); }
}
fn g(_: &mut A, x: u32) { println!("{}", x + 10); }
fn call(fun: fn(&mut A, u32)) {
    let mut x = A {};
    fun(&mut x, 5);
}
fn main() {
    call(A::f);
    call(g);
}
```

Output
5 15

Constructors

Rust does not have a constructor-like method similar to what C++ has. This is because any struct has a clear initialization way where each field **MUST BE INITIALIZED**. However, constructors can be simulated via static methods:

Rust

```rust
struct MyClass {
    value: i32
}
impl MyClass {
    fn create(val: i32)-> MyClass {
        return MyClass { value: val };
    }
}
fn main() {
    let m = MyClass::create(10);
    print!("{}",m.value);
}
```

C++

```cpp
class MyClass {
    public:
        int value;
        MyClass(int v): value(v) {}
};
void main() {
    MyClass m(10);
    printf("%d\n",m.value);
}
```

Output

```
10
```

Constructors

Rust does not have a constructor-like method similar to what C++ has. This is because any struct has a clear initialization way where each field **MUST BE INITIALIZED**. However, constructors can be simulated via static methods:

Rust

```rust
struct MyClass {
    value: i32
}
impl MyClass {
    fn create(val: i32)-> MyClass {
        return MyClass { value: val };
    }
}
fn main() {
```

This **static** method acts as a constructor. It creates a new MyClass object and returns it (this transfers the ownership towards the variable "m").

C++

```cpp
class MyClass {
    public:
        int value;
        MyClass(int v): value(v) {}
};
void main() {
    MyClass m(10);
    printf("%d\n",m.value);
}
```

Output

```
10
```

Constructors

Rust does not have a constructor-like method similar to what C++ has. This is because any struct has a clear initialization way where each field **MUST BE INITIALIZED**. However, constructors can be simulated via static methods:

Rust

```rust
struct MyClass {
    value: i32
}
impl MyClass {
    fn create(val: i32)-> MyClass {
        return MyClass { value: val };
    }
}
fn main() {
    let m = MyClass::create(10);
    print!("{}",m.value);
}
```

```asm
mov     ecx,10
call    MyClass::create
mov     dword ptr [m],eax
```

```asm
push  rax
mov   dword ptr [rsp],ecx
mov   eax,dword ptr [rsp]
pop   rcx
ret
```

C++

```cpp
class MyClass {
    public:
        int value;
        MyClass(int
...
m.
}
```

Output

10

Constructors

Rust does not have a constructor-like method similar to what C++ has. This is because any struct has a clear initialization way where each field **MUST BE INITIALIZED**. However, constructors can be simulated via static methods:

Rust

```rust
struct MyClass {
    value: i32
}
impl MyClass {
    fn create(val: i32)-> MyClass {
        return MyClass { value: val };
    }
}
fn main() {
    let m = MyClass::create(10);
    print!("{}",m.value);
}
```

C++

```cpp
class MyClass {
    public:
        int value;
        MyClass(int v): value(v) {}
}
```

This means that in reality, what Rust does is to copy the value that we get from parameter “val” to “m” variable from the main function.

Output

```
10
```

Constructors

Rust also have a special type call Self that refers to the current type (not object). It is often useful when returning an object of that type.

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn create(val: i32) -> Self {
        return MyClass { value: val };
    }
}
fn main() {
    let m = MyClass::create(10);
    print!("{}", m.value);
}
```

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn create(val: i32) -> MyClass {
        return MyClass { value: val };
    }
}
fn main() {
    let m = MyClass::create(10);
    print!("{}", m.value);
}
```

Output

```
10
```

Constructors

Let's try a more complex case (where the class has multiple members).

Rust

```
struct MyClass {
    value: i32,
    data: [u8;30]
}
impl MyClass {
    fn new(val: i32, d: u8)-> MyClass {
        return MyClass { value: val, data: [d;30] };
    }
}

fn main() {
    let m = MyClass::new(1,2);
    print!("{}",m.value);
}
```

Constructors

Let's try a more complex case (where the class has multiple members).

Rust

```rust
struct MyClass {
    value: i32,
    data: [u8;30]
}
impl MyClass {
    fn new(val: i32, d: u8)-> MyClass {
        return MyClass { value: val, data: [d;30] };
    }
}

fn main() {
    let m = MyClass::new(1,2);
    print!("{}",m.value);
}
```

```
lea         rcx,[m]
mov         edx,1
mov         r8d,2
call        MyClass::new
```

Constructors

Let's try a more complex case (where the class has multiple members).

Rust

```rust
struct MyClass {
    value: i32,
    data: [u8;30]
}
impl MyClass {
    fn new(val: i32, d: u8)-> MyClass {
        return MyClass { value: val, data: [d;30] };
    }
}

fn main() {
    let m = MyClass::new(1,2);
    print!("{}",m.value);
}
```

```asm
mov     byte ptr [rsp+2Bh],r8b
mov     eax,edx // eax = 1
mov     dl,byte ptr [rsp+2Bh] // edx = 2
mov     dword ptr [rsp+2Ch],eax
mov     qword ptr [rsp+30h],rcx
mov     qword ptr [rsp+38h],rcx
mov     dword ptr [rsp+60h],eax
mov     byte ptr [rsp+67h],dl
lea     rcx,[rsp+42h]
mov     r8d,30
call    memset
mov     edx,dword ptr [rsp+2Ch]
mov     rcx,qword ptr [rsp+30h]
mov     rax,qword ptr [rsp+38h]
mov     dword ptr [rcx],edx // m.value = 1
mov     rdx,qword ptr [rsp+42h]
mov     qword ptr [rcx+4],rdx
mov     rdx,qword ptr [rsp+4Ah]
mov     qword ptr [rcx+0Ch],rdx
mov     rdx,qword ptr [rsp+52h]
mov     qword ptr [rcx+14h],rdx
mov     edx,dword ptr [rsp+5Ah]
mov     dword ptr [rcx+1Ch],edx
mov     dx,word ptr [rsp+5Eh]
mov     word ptr [rcx+20h],dx
```

memcpy

Constructors

Let’s try a more complex case (where the class has multiple members).

Rust

```rust
struct MyClass {
    value: i32,
    data: [u8;30]
}
impl MyClass {
    fn new(val: i32, d: u8)-> MyClass {
        return MyClass { value: val,
                         data: [d;30]
                       };
    }
}

fn main() {
    let m = MyClass::new(1,2);
    print!("{}",m.value);
}
```

C++ (approximation)

```cpp
class MyClass
{
    public:
        int value;
        uint8_t data[30];
        static void fn_new(MyClass * output, int val, uint8_t d)
        {
            MyClass local_obj;
            local_obj.value = val;
            memset(local_obj.data,d,30);
            memcpy(output,&local_obj,sizeof(MyClass));
        }
};
void main() {
    MyClass m;
    MyClass::fn_new(&m, 1, 2);
    printf("%d\n",m.value);
}
```

Constructors

In reality, there is no real difference on how Rust constructs an object (as opposite on how C++ does it). Both of them receive the address where the actual object is located and construct it there.

Usually, Rust uses names like:

- **new(…)**
- **from(…)**
- **with_...(…)**

to describe a constructor. However, any name can be used.

OBS: *from is part of a trait and while it is used to construct an object it is usually associated with that trait.*

Constructors

Keep in mind that defining a function similar to a constructor does not imply than an object can not be created in different ways. In the next example, we create an object of type MyClass using two different methods (::create(...)) and structure initialization).

Rust (via create method)

```
struct MyClass {
    value: i32
}
impl MyClass {
    fn create(val: i32)-> MyClass {
        return MyClass { value: val };
    }
}
fn main() {
    let m = MyClass::create(10);
    print!("{}",m.value);
}
```

Rust (via structure initialization).

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn create(val: i32) -> MyClass {
        return MyClass { value: val };
    }
}
fn main() {
    let m = MyClass{value:10};
    print!("{}", m.value);
}
```

Output

```
10
```

Constructors

One advantage of construction an object like this, is that we can return an error when trying to construct an object, while using the constructor concept in C++ makes this task more complicated.

Let’s assume that we have an object (of type ***Student***). For each student we have a name and a grade → but the grade should be between 1 and 10.

Using a constructor (like in C++) you can not return an error (so in theory every object is valid). In Rust, we can return an Option<> or a Result<> and only if the result is valid (Some for Option or Ok for Result) we obtain an instance of a specific type.

Constructors

Let's see an example:

Rust

```
#[derive(Debug)]
struct Student {
    grade: i32,
    name: String
}
impl Student {
    fn new(stud_name: &str, stud_grade: i32) -> Option<Student> {
        if (stud_name.len()>0) && (stud_grade>=1) && (stud_grade<=10) {
            return Some(Student{grade: stud_grade, name: String::from(stud_name)});
        }
        return None;
    }
}
fn main() {
    let s1 = Student::new("Andrei",-5);
    let s2 = Student::new("Dragos",10);
    println!("s1={:?}",s1);
    println!("s2={:?}",s2);
}
```

Output

```
s1=None
s2=Some(Student { grade: 10, name: "Dragos" })
```

Constructors

Keep in mind that static functions are possible in C++ as well. This means that the same technique can be used there (create an object via a static function). The only difference is if we need to allocate a class in the heap or if we need to create an array. Since C++ builds a class directly in the allocated memory, there is a need of a constructor method that can be called automatically when an object is created.

Rust works by creating a temporary object first and then assigned it to the actual object (transfer the ownership). Because of this, any kind of static function will work as we will need to provide that temporary object first , and then the assignment is performed by Rust.

Constructors

Obviously, there is no implicit default constructor in Rust. However, it is a common practice to name it **new**, while other constructors that imply creating from a specific type prefer the prefix **from** (as a derivation from the trait From).

Rust

```rust
#[derive(Debug)]
struct MyClass {
    value: i32,
}
impl MyClass {
    fn new() -> MyClass { MyClass{value:0} }
    fn from_i32(val: i32) -> MyClass { MyClass{value:val} }
}
fn main() {
    let m1 = MyClass::new();
    let m2 = MyClass::from_i32(10);
    println!("m1={:?}",m1);
    println!("m2={:?}",m2);
}
```

Output

```
m1=MyClass { value: 0 }
m2=MyClass { value: 10 }
```

Functional update syntax

The usage of .. operator is also called **functional update syntax**. It implies that you can use this to call another initialization method (that will be called first) followed by you own changes. Let's see some example:

Rust

```
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: &'static str,
}
fn main() {
    let obj = Test {
        x: 1,
        ..Test { x: 5, y: 3, name: "abc" }
    };
    println!("obj={:?}", obj);
}
```

Output

```
obj=Test { x: 1, y: 3, name: "abc" }
```

In this case, first the `..Test { x: 5, y: 3, name: "abc" }` is called that instantiate **obj** with {x=5,y=3,name="abc"}; Then, x is being overwritten with value 1.

Functional update syntax

The usage of .. operator is also called **functional update syntax**. It implies that you can use this to call another initialization method (that will be called first) followed by you own changes. Let's see some example:

Rust

```
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: &'static str,
}
impl Test {
    fn new() -> Test { Test { x: 0, y: 0, name: "" } }
}
fn main() {
    let obj = Test {
        x: 1,
        name: "xyz",
        ..Test::new()
    };
    println!("obj={:?}", obj);
}
```

Output

```
obj=Test { x: 1, y: 0, name: "xyz" }
```

In this case, first the `..Test::new()` is called that instantiate **obj** with {x=0,y=0,name=""}; Then, x is being overwritten with value 1, and name with value "xyz"

Functional update syntax

The usage of .. operator is also called **functional update syntax**. It implies that you can use this to call another initialization method (that will be called first) followed by you own changes. Let's see some example:

Rust

```rust
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: &'static str,
}
impl Test {
    fn new(val: i32) -> Test { Test { x: val, y: val, name: "" } }
}
fn main() {
    let obj = Test {
        name: "xyz",
        ..Test::new(5)
    };
    println!("obj={:?}", obj);
}
```

Output

```
obj=Test { x: 5, y: 5, name: "xyz" }
```

In this case, first the `..Test::new(5)` is called that instantiate **obj** with {x=5,y=5,name=""}; Then, name is overwritten with value "xyz"

Functional update syntax

The `..` operator has to be the last from the declaration.

Rust

```rust
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: &'static str,
}
impl Test {
    fn new(val: i32) -> Test { Test { x: val, y: val, name: "" } }
}
fn main() {
    let obj = Test {
        name: "xyz",
        ..Test::new(5),
        x: 1
    };
    println!("obj={:?}", obj);
}
```

Error

```
error: cannot use a comma after the base struct
  --> src\main.rs:13:9
   |
13 |         ..Test::new(5),
   |         ^^^^^^^^^^^^^^- help: remove this comma
   |
   = note: the base struct must always be the last field
```

Functional update syntax

When using functional update syntax, you can also use another object (of the same type) as your base:

Rust

```
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: &'static str,
}
impl Test {
    fn new(val: i32) -> Test { Test { x: val, y: val, name: "" } }
}
fn main() {
    let base = Test::new(5);
    let obj = Test {
        name: "xyz",
        ..base
    };
    println!("obj={:?}, base={:?}", obj,base);
}
```

Output

```
obj=Test { x: 5, y: 5, name: "xyz" },
base=Test { x: 5, y: 5, name: "" }
```

Functional update syntax

However, there are a couple of pitfalls that we need to take into consideration:

Rust

```rust
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: String,
}
impl Test {
    fn new(val: i32) -> Test { Test { x: val, y: val, name: String::from("123") } }
}
fn main() {
    let mut base = Test::new(5);
    base.x = 123;
    let obj = Test {
        name: String::from("abc"),
        ..base
    };
    println!("obj={:?}, base={:?}", obj,base);
}
```

Output

```
obj=Test { x: 123, y: 5, name: "abc" },
base=Test { x: 123, y: 5, name: "123" }
```

Notice tha this snipped works as expected !

Functional update syntax

However, there are a couple of pitfalls that we need to take into consideration:

Rust

```
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: String,
}
impl Test {
    fn new(val: i32) -> Test { Test { 
}
fn main() {
    let mut base = Test::new(5);
    base.x = 123;
    let obj = Test {
        x: 10,
        ..base
    };
    println!("obj={:?}, base={:?}", obj,base);
}
```

Error

```
error[E0382]: borrow of partially moved value: `base`
  --> src\main.rs:17:41
   |
13 |         let obj = Test {
   |  ________________-
14 | |           x: 10,
15 | |           ..base
16 | |       };
   | |_____- value partially moved here
17 |         println!("obj={:?}, base={:?}", obj,base);
   |                                             ^^^^ value borrowed here after partial move
   |
   = note: partial move occurs because `base.name` has type `String`, which does not
implement the `Copy` trait
```

Functional update syntax

Let’s analyze a little bit better what the next piece of code implies:

```
let obj = Test { x: 10,
                 ..base
               };
```

```
struct Test {
    x: i32,
    y: i32,
    name: String,
}
```

Steps:

1. Initialize obj with all fields that are provided (in our case → “x”)
2. Copy/Move all elements from base that are not needed by the current initialization (in our case, since we already initialized “x”, we will assign “y” and “name”). For “y” everything is ok, but “name” will be moved as it does not contain the Copy trait.
3. As such, trying to print ***base*** after this step is invalid (as it has a partially moved member – name).

Functional update syntax

Now the code works, but notice that we **<u>don’t print base.name</u>** that was moved !!!

Rust

```
#[derive(Debug)]
struct Test {
    x: i32,
    y: i32,
    name: String,
}
impl Test {
    fn new(val: i32) -> Test { Test { x: val, y: val, name: String::from("123") } }
}
fn main() {
    let mut base = Test::new(5);
    base.x = 123;
    let obj = Test {
        x: 10,
        ..base
    };
    println!("obj={:?}, base.x={}, base.y={}", obj,base.x, base.y);
}
```

Output

```
obj=Test { x: 10, y: 5, name: "123" }, base.x=123, base.y=5
```

Method overloading

Rust ***does not support*** method overloading (in the sense that there can not be two methods with the same name as part of the same implementation of one class). We emphasize the word: "***same implementation of one class***" as methods with the same name are allowed with traits (we will discuss about this later) or with generics/templates.

One major advantage here is clarity (if you have multiple functions with the same name, its is not always clear how parameters must be converted to match one of the functions). If you only have one function with a specific name, this issue will **NOT** be encountered anymore.

Method overloading

Let's see an example:

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn add(&mut self, v1: i32) {
        self.value+= v1;
    }
    fn add(&mut self, v1: i32, v2: i32) {
        self.value+= v1+v2;
    }
}
fn main() {
    let m = MyClass{value:0};
    m.add(10);
    m.add(10,20);
    println!("{}",m.value);
}
```

Error

```
error[E0201]: duplicate definitions with name `add`:
  --> src\main.rs:8:5
   |
5  | /     fn add(&mut self, v1: i32) {
6  | |         self.value+= v1;
7  | |     }
   | |_____- previous definition of `add` here
8  | /     fn add(&mut self, v1: i32, v2: i32) {
9  | |         self.value+= v1+v2;
10 | |     }
   | |_____^ duplicate definition
```

Method overloading

The solution in this case is to change the name of those two methods:

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn add_one(&mut self, v1: i32) {
        self.value+= v1;
    }
    fn add_two(&mut self, v1: i32, v2: i32) {
        self.value+= v1+v2;
    }
}
fn main() {
    let mut m = MyClass{value:0};
    m.add_one(10);
    m.add_two(10,20);
    println!("{}",m.value);
}
```

Output
40

Destructors

Rust does not have a destructor method (in the sense of a specific method with the same name as the class) as C++ does. However, there is a special trait called Drop that can be used to define a function with a similar scope.

Furthermore, the lifetime of one object or its transformation can be controlled via methods that receive **self** as an argument (notice that it is **self** and not **&self** or **&mut self**).

This technique transfers the ownership and as a result one can convert that object into another one, or it can drop it.

We will discuss more about destructors when we talk about traits.

Destructors

Let's see an example:

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn destruct_me(self) {
        println!("Destruct object !");
    }
}
fn main() {
    let m = MyClass{value:0};
    m.destruct_me();
    println!("End program");
}
```

Output

```
Destruct object !
End program
```

After this point, "m" lifetime is over and any data that it contains will be dropped.

Destructors

Let’s see an example:

Rust

```
struct MyClass {
    value: i32,
}
impl MyClass {
    fn destruct_me(self) {
        println!("Destruct object !");
    }
}
fn main() {
    let m = MyClass{value:0};
    m.destruct_me();
    println!("m.value = {}",m.value);
}
```

Error

```
error[E0382]: borrow of moved value: `m`
  --> src\main.rs:12:29
   |
10 |     let m = MyClass{value:0};
   |         - move occurs because `m` has type `MyClass`, which does not
   |           implement the `Copy` trait
11 |     m.destruct_me();
   |       ------------- `m` moved due to this method call
12 |     println!("m.value = {}",m.value);
   |                             ^^^^^^^ value borrowed here after move
   |
note: this function takes ownership of the receiver `self`, which moves `m`
  --> src\main.rs:5:20
   |
5  |     fn destruct_me(self) {
   |                    ^^^^
```

Consuming an object

Let's see an example where we convert one object into another (by converting we refer to a transfer of ownership between object fields). This is often known as consuming one object and producing another one !

Rust

```rust
#[derive(Debug)]
struct Student { math: i32, english: i32, name: String }
#[derive(Debug)]
struct StudentAverage { grade: i32, name: String }
impl Student {
    fn convert_to_student_average(self)->StudentAverage {
        StudentAverage{grade: (self.math+self.english)/2, name: self.name}
    }
}
fn main() {
    let s = Student{math:10, english:8, name: String::from("John")};
    println!("Student = {:?}",s);
    let sa = s.convert_to_student_average();
    println!("Average = {:?}",sa);
}
```

Output

```
Student = Student { math: 10, english: 8, name: "John" }
Average = StudentAverage { grade: 9, name: "John" }
```

Consuming an object

Let’s see an example where we convert one object into another (by converting we refer to a transfer of ownership between object fields). This is often known as consuming one object and producing another one !

Rust

```rust
#[derive(Debug)]
struct Student { math: i32, english: i32,
#[derive(Debug)]
struct StudentAverage { grade: i32, name:
impl Student {
    fn convert_to_student_average(self)->S
}
fn main() {
    let s = Student{math:10, english:8, name: String::from("John")};
    println!("Student = {:?}",s);
    let sa = s.convert_to_student_average();
    println!("Average = {:?}",sa);
    println!("Student = {:?}",s);
}
```

Error

```
error[E0382]: borrow of moved value: `s`
  --> src\main.rs:22:31
   |
18 |     let s = Student{math:10, english:8, name: String::from("John")};
   |         - move occurs because `s` has type `Student`, which does not
implement the `Copy` trait
19 |     println!("Student = {:?}",s);
20 |     let sa = s.convert_to_student_average();
   |           --------------------------- `s` moved due to this method call
21 |     println!("Average = {:?}",sa);
22 |     println!("Student = {:?}",s);
   |                               ^ value borrowed here after move
```

Consuming an object

There are several conventions that are usually used in Rust when writing a method that consumes/converts an object:

1. use **into_<type>** if you want to consume current type and obtained a new object by transferring ownership. This type of method receives a **self** as a first argument.

```
struct ClassA { /* data members */ }
struct ClassB { /* data members */ }
impl ClassA { fn into_classB(self, /* other parameters */ ) -> ClassB {...} }
```

2. use **to_<type>** if you want to create a new object and keep the original object (usually this means making a copy/clone of some of the data members of the original object). This type of method receives a **&self** as a first argument.

```
struct ClassA { /* data members */ }
struct ClassB { /* data members */ }
impl ClassA { fn to_classB(&self, /* other parameters */ ) -> ClassB {...} }
```

Consuming an object

There are several conventions that are usually used in Rust when writing a method that consumes/converts an object:

3. use **as_<type>** if you want to convert an immutable reference of type "A" to an immutable reference of type "B". This type of method receives a **&self** as a first argument. Usually this means that type "A" has a data member of type "B".

```
struct ClassA { /* data members */ }
struct ClassB { /* data members */ }
impl ClassA { fn as_classB(&self, /* other parameters */ ) -> &ClassB {...} }
```

Consuming an object

Let's see how these conversion will look like for our ***Student*** structure

Rust

```
struct Student {
    math: i32,
    english: i32,
    name: String,
}
struct StudentAverage {
    grade: i32,
    name: String,
}
impl Student {
    fn into_student_average(self) -> StudentAverage {
        StudentAverage {
            grade: (self.math + self.english) / 2,
            name: self.name,
        }
    }
    fn to_student_average(&self) -> StudentAverage {
        StudentAverage {
            grade: (self.math + self.english) / 2,
            name: self.name.clone(),
        }
    }
}
```

Ownership of "Student::name" is transferred

A copy/clone of "Student::name" is made

Enums

Implementing methods (static and non-static) is not limited to structures, it works similar for ***enums***. To access the ***enum*** value, use the ***self*** keyword

Rust

```
enum Value {
    Int(i32),
    Float(f32)
}
impl Value {
    fn is_int(&self)->bool {
        match self {
            Value::Int(_) => { return true; }
            _ => { return false;}
        }
    }
}
fn main() {
    let x = Value::Int(10);
    let y = Value::Float(1.5);
    println!("x is int: {}",x.is_int());
    println!("y is int: {}",y.is_int());
}
```

Output

```
x is int: true
y is int: false
```

Enums

The same logic could have been obtained via an "if let" statement, "while let" statement or "matches!" macro, instead of using a match.

Rust

```rust
enum Value {
    Int(i32),
    Float(f32)
}
impl Value {
    fn is_int(&self)->bool {
        match self {
            Value::Int(_) => { return true; }
            _ => { return false;}
        }
    }
}
fn main() {
    let x = Value::Int(10);
    let y = Value::Float(1.5);
    println!("x is int: {}",x.is_int());
    println!("y is int: {}",y.is_int());
}
```

Rust

```rust
fn is_int(&self)->bool {
    return if let Value::Int(_)=self { true } else { false }
}
```

Rust

```rust
fn is_int(&self)->bool {
    if let Value::Int(_) = self {
        return true;
    }
    return false;
}
```

Output

```
x is int: true
y is int: false
```

Enums

Static methods can also be implemented for an ***enum*** (they are in particular useful when creating enum objects).

Rust

```
#[derive(Debug)]
enum Value {
    Int(i32),
    Float(f32)
}
impl Value {
    fn from_i32(value: i32)->Value {
        return Value::Int(value);
    }
    fn from_f32(value: f32)->Value {
        return Value::Float(value);
    }
}
fn main() {
    let x = Value::from_i32(10);
    let y = Value::from_f32(1.5);
    println!("{x:?},{y:?}");
}
```

Output

```
Int(10),Float(1.5)
```

Traits

Traits

In Rust a trait is a set of characteristics that an object has. Formally, a trait is very similar to an interface. However, from a semantic point of view, it is closer to a C++ abstract class.

From the semantic point of view, a trait is a list of methods that can be implemented for an existing type (**IMPORTANT**: not necessarily a newly created type, but also types that are already defined).

To implement a trait for an existing structure/enum, use the **impl** keyword.

Rust

```rust
trait MyTrait {
    // methods
}
impl MyTrait for MyClass {
    // implement methods
}
```

Traits

Let’s see a simple example:

Rust

```rust
struct MyClass {
    x: i32
}
trait IncrementAndDecrement {
    fn inc(&mut self);
    fn dec(&mut self);
}
impl IncrementAndDecrement for MyClass {
    fn inc(&mut self) { self.x+=1; }
    fn dec(&mut self) { self.x-=1; }
}
fn main() {
    let mut m = MyClass{x:3};
    m.inc();m.inc();m.inc();
    m.dec();
    println!("X = {}",m.x);
}
```

Output
5

Traits

Let's see a simple example:

Rust

```
struct MyClass {
    x: i32
}
trait IncrementAndDecrement {
    fn inc(&mut self);
    fn dec(&mut self);
}
impl IncrementAndDecrement for MyClass {
    fn inc(&mut self) { self.x+=1; }
    fn dec(&mut self) { self.x-=1; }
}
fn main() {
    let mut m = MyClass{x:3};
    m.inc();m.inc();m.inc();
    m.dec();
    println!("X = {}",m.x);
}
```

C++

```
class IncrementAndDecrement {
    public:
        virtual void inc() = 0;
        virtual void dec() = 0;
};
class MyClass: public IncrementAndDecrement {
public:
    int x;
    virtual void inc() override { x++; };
    virtual void dec() override { x--; };
};
void main() {
    MyClass m;
    m.x = 3;
    m.inc();m.inc();m.inc();
    m.dec();
    printf("X = %d",m.x);
}
```

Output

```
5
```

Traits

Let’s see a simple example:

Rust

```
struct MyClass {
    x: i32
}
trait IncrementAndDecrement {
    fn inc(&mut self);
    fn dec(&mut self);
}
impl IncrementAndDecrement for MyClass {
    fn inc(&mut self) { self.x+=
    fn dec(&mut self) { self.x-=
}
fn main() {
    let mut m = MyClass{x:3};
    m.inc();m.inc();m.inc();
    m.dec();
    println!("X = {}",m.x);
}
```

```
mov         dword ptr [m],3
lea         rcx,[m]
call        first::impl$0::inc
lea         rcx,[m]
call        first::impl$0::inc
lea         rcx,[m]
call        first::impl$0::inc
lea         rcx,[m]
call        first::impl$0::dec
```

Output

5

Notice that the linkage is done statically (even if **inc** and **dec** are equivalent to a virtual method).

Traits

When implementing a trait, we can use the type **Self** to refer to the type where we implement the trait. This allows to define a trait and be more generic (not needing to specify the type of some parameters).

Rust

```rust
struct MyClass {
    x: i32
}
trait IsBigger {
    fn is_bigger(&self, object: &Self) -> bool;
}
impl IsBigger for MyClass {
    fn is_bigger(&self, object: &Self) -> bool {
        return if self.x>object.x { true } else { false };
    }
}
fn main() {
    let m1 = MyClass{x:3};
    let m2 = MyClass{x:2};
    println!("is m1 > m2 => {}",m1.is_bigger(&m2));
}
```

Output

```
is m1 > m2 => true
```

Traits

However, a virtual method (in C++) is interesting from the polymorphic point of view. This behavior can be modeled Rust using the **dyn** keyword:

Rust

```
struct ClassA { }
struct ClassB {  }
trait Name { fn get_name(&self) -> &str; }
impl Name for ClassA {
    fn get_name(&self) -> &str { "ClassA" }
}
impl Name for ClassB {
    fn get_name(&self) -> &str { "ClassB" }
}
fn print_name(obj: &dyn Name) {
    println!("{}",obj.get_name());
}
fn main() {
    let obj_a = ClassA{};
    let obj_b = ClassB{};
    print_name(&obj_a);
    print_name(&obj_b);
}
```

Output

```
ClassA
ClassB
```

Traits

However, a virtual method (in C++) is interesting from the polymorphic point of view. This behavior can be modeled Rust using the **dyn** keyword:

Rust

```
struct ClassA { }
struct ClassB {  }
trait Name { fn get_name(&self) -> &str; }
impl Name for ClassA {
    fn get_name(&self) -> &str { "ClassA" }
}
impl Name for ClassB {
    fn get_name(&self) -> &str { "ClassB" }
}
fn print_name(obj: &dyn Name) {
    println!("{}",obj.get_name());
}
fn main() {
    let obj_a = ClassA{};
    let obj_b = ClassB{};
    print_name(&obj_a);
    print_name(&obj_b);
}
```

Output

```
ClassA
ClassB
```

Notice the usage of **&dyn Name** as the type of obj. This translates that obj is a reference to a type that implements the trait ***Name***.

Traits

However, a virtual method (in C++) is interesting from the polymorphic point of view. This behavior can be modeled Rust using the **dyn** keyword:

Rust

```
struct ClassA { }
struct ClassB {  }
trait Name { fn get_name(&self) -> &str; }
impl Name for ClassA {
    fn get_name(&self) -> &str { "ClassA" }
}
impl Name for ClassB {
    fn get_name(&self) -> &str { "ClassB" }
}
fn print_name(obj: &dyn Name) {
    println!("{}",obj.get_name());
}
fn main() {
    let obj_a = ClassA{};
    let obj_b = ClassB{};
    print_name(&obj_a);
    print_name(&obj_b);
}
```

Output

```
ClassA
ClassB
```

This actually translate in the following way: when sending a dynamic reference towards a trait, Rust send two parameters:

1. a pointer to the object (self) via register RCX
2. a pointer to a vfptr (similar like in C++) via register RDX

```
lea         rcx,[obj_a]
lea         rdx,[impl$<first::ClassA, first::Name>::vtable$ (07FF60708D498h)]
call        first::print_name

lea         rcx,[obj_b]
lea         rdx,[impl$<first::ClassB, first::Name>::vtable$ (07FF60708D4B8h)]
call        first::print_name
```

Traits

However, a virtual method (in C++) is interesting from the polymorphic point of view. This behavior can be modeled Rust using the **dyn** keyword:

Rust

```rust
struct ClassA { }
struct ClassB {  }
trait Name { fn get_name(&self
impl Name for ClassA {
    fn get_name(&self) -> &str
}
impl Name for ClassB {
    fn get_name(&self) -> &str { "ClassB" }
}
fn print_name(obj: &dyn Name) {
    println!("{}",obj.get_name());
}
fn main() {
    let obj_a = ClassA{};
    let obj_b = ClassB{};
    print_name(&obj_a);
    print_name(&obj_b);
}
```

Similar to C++, all virtual/dynamic methods are kept in a list (that is referred by vfptr pointer). As a difference from C++, there is no need for redirection (as vfptr pointer is provided directly via a register).

```
sub         rsp,98h
mov         qword ptr [self],rcx
mov         qword ptr [vfptr],rdx
mov         rax,qword ptr [vfptr+18h]
call        rax // Name::get_name()
```

Output

```
ClassA
ClassB
```

Traits

This means that the size of an object that implements some traits does not change in Rust. "ClassA" in both Rust and C++ has one member ("x") that has 4 bytes. However, in C++ due to the virtual method **get_name**, an instance of ClassA also contains a pointer to a vfptr (and as such a size of 8 (for 32 bytes) or 12/16 for 64 bytes).

Rust

```rust
struct ClassA {
    x: i32,
}
trait Name {
    fn get_name(&self) -> &str;
}
impl Name for ClassA {
    fn get_name(&self) -> &str {
        "ClassA"
    }
}
fn main() {
    println!("{}", std::mem::size_of::<ClassA>())
}
```

Output

4

C++

```cpp
class Name {
    public:
        virtual const char * get_name() = 0;
};
class ClassA: public Name {
    int x;
public:
    virtual const char * get_name() override {
        return "ClassA";
    }
};
void main() {
    printf("%d",sizeof(ClassA));
}
```

Output

8

Traits

Furthermore, the same logic applies for arrays (or for any kind of structure/enum that uses a structure that implements a trait that define a virtual method.

Rust

```
struct ClassA {
    x: i32,
}
trait Name {
    fn get_name(&self) -> &str;
}
impl Name for ClassA {
    fn get_name(&self) -> &str {
        "ClassA"
    }
}
fn main() {
    println!("{}", std::mem::size_of::<[ClassA;10]>());
}
```

Output
40

Traits

So … let's analyze and see how the classic polymorphism example works in Rust.

C++ (classic polymorphism example)

```
struct Figure {
    virtual const char * get_name() = 0;
};
struct Circle: public Figure {
    virtual const char * get_name() override {  return "Circle";}
};
struct Rectangle: public Figure {
    virtual const char * get_name() override {  return "Rectangle";}
};
struct Triangle: public Figure {
    virtual const char * get_name() override {  return "Triangle";}
};
void main() {
    Figure* fig[3];
    fig[0] = new Circle();
    fig[1] = new Rectangle();
    fig[2] = new Triangle();
    for (auto i = 0;i<2;i++) {
        printf("%s\n",fig[i]->get_name());
    }
}
```

Output

```
Circle
Rectangle
Triangle
```

Traits

Let's recreate the same example for polymorphism in Rust.

We will do this in 3 steps:

1. Write the Figure trait and implement it for Circle, Rectangle and Triangle
2. Write initialization methods for Circle, Rectangle and Triangle
3. Discuss how **main** function should be written in order to illustrate the polymorphism.

Traits

Step 1: Write the Figure trait and implement it for Circle, Rectangle and Triangle

Rust

```
trait Figure {
    fn get_name(&self) -> &str;
}

struct Circle { x: i32, y:i32, r: i32 }
struct Rectangle { x: i32, y:i32, w:i32, h:i32 }
struct Triangle { x: [i32;3], y:[i32;3] }

impl Figure for Circle {
    fn get_name(&self) -> &str { "Circle" }
}
impl Figure for Rectangle {
    fn get_name(&self) -> &str { "Rectangle" }
}
impl Figure for Triangle {
    fn get_name(&self) -> &str { "Triangle" }
}
```

Traits

Step 2: Write initialization methods for Circle, Rectangle and Triangle

Rust

```
impl Circle {
    fn new()->Circle {
        Circle{x:0,y:0,r:1}
    }
}
impl Rectangle {
    fn new()->Rectangle {
        Rectangle{x:0,y:0,w:100,h:20}
    }
}
impl Triangle {
    fn new()->Triangle {
        Triangle{x:[0,1,2],y:[0,1,0]}
    }
}
```

Traits

Step 3: Discuss how **main** function should be written in order to illustrate the polymorphism.

Rust

```
fn main() {
    let figuri = [
        Box::new(Circle::new()),
        Box::new(Rectangle::new()),
        Box::new(Triangle::new())
    ];
    for fig in figuri.iter() {
        println!("{}",fig.get_name());
    }
}
```

Error

```
error[E0308]: mismatched types
  --> src\main.rs:38:18
   |
38 |         Box::new(Rectangle::new()),
   |                  ^^^^^^^^^^^^^^^^ expected struct `Circle`,
   |                                   found struct `Rectangle`

error[E0308]: mismatched types
  --> src\main.rs:39:18
   |
39 |         Box::new(Triangle::new())
   |                  ^^^^^^^^^^^^^^^ expected struct `Circle`, found
   |                                  struct `Triangle`
```

The fact is that we can not create an array with traits similar to how we do it in C++ (Rust assumes that the first item is the type of array and as such for this example, the code will not compile).

Traits

Step 3: Discuss how **main** function should be written in order to illustrate the polymorphism.

Rust

```
fn main() {
    let figuri: [Box::<dyn Figure>;3] = [
        Box::new(Circle::new()),
        Box::new(Rectangle::new()),
        Box::new(Triangle::new())
    ];
    for fig in figuri.iter() {
        println!("{}",fig.get_name());
    }
}
```

Output

```
Circle
Rectangle
Triangle
```

Now the code works and output a similar result as the code from C++;

Traits

Step 3: Discuss how **main** function should be written in order to illustrate the polymorphism.

Rust

```
fn main() {
    let figuri: [Box::<dyn Figure>;3] = [
        Box::new(Circle::new()),
        Box::new(Rectangle::new()),
        Box::new(Triangle::new())
    ];
    for fig in figuri.iter() {
        println!("{}",fig.get_name());
    }
}
```

"figure" layout	
[0]	ptr to a Circle object
	ptr to vtable of trait **Figure** for Circle object
[1]	ptr to a Rectangle object
	ptr to vtable of trait **Figure** for Rectangle object
[2]	ptr to a Triangle object
	ptr to vtable of trait **Figure** for Triangle object

Let's see how "figure" is organized in memory. Notice that each element in the array consists out of two pointers (one towards the data (a Circle struct, a Rectangle struct or a Triangle struct) and the second one towards the vtable for trait Figure that was implemented for Circle, Rectangle and Triangle.

Traits

Step 3: Discuss how **main** function should be written in order to illustrate the polymorphism.

Rust

```
fn main() {
    let figuri: [dyn Figure;3] = [
        Circle::new(),
        Rectangle::new(),
        Triangle::new()
    ];
    for fig in figuri.iter() {
        println!("{}",fig.get_name());
    }
}
```

Error

```
error[E0277]: the size for values of type `dyn Figure` cannot be known at compilation time
  --> src\main.rs:36:17
   |
36 |     let figuri: [dyn Figure;3] = [
   |                 ^^^^^^^^^^^^^^ doesn't have a size known at compile-time
   |
   = help: the trait `Sized` is not implemented for `dyn Figure`
   = note: slice and array elements must have `Sized` type
```

Keep in mind that we can not use a “dyn Figure type” outside of a box as we can not know at compile time the size of an object that implements Figure trait.

Traits

Step 3: Discuss how **main** function should be written in order to illustrate the polymorphism.

Rust

```
fn main() {
    let mut figuri = Vec::<Box<dyn Figure>>::new();
    figuri.push(Box::new(Circle::new()));
    figuri.push(Box::new(Rectangle::new()));
    figuri.push(Box::new(Triangle::new()));
    for fig in figuri.iter() {
        println!("{}",fig.get_name());
    }
}
```

Output

```
Circle
Rectangle
Triangle
```

The same can be done with a vector (instead of an array) with similar results.

Traits

The previous code can be adjusted so that we can returned a boxed trait from a function. Let's see how ***get_a_figure*** looks like in assembly:

Rust

```
fn get_a_figure(id: i32) -> Box<dyn Figure> {
    if id == 0 { return Box::new(Circle::new()); }
    if id == 1 { return Box::new(Rectangle::new()); }
    Box::new(Triangle::new())
}
fn main() {
    let mut figuri = Vec::<Box<dyn Figure>>::new();
    for i in 0..3 {
        figuri.push(get_a_figure(i));
    }
    for fig in figuri.iter() {
        println!("{}", fig.get_name());
    }
}
```

Output

```
Circle
Rectangle
Triangle
```

Traits

The previous code can be adjusted so that we
function. Let’s see how ***get_a_figure*** looks lik

Rust

```rust
fn get_a_figure(id: i32) -> Box<dyn Figure> {
    if id == 0 { return Box::new(Circle::new()); }
    if id == 1 { return Box::new(Rectangle::new()); }
    Box::new(Triangle::new())
}
fn main() {
    let mut figuri = Vec::<Box<dyn Figure>>::new();
    for i in 0..3 {
        figuri.push(get_a_figure(i));
    }
    for fig in figuri.iter() {
        println!("{}", fig.get_name());
    }
}
```

```asm
            lea     rcx,[temp_stack_circle]
            call    first::Circle::new
            mov     ecx,12 // size of a circle
            mov     edx,4
            call    alloc::alloc::exchange_malloc
            mov     qword ptr [ptr_to_circle],rax
            jmp     RETURN_FIGURE_FROM_CIRCLE
            ...

RETURN_FIGURE_FROM_CIRCLE:
            mov     rcx,qword ptr [ptr_to_circle]
            mov     rax,rcx
            mov     rdx,qword ptr [temp_stack_circle]
            mov     qword ptr [rcx],rdx
            mov     edx,dword ptr [temp_stack_circle.r]
            mov     dword ptr [ptr_to_circle.r],edx
            mov     qword ptr [res.data_pointer],rax
            lea     rax,[impl<Circle, Figure>::vtable]
            mov     qword ptr [res.vtable],rax
            jmp     RETURN_FROM_FUNCTION
            ...

RETURN_FROM_FUNCTION:
            mov     rax,qword ptr [res.data_pointer]
            mov     rdx,qword ptr [res.vtable]
            add     rsp,0A0h
            pop     rbp
            ret
```

Traits

The previous code can be adjusted so that we can returned a boxed trait from a function. Let's see how ***get_a_figure*** looks like in assembly:

Rust

```
fn get_a_figure(id: i32) -> Box<dyn Figure> {
    if id == 0 { return Box::new(Circle::new()); }
    if id == 1 { return Box::new(Rectangle::new()); }
    Box::new(Triangle::new())
}
fn main() {
    let mut figuri = Vec::<Box<dyn Figure>>::new();
    for i in 0..3 {
        figuri.push(get_a_figure(i));
    }
    for fig in figuri.iter() {
        println!("{}", fig.get_name());
    }
}
```

C++ (approximation)

```
struct Figure_result {
    void* ptr_to_data;
    void* ptr_to_vtable;
};
Figure_result get_a_figure(int idx) {
    if (idx == 0) {
        Circle temp_stack_circle = Circle::new();
        Circle* ptr_to_circle = new Circle();
        memcpy(ptr_to_circle,
               temp_stack_circle,
               sizeof(Circle));
        Figure_result res;
        res.ptr_to_data = ptr_to_circle;
        res.ptr_to_vtable = 0xFF1122....;
        return res;
    }
    ...
}
```

A hardcoded address in process memory where the **vtable** for Circle is located.

Traits

Keep in mind that returning a boxed (dynamic) type is different than returning an implementation of a trait . The next code will not compile as Rust will assume that all return branches must return the same thing (a circle) just like the first return branch does.

Rust

```rust
fn get_a_figure(id: i32) -> impl Figure {
    if id == 0 {
        return Circle::new();
    }
    if id == 1 {
        return Rectangle::new();
    }
    return Triangle::new();
}
```

Error

```
error[E0308]: mismatched types
  --> src\main.rs:75:16
   |
70 | fn get_a_figure(id: i32) -> impl Figure {
   |                             ----------- expected `_` because of return type
...
75 |         return Rectangle::new();
   |                ^^^^^^^^^^^^^^^^ expected struct `Circle`, found struct `Rectangle`
```

Traits

If we return the exact same type from all branches of the get_a_figure function, the code compiles.

Rust

```
fn get_a_figure(id: i32) -> impl Figure {
    if id == 0 {
        return Circle::new();
    }
    return Circle::new();
}
fn main() {
    let a = get_a_figure(0);
    println!("{}",a.get_name());
}
```

Output
Circle

Let’s see what happens when we create the “a” variable.

Traits

If we return the exact same type from all branches of the get_a_figure function, the code compiles.

Rust

```
fn get_a_figure(id: i32) -> impl Figure {
    if id == 0 {
        return Circle::new();
    }
    return Circle::new();
}
fn main() {
    let a = get_a_figure(0);
    println!("{}",a.get_name());
}
```

```
lea         rcx,[a]
xor         edx,edx // edx = 0 (first parameter: id = 0)
call        first::get_a_figure
```

Let’s see what happens when we create the “a” variable.

Traits

If we return the exact same type from all branches of the get_a_figure function, the code compiles.

Rust

```rust
fn get_a_figure(id: i32) -> impl Figure {
    if id == 0 {
        return Circle::new();
    }
    return Circle::new();
}
fn main() {
    let a = get_a_figure(0);
    println!("{}",a.get_name());
}
```

```asm
            sub         rsp,38h
            mov         qword ptr [address_of_a],rcx

            cmp         edx,0
            jne         IDX_IS_NOT_ZERO
            mov         rcx,qword ptr [address_of_a]
            call        Circle::new
            jmp         RETURN_FROM_FUNCTION

IDX_IS_NOT_ZERO:
            mov         rcx,qword ptr [address_of_a]
            call        Circle::new

RETURN_FROM_FUNCTION:
            mov         rax,qword ptr [address_of_a]
            add         rsp,38h
            ret
```

Let’s see what happens when we create the “a” variable.

Traits

If we return the exact same type from all branches of the get_a_figure function, the code compiles.

Rust

```rust
fn get_a_figure(id: i32) -> impl Figure {
    if id == 0 {
        return Circle::new();
    }
    return Circle::new();
}
fn main() {
    let a = get_a_figure(0);
    println!("{}",a.get_name());
}
```

This means that even if semantically “a” is of type “impl Figure”, in reality “a” is a Circle object (with the exception that we can only access Figure related methods).

C++ (approximation)

```cpp
void get_a_figure(void* result, int idx) {
    if (idx == 0) {
        Circle temp = Circle::new();
        memcpy(result, temp, sizeof(Circle));
        return;
    }
    Circle temp = Circle::new();
    memcpy(result, temp, sizeof(Circle));
    return;
}
void main() {
    uint8_t data[sizeof(Circle)];
    get_a_figure(data,0);
    Figure* figure = reinterpret_cast<Figure*>(data);
}
```

Traits

If we return the exact same type from all branches of the get_a_figure function, the code compiles.

Rust

```rust
fn get_a_figure(id: i32) -> impl Figure {
    if id == 0 {
        return Circle::new();
    }
    return Circle::new();
}
fn main() {
    let a = get_a_figure(0);
    println!("{}",a.get_name());
}
```

Rust

```rust
fn get_a_figure(id: i32) -> Circle {
    if id == 0 {
        return Circle::new();
    }
    return Circle::new();
}
fn main() {
    let a = get_a_figure(0);
    println!("{}",a.get_name());
}
```

As such → these two pieces of code are similar (in terms on how the compiler generates code). The assembly code (for x64) is actually identical for both cases (even if from the semantic point of view, "a" has a different type).

Traits

Methods from a trait can have a default implementation (much like a virtual method from C++). This means that if that method is not overridden, the default implementation will be used. To implement a trait without override its method, use:

```
impl <trait_name> for <type> { }
```

Keep in mind that this is possible only if all method from the trait have a default implementation !

Rust

```rust
struct ClassA {}
struct ClassB {}
trait Name { fn get_name(&self) -> &str { "Default name“ } }
impl Name for ClassA {}
impl Name for ClassB { fn get_name(&self) -> &str { "ClassB" } }
fn main() {
    let a = ClassA{};
    let b = ClassB{};
    println!("a = {}",a.get_name());
    println!("b = {}",b.get_name());
}
```

Traits

A trait can have both default (implemented methods) and unimplemented method and they can use one each other.

Rust

```
struct ClassA {}
trait Message {
    fn get_name(&self) -> &str {
        "Default name"
    }
    fn print_message(&self);
}
impl Message for ClassA {
    fn print_message(&self) {
        println!("Hello from '{}'",self.get_name());
    }
}
fn main() {
    let a = ClassA{};
    a.print_message();
}
```

Notice that print_message is implemented in ClassA and uses **get_name** that has a default implementation in trait ***Message***.

Traits

What's different in Rust in terms of how a ***trait*** work, is that a ***trait*** can be implemented for other types as well (even if they are not defined in that program → e.g. for example a system type).

In this case, we create a new trait, called **BitCount** that can be implemented for type u32.

As a result, every variable or constant of type u32 will have a function called compute_bit_count that counts how many bits with value 1 a value has.

Rust

```
trait BitCount {
    fn compute_bit_count(&self) -> u32;
}
impl BitCount for u32 {
    fn compute_bit_count(&self) -> u32 {
        let mut value = *self;
        let mut count = 0u32;
        while value>0 {
            count = count + (value % 2);
            value = value / 2;
        }
        return count;
    }
}
fn main() {
    let x = 24u32; // 24 = 11000
    println!("Bits in x = {}",x.compute_bit_count());
}
```

Output
2

Traits

Notice that you have to implement this trait for every type in order to work. The following code will not compile as i32 does not implement the trait.

Rust

```
trait BitCount {
    fn compute_bit_count(&self) -> u32;
}
impl BitCount for u32 {
    fn compute_bit_count(&self) -> u32 {
        let mut value = *self;
        let mut count = 0u32;
        while value>0 {
            count = count + (value % 2);
            value = value / 2;
        }
        return count;
    }
}
fn main() {
    println!("Bits in 24u32 = {}",24u32.compute_bit_count());
    println!("Bits in 24i32 = {}",24i32.compute_bit_count());
}
```

Error

```
error[E0599]: no method named `compute_bit_count` found for type `i32` in the current
scope
  --> src\main.rs:17:41
   |
17 |     println!("Bits in 24i32 = {}",24i32.compute_bit_count());
   |                                          ^^^^^^^^^^^^^^^^^ method not found in `i32`
   |
   = help: items from traits can only be used if the trait is implemented and in scope
note: `BitCount` defines an item `compute_bit_count`, perhaps you need to implement it
  --> src\main.rs:1:1
   |
1  | trait BitCount {
   | ^
```

Traits

Another interesting example is the following. There is **<u>no method</u>** in class String that can be used to set/change the existing string with a different one. You can obviously run a .clear() followed by a .push_str(...) to do this, but you can also do it using traits ☺

Rust

```
trait StringSetter {
    fn set(&mut self, text: &str);
}
impl StringSetter for String {
    fn set(&mut self, text: &str) {
        self.clear();
        self.push_str(text);
    }
}
fn main() {
    let mut s = String::from("abc");
    println!("S = {}",s);
    s.set("123456");
    println!("S = {}",s);
}
```

Output

```
S = abc
S = 123456
```

Traits

A trait can also have constants defined as part of the trait. That constant should be seen as a static variable (it does not affect in any way the size of the structure that implements that trait).

Rust

```rust
struct RON {
    amount: i32
}
trait Currency {
    const DEFAULT:i32 = 100;
    fn set(&mut self, value: i32);
}
impl Currency for RON { fn set(&mut self, value: i32) { self.amount = value; } }
fn main() {
    let mut m = RON{amount:0};
    println!("m = {}",m.amount);
    m.set(RON::DEFAULT);
    println!("m = {}",m.amount);
    println!("size of RON = {}",std::mem::size_of::<RON>());
}
```

Output

```
m = 0
m = 100
size of RON = 4
```

Traits

A constant value defined in a trait does not necessarily need to be instantiated as part of the trait definition. However, that constant needs to be initialized in implementation.

Rust

```rust
struct RON {
    amount: i32
}
trait Currency {
    const DEFAULT:i32;
    fn set(&mut self, value: i32);
}
impl Currency for RON {
    fn set(&mut self, value: i32) { self.amount = value; }
}
fn main() {
    let mut m = RON{amount:0};
    println!("m = {}",m.amount);
    m.set(RON::DEFAULT);
    println!("m = {}",m.amount);
    println!("size of RON = {}",std::mem::size_of::<RON>());
}
```

Error

```
error[E0046]: not all trait items implemented, missing: `DEFAULT`
 --> src\main.rs:8:1
  |
5 |     const DEFAULT:i32;
  |     ------------------ `DEFAULT` from trait
...
8 | impl Currency for RON {
  | ^^^^^^^^^^^^^^^^^^^^^ missing `DEFAULT` in implementation
```

Traits

A constant value defined in a trait does not necessarily need to be instantiated as part of the trait definition. However, that constant needs to be initialize in implementation.

Rust

```
struct RON {
    amount: i32
}
trait Currency {
    const DEFAULT:i32;
    fn set(&mut self, value: i32);
}
impl Currency for RON {
    const DEFAULT:i32 = 1234;
    fn set(&mut self, value: i32) { self.amount = value; }
}
fn main() {
    let mut m = RON{amount:0};
    println!("m = {}",m.amount);
    m.set(RON::DEFAULT);
    println!("m = {}",m.amount);
    println!("size of RON = {}",std::mem::size_of::<RON>());
}
```

Output

```
m = 0
m = 1234
size of RON = 4
```

Traits

Similar to constant values, a trait can have types defined within the trait. And just like constant values, the actual type of a defined type within a trait can be set up at the trait or implementation level.

Let’s analyze the following problem:

- We need to convert from both Celsius and Fahrenheit to Kelvin
- Let’s also consider that Celsius is represented as an i32, while Fahrenheit is stored in an f32 value.
- To do this, we will define two types (Celsius and Fahrenheit) and a trait (that describe how the conversion to Kelvin is performed.
- We will also define a third type (Kelvin) that just returns its value. We will use it for a different discussion.

Traits

Step 1: Define structures for Celsius, Fahrenheit and Kelvin as well as the conversion trait.

Rust

```rust
struct Celsius {
    value: i32,
}
struct Fahrenheit {
    value: f32,
}
struct Kelvin {
    value: f32
}
trait TemperatureConverter {
    type ConversionOutput;
    fn to_kelvin(&self) -> Self::ConversionOutput;
}
```

Notice that trait TemperatureConverter has an inner type (ConversionOutput) that is not yet defined !

Traits

Step 2: Implement TemperatureConverter for both Celsius, Fahrenheit and Kelvin types.

Rust

```
impl TemperatureConverter for Celsius {
    type ConversionOutput = i32;
    fn to_kelvin(&self) -> Self::ConversionOutput { return self.value + 273; }
}

impl TemperatureConverter for Fahrenheit {
    type ConversionOutput = f32;
    fn to_kelvin(&self) -> Self::ConversionOutput { return ((self.value - 32.0) / 1.8) + 273.15; }
}

impl TemperatureConverter for Kelvin {
    type ConversionOutput = f32;
    fn to_kelvin(&self) -> Self::ConversionOutput { self.value }
}
```

Notice that we have different formulas for those three types, and that we define **ConversionOutput** for all implementations (i32 for *Celsius* and f32 for *Fahrenheit* and *Kelvin*).

Traits

Step 3: Write a **main** function that showcase how the trait works.

Rust

```rust
fn main() {
    let c = Celsius { value: 24 };
    println!("Celsius({}) = Kelvin({})", c.value, c.to_kelvin());
    let f = Fahrenheit { value: 100.5 };
    println!("Fahrenheit({}) = Kelvin({})", f.value, f.to_kelvin());
    let k = Kelvin { value: 50.2 };
    println!("Kelvin({}) = Kelvin({})", k.value, k.to_kelvin());
}
```

Output

```
Celsius(24) = Kelvin(297)
Fahrenheit(100.5) = Kelvin(311.20557)
Kelvin(50.2) = Kelvin(50.2)
```

OBS: *This technique is similar to the usage of templates / generics. We will however discuss about templates/generics and their usage with structs/enums and traits in another course.*

Traits

Keep in mind that using this technique (an inner type that is defined in the implementation of the trait) will not allow any kind of polymorphism as there is no similar definition for the trait methods.

Rust

```
fn main() {
    let a:[Box<dyn TemperatureConverter>;2] = [
        Box::new(Celsius { value: 24 }),
        Box::new(Fahrenheit { value: 100.5 })
    ];
    for i in a.iter() {
        println!("{}",i.to_kelvin());
    }
}
```

Error

```
error[E0191]: the value of the associated type `ConversionOutput` (from trait
`TemperatureConverter`) must be specified
  --> src\main.rs:24:20
   |
8  |     type ConversionOutput;
   |     ---------------------- `ConversionOutput` defined here
...
24 |     let a:[Box<dyn TemperatureConverter>;2] = [
   |                    ^^^^^^^^^^^^^^^^^^^^ help: specify the associated type:
   |                    `TemperatureConverter<ConversionOutput = Type>`
```

Traits

Keep in mind that even if we modify the way we define the Box (by adding an explicit request for the ConversionOutput type, all elements from the list MUST have the same ConversionOutputType !

Rust

```
fn main() {
    let a: [Box<dyn TemperatureConverter<ConversionOutput = f32>>; 3] = [
        Box::new(Celsius { value: 24 }),
        Box::new(Kelvin { value: 150.2 }),
        Box::new(Fahrenheit { value: 100.5 }),
    ];
    for i in a.iter() { println!("{}", i.to_kelvin()) }
}
```

Error

```
error[E0271]: type mismatch resolving `<Celsius as TemperatureConverter>::ConversionOutput == f32`
  --> src\main.rs:35:9
   |
35 |         Box::new(Celsius { value: 24 }),
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ type mismatch resolving `<Celsius as
   |                                         TemperatureConverter>::ConversionOutput == f32`
note: expected this to be `f32`
  --> src\main.rs:16:29
16 |     type ConversionOutput = i32;
```

Traits

Now it works. Keep in mind that both Kelvin and Fahrenheit type have the same type for the ConversionOutput (f32).

Rust

```
fn main() {
    let a: [Box<dyn TemperatureConverter<ConversionOutput = f32>>; 2] = [
        Box::new(Kelvin { value: 150.2 }),
        Box::new(Fahrenheit { value: 100.5 }),
    ];
    for i in a.iter() {
        println!("{}", i.to_kelvin());
    }
}
```

Output

```
150.2
311.20557
```

OBS: While this technique is working, it is not usually used for polymorphism (as it implies to make sure that types that have a super-trait have the same internal type – thus making the concept of internal type less relevant as it can be hardcoded).

Traits

A trait can also contain static methods, that can have a default behavior or not, and in the last case, those methods should be implemented for types that implement the trait. Obviously, since a static method in a trait is not linked to an instance of the type that implements that trait, things like polymorphism can not be achieved with these methods.

Rust

```rust
trait Addition {
    fn compute(v1:i32, v2:i32) -> i32;
}
struct ClassA { }
impl Addition for ClassA {
    fn compute(v1:i32, v2:i32) -> i32 {
        v1+v2
    }
}

fn main() {
    println!("{}",ClassA::compute(10, 20));
}
```

Output

```
30
```

Traits

A structure/enum can implement multiple traits. What happens if there are two traits that define a method with the same name ?

Rust

```rust
trait TraitA { fn compute(&self, value:i32) -> i32; }
trait TraitB { fn compute(&self, value:i32) -> i32; }
struct ClassA { value: i32 }
impl TraitA for ClassA {
    fn compute(&self, value:i32) -> i32 {
        return self.value * value;
    }
}
impl TraitB for ClassA {
    fn compute(&self, value:i32) -> i32 {
        return self.value / value;
    }
}
fn main() {
    let x = ClassA{value:10};
    println!("{}",x.compute(5));
}
```

Error

```
error[E0034]: multiple applicable items in scope
  --> src\main.rs:23:21
   |
23 |     println!("{}",x.compute(5));
   |                     ^^^^^^^ multiple `compute` found
   |
note: candidate #1 is defined in an impl of the trait `TraitA`
for the type `ClassA`
  --> src\main.rs:11:5
   |
11 |     fn compute(&self, value:i32) -> i32 {
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
note: candidate #2 is defined in an impl of the trait `TraitB`
for the type `ClassA`
  --> src\main.rs:16:5
   |
16 |     fn compute(&self, value:i32) -> i32 {
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: disambiguate the associated function for candidate #1
   |
23 |     println!("{}",TraitA::compute(&x, 5));
   |                   ~~~~~~~~~~~~~~~~~~~~~~
help: disambiguate the associated function for candidate #2
   |
23 |     println!("{}",TraitB::compute(&x, 5));
```

Traits

The solution is to specifically explain Rust that, it needs to call a function defined from a specific trait. The format for this call is:

<type-name as trait-name>::*method*(&obj, $Param_1$, $Param_2$, … $Param_n$)

Where obj is on object of type type-name that implements trait-name

Rust

```
trait TraitA { fn compute(&self, value:i32) -> i32; }
trait TraitB { fn compute(&self, value:i32) -> i32; }
struct ClassA { value: i32 }
impl TraitA for ClassA {…}
impl TraitB for ClassA {…}

fn main() {
    let x = ClassA{value:10};
    println!("{}",<ClassA as TraitA>::compute(&x,5));
    println!("{}",<ClassA as TraitB>::compute(&x,5));
}
```

Output

```
50
2
```

Traits

The solution is to specifically explain Rust that, it needs to call a function defined from a specific trait. The format for this call is:

<type-name as trait-name>::*method*(&obj, $Param_1$, $Param_2$, … $Param_n$)

Where obj is on object of type type-name that implements trait-name

Rust

```
trait TraitA { fn compute(&self, value:i32) -> i32; }
trait TraitB { fn compute(&self, value:i32) -> i32; }
struct ClassA { value: i32 }
impl TraitA for ClassA {…}
impl TraitB for ClassA {…}

fn main() {
    let x = ClassA{value:10};
    println!("{}",<ClassA as TraitA>::compute(&x,5));
    println!("{}",<ClassA as TraitB>::compute(&x,5));
}
```

Output

Alternatively, the following format can be used:

Rust

```
fn main() {
    let x = ClassA{value:10};
    println!("{}", TraitA::compute(&x,5));
    println!("{}", TraitB::compute(&x,5));
}
```

Super traits

Super traits

Rust does not have an inheritance model , similar to what other languages have where a type can be derived from another type and as such inherits all of its parent properties, data members and methods.

However, Rust allows a certain type of inheritance by providing the concept of a super trait. If “A” is a super trait for “B” , then any structure or enum that implements “B” must also implement “A”

The format is similar to the way inheritance is done in C++ (name of the trait, followed by ‘:’ and the name of the super trait).

Rust

```
trait MyTrait : MySuperTrait {
    // methods
}
impl MyTrait for MyClass {
    // implement methods
}
```

Super traits

Let's see an example:

Rust

```rust
trait Vehicle {
    fn get_name(&self) -> &str;
}
trait Car: Vehicle {
    fn get_max_speed(&self) -> u32;
}
struct Dacia { }
impl Car for Dacia {
    fn get_max_speed(&self) -> u32 {
        return 140;
    }
}

fn main() {
    let d = Dacia{};
    println!("max_speed = {}",d.get_max_speed());
}
```

Error

```
error[E0277]: the trait bound `Dacia: Vehicle` is not satisfied
 --> src\main.rs:8:6
  |
8 | impl Car for Dacia {
  |      ^^^ the trait `Vehicle` is not implemented for `Dacia`
  |
note: required by a bound in `Car`
 --> src\main.rs:4:12
  |
4 | trait Car: Vehicle {
  |            ^^^^^^^ required by this bound in `Car`
```

The code will not compile because we haven't implemented the trait Vehicle for structure ***Dacia***. This is required because Vehicle is a super trait for the trait Car.

Super traits

Let's see an example:

Rust

```
trait Vehicle {
    fn get_name(&self) -> &str;
}
trait Car: Vehicle {
    fn get_max_speed(&self) -> u32;
}
struct Dacia {}
impl Car for Dacia {
    fn get_max_speed(&self) -> u32 { return 140; }
}
impl Vehicle for Dacia {
    fn get_name(&self) -> &str { return "Dacia"; }
}

fn main() {
    let d = Dacia {};
    println!("max_speed = {}", d.get_max_speed());
    println!("name= {}", d.get_name());
}
```

Output

```
max_speed = 140
name= Dacia
```

Super traits

Any trait derived from another trait has access to all of the methods defined in the super trait. Similar, via Self type, it can access any constant defined in the super trait and instantiated in the struct or current trait.

Rust

```rust
trait Vehicle {
    const MAX_SPEED: u32;
    fn get_name(&self) -> &str;
}
trait Car: Vehicle {
    fn print_speed(&self) { println!("Max speed for {} is {}", self.get_name(),Self::MAX_SPEED); }
}
struct Dacia {}
impl Car for Dacia { }
impl Vehicle for Dacia {
    const MAX_SPEED: u32 = 140;
    fn get_name(&self) -> &str { return "Dacia"; }
}
fn main() {
    let d = Dacia {};
    d.print_speed();
}
```

Output

```
Max speed for Dacia is 140
```

Super traits

Multiple inheritance is also possible as a trait can be a super trait for multiple traits.

Rust

```rust
trait Vehicle {
    fn get_name(&self) -> &str;
}
trait Car: Vehicle {
    fn get_speed(&self) ->  u32;
}
trait Color:  Vehicle {
    fn get_color(&self) -> &str;
}
struct Dacia {}
impl Car for Dacia { fn get_speed(&self) ->  u32 { 140 } }
impl Color for Dacia { fn get_color(&self) ->  &str { "Blue" } }
impl Vehicle for Dacia { fn get_name(&self) -> &str { "Dacia "} }
fn main() {
    let d = Dacia {};
    println!("Name  = {}",d.get_name());
    println!("Speed = {}",d.get_speed());
    println!("Color = {}",d.get_color());
}
```

Output

```
Name  = Dacia
Speed = 140
Color = Blue
```

Super traits

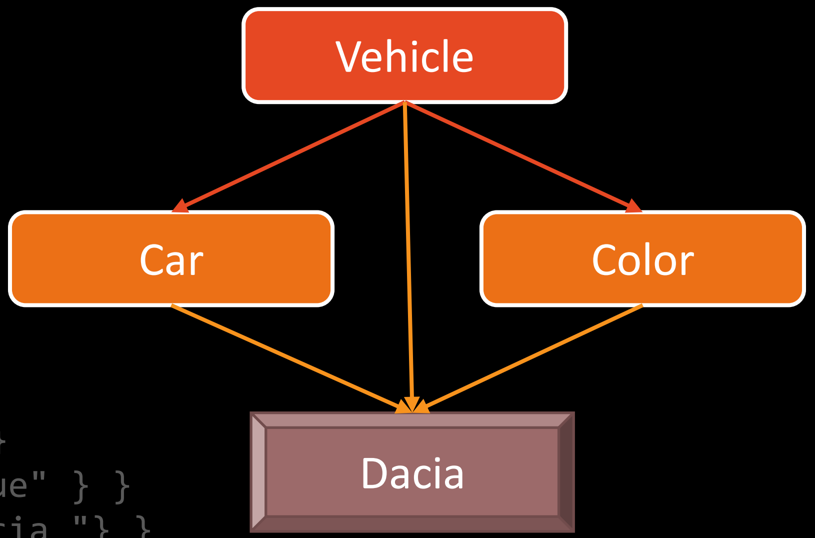
Multiple inheritance is also possible as a trait can be a super trait for multiple traits.

Rust

```
trait Vehicle {
    fn get_name(&self) -> &str;
}
trait Car: Vehicle {
    fn get_speed(&self) ->  u32;
}
tra

}
str
impl Car for Dacia { fn get_speed(&self) ->  u32 { 140 } }
impl Color for Dacia { fn get_color(&self) ->  &str { "Blue" } }
impl Vehicle for Dacia { fn get_name(&self) -> &str { "Dacia "} }
fn main() {
    let d = Dacia {};
    println!("Name  = {}",d.get_name());
    println!("Speed = {}",d.get_speed());
    println!("Color = {}",d.get_color());
}
```

This approach solves and the fact that a trait does not have any data members solves the diamond problem associated with multiple inheritance.



Super traits

A similar code in C++ would look like this.

Rust

```rust
trait Vehicle {
    fn get_name(&self) -> &str;
}
trait Car: Vehicle {
    fn get_speed(&self) ->  u32;
}
trait Color:  Vehicle {
    fn get_color(&self) -> &str;
}
struct Dacia {}
impl Car for Dacia { … }
impl Color for Dacia { … }
impl Vehicle for Dacia { … }
fn main() {
    let d = Dacia {};
    println!("Name  = {}",d.get_name());
    println!("Speed = {}",d.get_speed());
    println!("Color = {}",d.get_color());
}
```

C++

```cpp
class Vehicle {
    virtual const char * get_name() = 0;
};
class Car: public Vehicle {
    virtual unsigned int get_speed() = 0;
};
class Color: public Vehicle {
    virtual const char * get_color() = 0;
};
class Dacia: public Car,public Color {
    virtual const char * get_name()  override {...}
    virtual unsigned int get_speed() override {...}
    virtual const char * get_color() override {...}
};
void main() {
    Dacia d;
    printf("Name  = {}",d.get_name());
    printf("Speed = {}",d.get_speed());
    printf("Color = {}",d.get_color());
}
```

Super traits

At the same time, multiple traits can be super trait for another trait. Semantically this is explained in the following way:

```
trait <name>: SuperTrait1 + SuperTrait2 + … SuperTraitn {…}
```

This is in particular useful when using templates/generics as it can be used to explain certain type of limitations (e.g. the type used in a generic must implement $Trait_1$, $Trait_2$, …).
This format is often referred as **trait combos**.

Rust

```rust
trait MyTrait : MySuperTrait + MySecondarySuperTrait + MyThirdSuperTrait {
    // methods
}
impl MyTrait for MyClass {
    // implement methods
}
```

Super traits

The same example → but with trait combos.

Rust

```
trait Vehicle {
    fn get_name(&self) -> &str;
}
trait Color {
    fn get_color(&self) -> &str;
}
trait Car: Vehicle + Color {
    fn get_speed(&self) -> u32;
}
struct Dacia {}
impl Car for Dacia { fn get_speed(&self) ->  u32 { 140 } }
impl Color for Dacia { fn get_color(&self) ->  &str { "Blue" } }
impl Vehicle for Dacia { fn get_name(&self) -> &str { "Dacia "} }
fn main() {
    let d = Dacia {};
    println!("Name  = {}",d.get_name());
    println!("Speed = {}",d.get_speed());
    println!("Color = {}",d.get_color());
}
```

Output

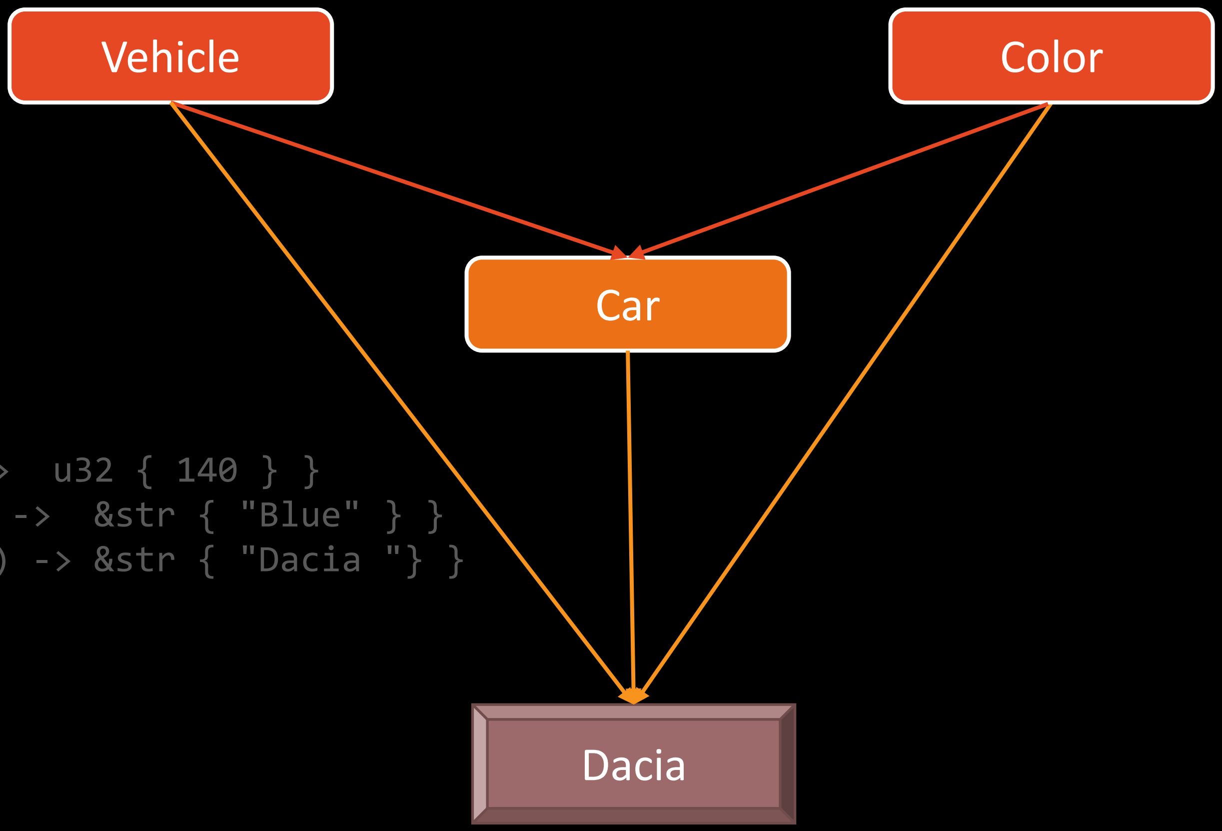
```
Name  = Dacia
Speed = 140
Color = Blue
```

Super traits

The same example → but with trait combos.

Rust

```
trait Vehicle {
    fn get_name(&self) -> &str;
}
trait Color {
    fn get_color(&self) -> &str;
}
trait Car: Vehicle + Color {
    fn get_speed(&self) -> u32;
}
struct Dacia {}
impl Car for Dacia { fn get_speed(&self) ->  u32 { 140 } }
impl Color for Dacia { fn get_color(&self) ->  &str { "Blue" } }
impl Vehicle for Dacia { fn get_name(&self) -> &str { "Dacia "} }
fn main() {
    let d = Dacia {};
    println!("Name  = {}",d.get_name());
    println!("Speed = {}",d.get_speed());
    println!("Color = {}",d.get_color());
}
```



Special Traits

Special Traits

Rust has some special traits that can be used to improve certain operations or how some types behave:

- Traits that reflect certain properties (Copy, Clone, Debug, etc)
- Traits that reflects operators (addition, substraction, etc)
- Traits that reflects comparations between types
- Traits that reflect casts and/or conversions between types

These traits can be overridden. In some cases, Rust can automatically implement some special traits via #[derive(…)] attribute.

Special Traits

To automatically tell Rust that it needs to implement a trait for a specific class, use #[derive(…)] attribute. The general format is:

#[***derive***(Trait$_1$, Trait$_2$, … Trait$_n$)]

List of these traits (that are also called derivable traits):

Trait	Usage
Copy	Support for Copy Semantics
Clone	Add support to clone an object
Debug	Debug information for an object
Hash	Provide a way to compute a hash for a reference (Compiler controlled)
Default	Default value for an object
Eq	Comparation support (equal)
PartialEq	Comparation support (equal and not equal)
Ord	Set an object to be comparable (can be ordered)
PartialOrd	Set an object to be partial comparable (can be ordered)

Special Traits (Copy & Clone)

Copy trait indicates "**Copy semantics**" for a specify trait. Clone is a super trait for Copy trait (so any implementation of Copy trait implies Clone traits as well).

*Rust (**Copy** trait definition)*

```
pub trait Copy: Clone {
    // Empty.
}
```

*Rust (**Clone** trait definition)*

```
pub trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {…}
}
```

Notice that Copy trait has no defined method. This is because this trait implies byte-wise copy for any object upon assignment. Clone imply Sized (a trait that indicates that the size of the object that has this trait, must be known at compile time). This is to be expected if Copy implies a byte-wise copy (a **memcpy**).

Special Traits (Copy & Clone)

Clone trait, however, can be implemented

Rust

```
struct MyNumber {
    value: i32,
}
impl Copy for MyNumber {}
impl Clone for MyNumber {
    fn clone(&self) -> Self {
        MyNumber {
            value: self.value + 1,
        }
    }
}
fn main() {
    let x = MyNumber { value: 1 };
    let y = x.clone();
    let z = x;
    println!("{},{},{}", x.value, y.value, z.value);
}
```

Output
1,2,1

Notice that y.value is 2 (this is to be expected as **x.clone()** increases the value of MyNumber.

Special Traits (Copy & Clone)

Clone trait, however, can be implemented

Rust

```
#[derive(Copy,Clone)]
struct MyNumber {
    value: i32,
}
fn main() {
    let x = MyNumber { value: 1 };
    let y = x.clone();
    let z = x;
    println!("{},{},{}", x.value, y.value, z.value);
}
```

Output
1,1,1

OBS: *Notice that the default implementation (obtained via #[derive(Copy,Clone)] uses byte wise copy for both clone and assignment.*

Special Traits (Copy & Clone)

A newly create struct can implement Copy trait only if all of its fields implement Copy trait.

Rust

```rust
#[derive(Copy,Clone)]
struct MyNumber {
    value: i32,
    name:String
}
fn main() {
    let x = MyNumber { value: 1, name: "123".to_string() };
}
```

Error

```
error[E0204]: the trait `Copy` may not be implemented for this type
 --> src\main.rs:1:10
  |
1 | #[derive(Copy,Clone)]
  |          ^^^^
...
4 |     name:String
  |     ----------- this field does not implement `Copy`
  |
```

In this case, one of the fields (name) does not implement Copy trait and as such the entire structure can not implemented it.

Special Traits (Display & Debug)

Rust has two traits (Display and Debug) that should be used to display an object. Both Debug and Display traits have the same methods, however there are some differences between them:

- Debug trait can be used with #[derive(…)], Display can't
- Display is designed for user-facing, while Debug is merely a developer way of validating information about an object.
- Debug requires a special format {:?}

*Rust (**Display** trait definition)*

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

*Rust (**Debug** trait definition)*

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

Special Traits (Display & Debug)

Let's see some examples:

Rust

```rust
#[derive(Debug)]
struct MyNumber {
    value: i32,
}
fn main() {
    let x = MyNumber { value: 1 };
    println!("{:?}",x);
}
```

Output
MyNumber { value: 1 }

Notice that it is fairly easy to print any kind of object if we implement (via #[derive(Debug)]) the Debug trait for it. Rust will create a default implementation for this trait that will print each field from that structure.

Special Traits (Display & Debug)

Let’s see some examples:

Rust

```
use std::fmt::Display;
use std::fmt;
struct MyNumber {
    value: i32,
}
impl Display for MyNumber {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> std::fmt::Result {
        f.write_str("MyNumber => with value = ")?;
        f.write_fmt(format_args!("{}",self.value))?;
        Ok(())
    }
}
fn main() {
    let x = MyNumber { value: 1 };
    println!("{}",x);
}
```

Output

MyNumber => with value = 1

Special Traits (Display & Debug)

Let's see some examples:

Rust

```rust
use std::fmt::Display;
use std::fmt;
struct MyNumber {
    value: i32,
}
impl Display for MyNumber {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> std::fmt::Result {
        f.write_str("MyNumber => with value = ")?;
        f.write_fmt(format_args!("{}",self.value))?;
        Ok(())
    }
}
fn main() {
    let x = MyNumber { value: 1 };
    println!("{}",x);
}
```

Output

MyNumber => with value = 1

Alternatively, the **write!** macro can be used !

Rust

```rust
fn fmt(&self, f: &mut fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "MyNumber => with value = {}",self.value)?;
    Ok(())
}
```

Special Traits (Default)

Default trait is used to describe a default initialization value for an object. It works like a static (constructor) method that creates an object. All basic types implement that trait. Furthermore, Default trait can be defined via #[derive(…)].

*Rust (**Default** trait definition)*

```
pub trait Default: Sized {
{
    fn default() -> Self;
}
```

Besides basic types, more than 150 types in Rust implement default.

Usage:

```
let x = Type::default();
let x: Type = Default::default()
```

Special Traits (Default)

Let's see some examples:

Rust

```
struct MyNumber {
    value: i32,
}
impl Default for MyNumber {
    fn default() -> Self {
        Self { value: 100 }
    }
}
fn main() {
    let x = MyNumber::default();
    let y = i32::default();      // 0 value
    let z = String::default();   // Empty string
    println!("{}",x.value);
    println!("{}",y);
    println!("[{}]",z);
}
```

Output

```
100
0
[]
```

Special Traits (Default)

Default trait can be automatically implemented via #[derive(...)] attribute. All of the structure members **MUST** implement Default trait as well.

Rust

```
#[derive(Default,Debug)]
struct MyNumber {
    value: i32,
    float: f32,
    flag: bool
}
fn main() {
    let x = MyNumber::default();
    let y: MyNumber = Default::default();
    println!("{:?}",x);
    println!("{:?}",y);
}
```

Output

```
MyNumber { value: 0, float: 0.0, flag: false }
MyNumber { value: 0, float: 0.0, flag: false }
```

Special Traits (Default)

Default trait can be automatically implemented via #[derive(...)] attribute. All of the structure members **MUST** implement Default trait as well.

Rust

```rust
struct MyStructWithoutDefault {
    value: i32
}

#[derive(Default,Debug)]
struct MyNumber {
    value: i32,
    float: f32,
    flag: bool,
    extra: MyStructWithoutDefault
}

fn main() {
    let x = MyNumber::default();
    println!("{:?}",x);
}
```

Error

```
error[E0277]: the trait bound `MyStructWithoutDefault: Default` is not satisfied
  --> src\main.rs:10:5
   |
5  | #[derive(Default,Debug)]
   |          ------- in this derive macro expansion
...
10 |     extra: MyStructWithoutDefault
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Default` is not implemented for
   |                                   `MyStructWithoutDefault`
```

Special Traits (Default)

When #[derive(…)] attribute is used to automatically implement the Default trait for an ***enum***, you MUST also specify the default variant (to do this add #[default] before the e default variant in the ***enum***).

Rust

```
use std::default;

#[derive(Debug, Default)]
enum Color {
    Red,
    #[default]
    Green,
    Blue,
    White
}
fn main() {
    let x = Color::default();
    println!("{:?}",x);
}
```

Output
Green

Special Traits (Default)

You can also overwrite some default value and keep the rest of them by using the following syntax `..Default::default()` when constructing an object (this is in fact another usage of functional update syntax in Rust):

Rust

```rust
#[derive(Debug, Default)]
struct MyStruct {
    x: i32,
    y: bool,
    z: f32,
    name: String
}
fn main() {
    let x = MyStruct::default();
    let y = MyStruct { x: 10, ..Default::default()};
    let z = MyStruct { name: "10".to_string(), y:true, ..Default::default()};
    println!("x = {:?}",x);
    println!("y = {:?}",y);
    println!("z = {:?}",z);
}
```

Output

```
x = MyStruct { x: 0, y: false, z: 0.0, name: "" }
y = MyStruct { x: 10, y: false, z: 0.0, name: "" }
z = MyStruct { x: 0, y: true, z: 0.0, name: "10" }
```

Special Traits (Eq and PartialEq)

Eq and PartialEq traits are used to describe if how to check the equality or difference between two object. PartialEq is the super trait of Eq.

*Rust (**PartialEq** trait definition)*

```
pub trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;

    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}
```

*Rust (**Eq** trait definition)*

```
pub trait Eq: PartialEq<Self> {
}
```

Notice that "ne" (not-equal) method has a default implementation. This mean that normally, a type that implements this trait only needs to overwrite the eq method.

The "ne" is useful for types (e.g. floating values) that have special cases (such as NaN) where different values (in term of bit comparation) might have the same interpretation.

Special Traits (Eq and PartialEq)

Let's see a simple example on how to use PartialEq:

Rust

```rust
struct MyStruct {
    value: i32
}
impl PartialEq for MyStruct {
    fn eq(&self, other: &Self) -> bool {
        self.value == other.value
    }
}
fn main() {
    let x = MyStruct{value: 10};
    let y = MyStruct{value: 10};
    if x == y {
        println!("x an y are equals !");
    }
}
```

Output

```
x an y are equals !
```

Special Traits (Eq and PartialEq)

PartialEq and Eq traits can be automatically implemented via #[derive(...)] attribute. Keep in mind that ***PartialEq*** is a super trait of ***Eq*** and as such if you derive from Eq you must derive from ***PartialEq*** as well. All of the members from that structure MUST implement PartialEq and/or Eq.

Rust

```
#[derive(PartialEq)]
struct MyStruct {
    value: i32
}
fn main() {
    let x = MyStruct{value: 10};
    let y = MyStruct{value: 10};
    if x == y {
        println!("x an y are equals !");
    }
}
```

Output

```
x an y are equals !
```

Special Traits (Eq and PartialEq)

PartialEq and Eq traits can be automatically implemented via #[derive(...)] attribute. Keep in mind that ***PartialEq*** is a super trait of ***Eq*** and as such if you derive from Eq you must derive from ***PartialEq*** as well. All of the members from that structure MUST implement PartialEq and/Oor Eq.

Rust

```rust
struct MyNonComparableStruct {
    field:i32
}
#[derive(Eq,PartialEq)]
struct MyStruct {
    value: i32,
    extra: MyNonComparableStruct
}
fn main() {
    let x = MyStruct{value: 10, extra: MyNonComparableStruct { field: 10 }};
    let y = MyStruct{value: 10, extra: MyNonComparableStruct { field: 10 }};
    if x == y {
        println!("x an y are equals !");
    }
}
```

Error

```
error[E0369]: binary operation `==` cannot be applied to type `MyNonComparableStruct`
 --> src\main.rs:7:5
  |
4 | #[derive(PartialEq)]
  |          --------- in this derive macro expansion
...
7 |     extra: MyNonComparableStruct
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
note: an implementation of `PartialEq<_>` might be missing for `MyNonComparableStruct`
```

Special Traits (Ord and PartialOrd)

Ord and PartialOrd traits describe a way to compare two objects. PartialOrd is a super trait of Ord, and PartialEq is a super trait of PartialOrd

*Rust (**PartialOrd** trait definition)*

```rust
pub trait PartialOrd<Rhs: ?Sized = Self>: PartialEq<Rhs>
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool {
        matches!(self.partial_cmp(other), Some(Less))
    }
    fn le(&self, other: &Rhs) -> bool {
        !matches!(self.partial_cmp(other), None | Some(Greater))
    }
    fn gt(&self, other: &Rhs) -> bool {
        matches!(self.partial_cmp(other), Some(Greater))
    }
    fn ge(&self, other: &Rhs) -> bool {
        matches!(self.partial_cmp(other), Some(Greater | Equal))
    }
}
```

*Rust (**Ordering**)*

```rust
pub enum Ordering {
    Less = -1,
    Equal = 0,
    Greater = 1,
}
```

Notice that the only method that needs to be implemented is **partial_cmp !**

By default, ***PartialOrd*** implements:

- lt → lower then
- le → lower or equal
- gt → greater then
- ge → greater or equal

Special Traits (Ord and PartialOrd)

As PartialEq is a super trait of PartialOrd, "**eq**" and "**ne**" methods are inherited from PartialEq. **Ord** trait also implements method like ***min***, ***max*** and ***clamp***.

*Rust (**Ord** trait definition)*

```
pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
    fn max(self, other: Self) -> Self where Self: Sized,
    {
        max_by(self, other, Ord::cmp)
    }
    fn min(self, other: Self) -> Self where Self: Sized,
    {
        min_by(self, other, Ord::cmp)
    }
    fn clamp(self, min: Self, max: Self) -> Self where Self: Sized,
    {
        assert!(min <= max);
        if self < min { min }
        else if self > max { max }
        else { self }
    }
}
```

Special Traits (Ord and PartialOrd)

Let’s see an example to understand how **max**, **min** and **clamp** methods work.

Rust

```
fn main() {
    println!("5.max(10)    = {}",5.max(10));
    println!("5.max(2)     = {}",5.max(2));
    println!("5.min(10)    = {}",5.min(10));
    println!("5.min(2)     = {}",5.min(2));
    println!("5.clamp(2,8) = {}",5.clamp(2,8));
    println!("5.clamp(7,9) = {}",5.clamp(7,9));
    println!("5.clamp(1,4) = {}",5.clamp(1,4));
}
```

Output

```
5.max(10)    = 10
5.max(2)     = 5
5.min(10)    = 5
5.min(2)     = 2
5.clamp(2,8) = 5
5.clamp(7,9) = 7
5.clamp(1,4) = 4
```

.clamp(…) method keeps a value within an interval. If it is lower than its lower bound, the value returned will the lower bound of the interval. If it is greater than the upper bound, the value return will be the upper bound of the interval. Otherwise, the value will remained unchanged.

Special Traits (Ord and PartialOrd)

Let's see a simple example that illustrates how to manually implement PartialOrd.

Rust

```
use std::cmp::Ordering;
struct MyStruct { value: i32 }
impl PartialEq for MyStruct {
    fn eq(&self, other: &Self) -> bool { self.value == other.value }
}
impl PartialOrd for MyStruct {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        if self.value>other.value { return Some(Ordering::Greater); }
        if self.value<other.value { return Some(Ordering::Less); }
        return Some(Ordering::Equal);
    }
}
fn main() {
    let x = MyStruct{value:10};
    let y = MyStruct{value:20};
    if y>x {
        println!("y is bigger than x");
    }
}
```

Output

```
y is bigger than x
```

Special Traits (Ord and PartialOrd)

When #[derive(...)] attribute is used to automatically implement the PartialOrd, keep in mind that the automatic logic is to compare each variable **<u>in the order</u>** they were added in the structure.

Rust

```
#[derive(PartialEq, PartialOrd)]
struct MyStruct {
    v1: i32,
    v2: i32,
    v3: i32
}
fn main() {
    let x = MyStruct { v1: 10, v2:20, v3:10 };
    let y = MyStruct { v1: 20, v2:10, v3:100 };
    let z = MyStruct { v1: 10, v2:20, v3:100 };
    let t = MyStruct { v1: 10, v2:10, v3:100 };
    println!("CMP(x,y) = {:?}",x.partial_cmp(&y));
    println!("CMP(x,z) = {:?}",x.partial_cmp(&z));
    println!("CMP(x,t) = {:?}",x.partial_cmp(&t));
    println!("CMP(x,x) = {:?}",x.partial_cmp(&x));
}
```

Output

```
CMP(x,y) = Some(Less)
CMP(x,z) = Some(Less)
CMP(x,t) = Some(Greater)
CMP(x,x) = Some(Equal)
```

Special Traits (Drop)

Rust does not have a destructor (in a traditional, descriptive, manually defined C++ way). However, there is a trait called Drop that serves a similar purpose (it contains a method that is being called when the scope of an object ends).

*Rust (**Drop** trait definition)*

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

While in most cases, you don't really need to implement this trait (as Rust will automatically destroy object), there are some scenarios (e.g. when managing an external resource, a socket, etc) when this trait might be required.

Drop can not be automatically implemented via #[derive(…)] attribute.

OBS: *Keep in mind that Rust will not allow you to call .drop() explicitly.*

Special Traits (Drop)

Let's see an example:

Rust

```
struct MyStruct {
    v: i32
}
impl Drop for MyStruct {
    fn drop(&mut self) {
        println!("Dropping (v={})",self.v);
    }
}
fn main() {
    let x = MyStruct{v:10};
    {
        let y = MyStruct{v:20};
        println!("Inner block scope will end right now !");
    }
    println!("Main block scope will end right now !");
}
```

Output

```
Inner block scope will end right now !
Dropping (v=20)
Main block scope will end right now !
Dropping (v=10)
```

y.drop()

x.drop()

Special Traits (Drop)

As previously stated, explicit destructor calls (via .drop()) method are not allowed.

Rust

```rust
struct MyStruct {
    v: i32
}
impl Drop for MyStruct {
    fn drop(&mut self) {
        println!("Dropping (v={})",self.v);
    }
}
fn main() {
    let x = MyStruct{v:10};
    x.drop();
}
```

Error

```
error[E0040]: explicit use of destructor method
  --> src\main.rs:13:7
   |
13 |     x.drop();
   |     --^^^^--
   |     | |
   |     | explicit destructor calls not allowed
   |     help: consider using `drop` function: `drop(x)`
```

OBS: *If allowed, these calls could lead to the wrong behavior of some objects (e.g. if the destructor closes some handles) if the object is being used after the call to .drop().*

Special Traits (Drop)

The order .drop() method is called is also different how C++ is doing. First it is called for the main object, then for every field from that object in the order of the declaration.

Rust

```rust
struct ClassA { v: i32 }
struct ClassB { v: i32 }
struct MyStruct { a: ClassA, b: ClassB }
impl Drop for MyStruct {
    fn drop(&mut self) { println!("Dropping MyStruct"); }
}
impl Drop for ClassA {
    fn drop(&mut self) { println!("Dropping ClassA"); }
}
impl Drop for ClassB {
    fn drop(&mut self) { println!("Dropping ClassB"); }
}
fn main() {
    let x = MyStruct { a: ClassA { v: 0 }, b: ClassB { v: 0 } };
}
```

Output

```
Dropping MyStruct
Dropping ClassA
Dropping ClassB
```

Special Traits (Drop)

Another observation is that Drop trait can not be implemented for object that have Copy semantics. This is because object that implement Copy trait are normally copied (via a ***memcpy*** method) and as such memory deallocation can be handled automatically.

Rust

```rust
#[derive(Copy,Clone)]
struct MyStruct {
    v: i32,
}
impl Drop for MyStruct {
    fn drop(&mut self) {
        println!("Dropping MyStruct");
    }
}

fn main() {
    let x = MyStruct { v: 0 };
}
```

Error

```
error[E0184]: the trait `Copy` may not be implemented for this type; the type has a destructor
 --> src\main.rs:1:10
  |
1 | #[derive(Copy,Clone)]
  |          ^^^^ Copy not allowed on types with destructors
```

Special Traits (Drop)

Another observation is that implementing Drop trait for a struct will disable the partial move ability. Let's analyze the following example:

Rust

```rust
struct Test {
    x: i32,
    name: String
}
fn main() {
    let t = Test{x:1, name: String::from("ABC") };
    let _s = t.name;
    println!("x={}",t.x);
}
```

Output

```
x=1
```

Notice that `let _s = t.name;` moves the value of field ***name*** from structure Test. But this is a partial move as the structure Test (through its member "x") is still available (we can actually print t.x).

Special Traits (Drop)

Now let's implement Drop trait for the same structure. We will notice that the same example does not work anymore (meaning that you can not move individual fields from a structure anymore – as the new Drop implementation implies the entire structure is being moved).

Rust

```rust
struct Test {
    x: i32,
    name: String
}
impl Drop for Test {
    fn drop(&mut self) { }
}
fn main() {
    let t = Test{x:1, name: String::from("ABC") };
    let _s = t.name;
    println!("x={}",t.x);
}
```

Error

```
error[E0509]: cannot move out of type `Test`, which implements the `Drop` trait
  --> src\main.rs:10:14
   |
10 |     let _s = t.name;
   |              ^^^^^^
   |              |
   |              cannot move out of here
   |              move occurs because `t.name` has type `String`, which does
   |              not implement the `Copy` trait
```

Special Traits (Sized)

Sized trait is a special trait that indicates that current type has a know size at compile time.

*Rust (**Sized** trait definition)*

```
pub trait Sized {
    // Empty.
}
```

This purpose is controlled by the compiler. You cand not implicitly implement it but it is very useful for bounds (in generics) where this trait might be required. It is also possible to relax the bounds that request a Sized object by adding '?' in front of it (?Sized). This removes the bound for an object to be Sized.

Special Traits (Sized)

Explicit implementation of Sized trait is not allowed:

Rust

```rust
struct MyStruct {
    v: i32,
}
impl Sized for MyStruct {

}

fn main() {
    let x = MyStruct { v: 0 };
}
```

Error

```
error[E0322]: explicit impls for the `Sized` trait are not permitted
 --> src\main.rs:4:1
  |
4 | impl Sized for MyStruct {
  | ^^^^^^^^^^^^^^^^^^^^^^^ impl of 'Sized' not allowed
```

Special Traits (Sized)

Notice that even if Sized can be a super trait for another trait, that trait can not be used to instantiate a dynamic object.

Rust

```rust
trait A: Sized {
}
struct S {
}
impl A for S {
}
fn main() {
    let y: Box<dyn A> = Box::new(S{});
}
```

Error

```
error[E0038]: the trait `A` cannot be made into an object
 --> src\main.rs:5:12
  |
5 |     let y: Box<dyn A> = Box::new(S {});
  |            ^^^^^^^^^^ `A` cannot be made into an object
  |
note: for a trait to be "object safe" it needs to allow building a vtable to allow the
call to be resolvable dynamically;
 --> src\main.rs:1:10
  |
1 | trait A: Sized {}
  |       -  ^^^^^ ...because it requires `Self: Sized`
  |       |
  |       this trait cannot be made into an object...
```

Special Traits (Deref and DerefMut)

Deref and DerefMut traits are used to explicit dereferencing operations (an equivalent to operator*/operator-> from C++). This mechanism is called **Deref coercion**.

*Rust (**Deref** trait definition)*

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

*Rust (**DerefMut** trait definition)*

```
pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

If a type A implements Deref (with Target type set to type B) then:

- &A can be coerced to &B
- A implicitly implements all methods from B

OBS: ***Deref*** *and* ***DerefMut*** *simulate the concept of inheritance (in the sense that methods and data member from another type (e.g. parent class) are accessible via the child object.*

Special Traits (Deref and DerefMut)

Let's see an example:

Rust

```
use std::ops::{Deref,DerefMut};
struct B { x: i32, y: i32 }
struct A { b: B, a: i32 }

impl A { fn new() -> A { A { b: B { x: 0, y: 0 }, a: 0 } } }
impl Deref for A {
    type Target = B;
    fn deref(&self) -> &Self::Target { &self.b }
}
impl DerefMut for A {
    fn deref_mut(&mut self) -> &mut Self::Target { &mut self.b }
}
fn increment_y(b: &mut B) { b.y += 1; }
fn main() {
    let mut a = A::new();
    a.x = 10;
    increment_y(&mut a);
    println!("From B: x={}, y={}, From A: a={}", a.x, a.y, a.a);
}
```

Output

From B: x=10, y=1, From A: a=0

Special Traits (Deref and DerefMut)

Let’s see an example:

Rust

```
use std::ops::{Deref,DerefMut};
struct B { x: i32, y: i32 }
struct A { b: B, a: i32 }

impl A { fn new() -> A { A { b: B { x: 0, y: 0 }, a: 0 } } }
impl Deref for A {
    type Target = B;
    fn deref(&self) -> &Self::Target { &self.b }
}
impl DerefMut for A {
    fn deref_mut(&mut self) -> &mut Self::Target { &mut self.b }
}
fn increment_y(b: &mut B) { b.y += 1; }
fn main() {
    let mut a = A::new();
    a.x = 10;
    increment_y(&mut a);
    println!("From B: x={}, y={}, From A: a={}", a.x, a.y, a.a);
}
```

Output

From B: x=10, y=1, From A: a=0

“A” type does not have any **.x** or **.y** fields. However, due to the ***Deref*** implementation, you can automatically access fields **.x** and **.y** from field **b** of type A

Special Traits (Deref and DerefMut)

Let's see an example:

Rust

```
use std::ops::{Deref,DerefMut};
struct B { x: i32, y: i32 }
struct A { b: B, a: i32 }

impl A { fn new() -> A { A { b: B { x: 0, y: 0 }, a: 0 } } }
impl Deref for A {
    type Target = B;
    fn deref(&self) -> &Self::Target { &self.b }
}
impl DerefMut for A {
    fn deref_mut(&mut self) -> &mut Self::Target { &mut self.b }
}
fn increment_y(b: &mut B) { b.y += 1; }
fn main() {
    let mut a = A::new();
    a.x = 10;
    increment_y(&mut a);
    println!("From B: x={}, y={}, From A: a={}", a.x, a.y, a.a);
}
```

Output

From B: x=10, y=1, From A: a=0

Due to the ***DerefMut*** implementation, you can automatically obtain a mutable reference to field **.x** field **b** of type A

Special Traits (Deref and DerefMut)

Let's see an example:

Rust

```
use std::ops::{Deref,DerefMut};
struct B { x: i32, y: i32 }
struct A { b: B, a: i32 }

impl A { fn new() -> A { A { b: B { x: 0, y: 0 }, a: 0 } } }
impl Deref for A {
    type Target = B;
    fn deref(&self) -> &Self::Target { &self.b }
}
impl DerefMut for A {
    fn deref_mut(&mut self) -> &mut Self::Target { &mut self.b }
}
fn increment_y(b: &mut B) { b.y += 1; }
fn main() {
    let mut a = A::new();
    a.x = 10;
    increment_y(&mut a);
    println!("From B: x={}, y={}, From A: a={}", a.x, a.y, a.a);
}
```

Output

From B: x=10, y=1, From A: a=0

Notice that **increment_y** expects a mutable reference to an object of type B. However, it can be called with a mutable reference of type A that can be **coerced** due to ***DerefMut*** to a mutable reference of type B.

Special Traits (From and Into)

From and Into traits are used to perform value-to-value conversion.

*Rust (**From** trait definition)*

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

*Rust (**Into** trait definition)*

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

It is recommended to avoid implemented Into but rather implement From. Implementing From will trigger the creation of Into as well due to the blanket implementation in the standard library.

*Rust (blanket implementation for **Into**)*

```
impl<T, U> Into<U> for T
where
    U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

Special Traits (From and Into)

Let’s see an example:

Rust

```rust
struct Test {
    value: i32,
}
impl From<i32> for Test {
    fn from(v: i32) -> Test {
        Test { value: v }
    }
}

fn main() {
    let a = Test::from(10);
    println!("a.value = {}", a.value);
    let b: Test = 11.into();
    println!("b.value = {}", b.value);
}
```

Output

```
a.value = 10
b.value = 11
```

Special Traits (From and Into)

Let’s see an example:

Rust

```
struct Test { value: i32 }
impl From<Test> for i32 {
    fn from(t: Test) -> i32 {
        t.value
    }
}
impl From<&Test> for i32 {
    fn from(t: &Test) -> i32 {
        t.value
    }
}
fn main() {
    let a = Test { value: 10 };
    let x: i32 = (&a).into();
    let b = Test { value: 20 };
    let y: i32 = b.into();
    println!("{x},{y}");
}
```

Output

```
10
20
```

When implementing **From** (if possible) consider implementing both for an object (with ownership transfer) and for a reference

Special Traits (From and Into)

From and Into traits also have a try version (**TryFrom** and **TryInto**).

*Rust (**From** trait definition)*

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}
```

*Rust (**Into** trait definition)*

```
pub trait TryInto<T>: Sized {
    type Error;

    fn try_into(self) -> Result<T, Self::Error>;
}
```

The difference from the From and Into forms is that these traits return a **Result** (allowing someone to validate if something can be converted into another object or not).

Special Traits (AsRef and AsMut)

AsRef and AsMut traits are used to perform cheap reference-to-reference conversion. Keep in mind that similar result can be obtained if using From or Into traits (but implemented over/for a reference or mutable reference).

*Rust (**AsRef** trait definition)*

```rust
pub trait AsRef<T: ?Sized> {
    fn as_ref(&self) -> &T;
}
```

*Rust (**AsMut** trait definition)*

```rust
pub trait AsMut<T: ?Sized>
   fn as_mut(&mut self) -> &mut T;
}
```

Rust also has two very similar traits (Borrow and BorrowMut) that resembles in terms of definition with AsRef and AsMut.

*Rust (**Borrow** trait definition)*

```rust
pub trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}
```

*Rust (**BorrowMut** trait definition)*

```rust
pub trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
```

Special Traits (AsRef and AsMut)

Let’s see an example:

Rust

```
#[derive(Debug)]
struct Test { x: i32 }
impl AsRef<i32> for Test {
    fn as_ref(&self) -> &i32 {
        return &self.x;
    }
}
impl AsMut<i32> for Test {
    fn as_mut(&mut self) -> &mut i32 {
        return &mut self.x;
    }
}
fn main() {
    let mut a = Test{x:10};
    let a_mut: &mut i32 = a.as_mut();
    *a_mut = 20;
    let a_ref: &i32 = a.as_ref();
    println!("{:?},{}", a, a_ref);
}
```

Output

Test { x: 20 },20

Special Traits (AsRef and AsMut)

Let's see an example (this time using borrow/borrow_mut):

Rust

```rust
use std::borrow::{Borrow,BorrowMut};

#[derive(Debug)]
struct Test {
    x: i32
}
impl Borrow<i32> for Test {
    fn borrow(&self) -> &i32 { return &self.x; }
}
impl BorrowMut<i32> for Test {
    fn borrow_mut(&mut self) -> &mut i32 { return &mut self.x; }
}
fn main() {
    let mut a = Test{x:10};
    let a_mut: &mut i32 = a.borrow_mut();
    *a_mut = 20;
    let a_ref: &i32 = a.borrow();
    println!("{:?},{}",a,a_ref);
}
```

Output

```
Test { x: 20 },20
```

Special Traits (AsRef and AsMut)

The main difference between AsRef/AsMut and Borrow/BorrowMut is that Borrow and BorrowMut have *several blanket implementations* that allows one to used them directly in a generic (e.g. in a where clause) without the need to actually implement them for a specific type.

Let's consider the following problem → we want to write a generic function that consumes an object but before it consumes it, it uses its reference to print it.

Let's see how we can implement such a function using both Borrow/BorrowMut and AsRef/AsMut.

Special Traits (AsRef and AsMut)

Solution (using borrow/borrow_mut):

Rust

```rust
use core::fmt;
use std::{borrow::Borrow, fmt::Display};
struct Point { x: i32, y: i32}
impl Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
         write!(f, "Point object => (x={}, y={})", self.x, self.y)
    }
}
fn print_value<T>(object: T)
where
    T: Borrow<T> + Display,
{
    let x = object.borrow();
    println!("obj = {}", x);
}
fn main() {
    let p = Point { x: 10, y: 20 }; print_value(p);
    let x = 10; print_value(x);
}
```

Output

```
obj = Point object => (x=10, y=20)
obj = 10
```

Special Traits (AsRef and AsMut)

Solution (using borrow/borrow_mut):

Rust

```
use core::fmt;
use std::{borrow::Borrow, fmt::Display};
struct Point { x: i32, y: i32}
impl Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Point object => (x={}, y={})", self.x, self.y)
    }
}
fn print_value<T>(obj
where
    T: Borrow<T> +
{
    let x = object.bo
    println!("obj = {}", x);
}
fn main() {
    let p = Point { x: 10, y: 20 }; print_value(p);
    let x = 10; print_value(x);
}
```

Output

```
obj = Point object => (x=10, y=20)
obj = 10
```

Notice that we require Borrow to be implemented for T but we haven't actually implemented it (this is because blanket implementation does it for us).

Special Traits (AsRef and AsMut)

Let's try the same code with AsRef:

Rust

```
use core::fmt;
use std::{borrow::Borrow, fmt::Display};
struct Point { x: i32, y: i32}
impl Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Point object => (x={}, y={})", self.x, self.y)
    }
}
fn print_value<T>(object: T)
where
    T: AsRef<T> + Display
{
    let x = object.as_ref();
    println!("obj = {}", x);
}
fn main() {
    let p = Point { x: 10, y: 20 }; print_value(p);
    let x = 10; print_value(x);
}
```

Notice that without the blanket implementation, we can not use AsRef/AsMut in a generic !

Error

```
error[E0277]: the trait bound `Point: AsRef<Point>` is not satisfied
  --> src\main.rs:22:17
   |
22 |     print_value(p);
   |     ----------- ^ the trait `AsRef<Point>` is not implemented for `Point`
   |     |
   |     required by a bound introduced by this call
   |
```

Operators

Operators

When creating different types, it is often required to overwrite how some mathematical operations work for them. In C++ this is accomplished by using the keyword “operator” and being able to write specific methods that describe how certain operation should behave.

In Rust, there are a set of traits that if implemented will result in a similar behavior. Keep in mind that there has to be a resemblance on how an operator should behave. Some operators like (&& and ||) use lazy evaluation and require bool parameters and as such can not be overwritten.

Operators

Most of the arithmetic (binary) operators have two possible forms:

A) Expr ⊕ Expr (binary operation)

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

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

*Notice that the method receives a self. This means that **ownership will be transferred** if Copy trait is not implemented !*

B) Variable ⊕= Expr (assignment)

```
pub trait OperationNameAssign<Rhs = Self> {

    fn operationname_assign(&mut self, rhs: Rhs);
}
```

With OperationName ***(sentence case)*** being the name assigned for the operation ⊕ and operationname ***(lowercased)*** the name of the method that needs to be implemented to overwrite that operation.

Operators

The next table contains a list of all binary operations that follow the previous described template:

Operator	Trait	Method
+	std::ops::**Add**	add
-	std::ops::**Sub**	sub
*	std::ops::**Mul**	mul
/	std::ops::**Div**	div
%	std::ops::**Rem**	rem
&	std::ops::**BitAnd**	bitand
|	std::ops::**BitOr**	bitor
^	std::ops::**BitXor**	bitxor
<<	std::ops::**Shl**	shl
>>	std::ops::**Shr**	shr

Operator	Trait	Method
+=	std::ops::**AddAssign**	add_assign
-=	std::ops::**SubAssign**	sub_assign
*=	std::ops::**MulAssign**	mul_assign
/=	std::ops::**DivAssign**	div_assign
%=	std::ops::**RemAssign**	rem_assign
&=	std::ops::**BitAndAssign**	bitand_assign
|=	std::ops::**BitOrAssign**	bitor_assign
^=	std::ops::**BitXorAssign**	bitxor_assign
<<=	std::ops::**ShlAssign**	shl_assign
>>=	std::ops::**ShrAssign**	shr_assign

Operators

Let's see a very simple example:

Rust

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

struct Test {
    value: i32,
}
impl Add<i32> for Test {
    type Output = i32;

    fn add(self, rhs: i32) -> Self::Output {
        self.value + rhs
    }
}
fn main() {
    let a = Test { value: 10 };
    let x = a + 10;
    println!("{x}");
}
```

Output

```
20
```

Operators

Notice that add method receives a **self** (meaning that the ownership of the object is transferred and as such, "a" will no longer be available after the addition.

Rust

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

struct Test { value: i32 }
impl Add<i32> for Test {
    type Output = i32;
    fn add(self, rhs: i32) -> Self::Output {
        self.value + rhs
    }
}
fn main() {
    let a = Test { value: 10 };
    let x = a + 10;
    println!("{x}");
    println!("{}",a.value);
}
```

Error

```
error[E0382]: borrow of moved value: `a`
  --> src\main.rs:17:19
   |
14 |     let a = Test { value: 10 };
   |         - move occurs because `a` has type `Test`, which
   |           does not implement the `Copy` trait
15 |     let x = a + 10;
   |             ------ `a` moved due to usage in operator
16 |     println!("{x}");
17 |     println!("{}",a.value);
   |                   ^^^^^^^ value borrowed here after move
```

Operators

You can, however, implement Add for a reference (in this case for &Test) and avoid transferring ownership.

Rust

```
use std::ops::Add;
struct Test {
    value: i32,
}
impl Add<i32> for &Test {
    type Output = i32;
    fn add(self, rhs: i32) -> Self::Output {
        self.value + rhs
    }
}
fn main() {
    let a = Test { value: 10 };
    let x = (&a) + 10;
    println!("{x}");
    println!("{}", a.value);
}
```

Output

```
20
10
```

Operators

You can, however, implement Add for a reference (in this case for &Test) and avoid transferring ownership.

Rust

```
use std::ops::Add;
struct Test {
    value: i32,
}
impl Add<i32> for &Test {
    type Output = i32;
    fn add(self, rhs: i32) -> Self::Output {
        self.value + rhs
    }
}
fn main() {
    let a = Test { value: 10 };
    let x = (&a) + 10;
    println!("{x}");
    println!("{}", a.value);
}
```

Output

```
20
10
```

Notice that the syntax is not the clear (you need to explicitly say that you want to add a reference (&a) with a number.

Operators

Notice that if Add is not implemented for self, adding an object with a number (for our case) will fail.

Rust

```
use std::ops::Add;
struct Test {
    value: i32,
}
impl Add<i32> for &Test {
    type Output = i32;
    fn add(self, rhs: i32) -> Self::Output {
        self.value + rhs
    }
}
fn main() {
    let a = Test { value: 10 };
    let x = a + 10;
    println!("{x}");
    println!("{}", a.value);
}
```

Error

```
error[E0369]: cannot add `{integer}` to `Test`
   --> src\main.rs:13:14
    |
13  |     let x = a+10;
    |             -^-- {integer}
    |             |
    |             Test
    |
note: an implementation of `Add<_>` might be missing for `Test`
```

Operators

You can however call the method .add(...) directly (this is different than the ***operator +*** as it will try to match the parameters and since Add trait is implemented for &Test, the code will compile !

Rust

```
use std::ops::Add;
struct Test {
    value: i32,
}
impl Add<i32> for &Test {
    type Output = i32;
    fn add(self, rhs: i32) -> Self::Output {
        self.value + rhs
    }
}
fn main() {
    let a = Test { value: 10 };
    let x = a.add(10);
    println!("{x}");
    println!("{}", a.value);
}
```

Output

```
20
10
```

Operators

You can implement multiple Add operations:

Rust

```
struct Test { value: i32 }
impl Add<i32> for Test {
    type Output = i32;
    fn add(self, rhs: i32) -> Self::Output { self.value + rhs }
}
impl Add<Test> for Test {
    type Output = Test;
    fn add(self, rhs: Test) -> Self::Output {
        Test { value: self.value + rhs.value }
    }
}
fn main() {
    let a = Test { value: 10 };
    let b = Test { value: 20 };
    let c = Test { value: 30 };
    let x = a + 10;
    let d = b + c;
    println!("{},{}", x, d.value);
}
```

Output
20,50

In this case we have two forms of Add:

1) Test + i32 => i32
2) Test + Test => Test

Operators

Let's see an example that uses an assignment.

Rust

```
use std::ops::SubAssign;

#[derive(Debug)]
struct Test {
    value: i32
}
impl SubAssign<i32> for Test {
    fn sub_assign(&mut self, rhs: i32)  {
        self.value -= rhs;
    }
}
fn main() {
    let mut a = Test { value: 10 };
    a -= 5;
    println!("{:?}",a);
}
```

Output
5

Operators

Rust also allows overwriting two unary operators (Neg and Not) that corresponds to the operator – (minus) and operator ! (exclamation mark) in front of an expression.

*Rust (**Neg** trait definition)*

```
pub trait Neg {
    type Output;

    fn neg(self) -> Self::Output;
}
```

*Rust (**Not** trait definition)*

```
pub trait Not {
    type Output;

    fn not(self) -> Self::Output;
}
```

OBS: *Keep in mind that this operator receives self (implying a transfer of ownership). This means that if you implement this for a type that does not have the Copy trait, that object will not be available after calling **Neg** or **Not** operators.*

Operators

Let’s see an example that uses unary operators:

Rust

```
use std::ops::{Neg,Not};
#[derive(Debug)]
struct Test {
    value: i32
}
impl Neg for Test {
    type Output = i32;
    fn neg(self)->Self::Output  { -self.value }
}
impl Not for Test {
    type Output = i32;
    fn not(self)->Self::Output  { 100-self.value }
}
fn main() {
    let a = Test { value: 10 };
    let x = -a;
    let b = Test { value: 10 };
    let y = !b;
    println!("{x},{y}");
}
```

Output

-10,90

Operators

Index and IndexMut traits are design to allow index operator overwriting in Rust, with Index being a super-trait for IndexMut.

*Rust (**Index** trait definition)*	*Rust (**IndexMut** trait definition)*

*Rust (**Index** trait definition)*

```
pub Index<Idx: ?Sized>
{
    type Target: ?Sized;
    fn index(&self, index: Idx)->&Self::Output;
}
```

*Rust (**IndexMut** trait definition)*

```
pub trait IndexMut<Idx: ?Sized>: Index<Idx>
{
    fn index_mut(&mut self, index: Idx)->&mut Self::Output;
}
```

Keep in mind the indexing operation in Rust return a reference or a mutable reference. This is a limitation as you can not create and return an object (except for the case where that object is part of the type).

OBS: *As a rule, in cases where index is out of range, you should panic !*

OBS2: ***container[idx]*** *is pretty much the syntax sugar for* ***container.index(idx)***

Operators

Let's see an example that uses index operators:

Rust

```
use std::ops::{Index,IndexMut};

#[derive(Debug)]
struct IPv4 {
    values: [u8;4]
}
impl Index<usize> for IPv4 {
    type Output = u8;
    fn index(&self, index: usize) -> &Self::Output {
        if index<4 { return &(self.values[index]); }
        panic!("Out of bounds !");
    }
}
impl IndexMut<usize> for IPv4 {
    fn index_mut(&mut self, index: usize) -> &mut Self::Output {
        if index<4 { return &mut (self.values[index]); }
        panic!("Out of bounds !");
    }
}
```

Main function

```
fn main() {
    let mut ip = IPv4{values: [0u8;4]};
    println!("IP = {}.{}.{}.{}",ip[0],ip[1],ip[2],ip[3]);
    ip[0] = 192;
    ip[1] = 168;
    ip[2] = 0;
    ip[3] = 1;
    println!("IP = {}.{}.{}.{}",ip[0],ip[1],ip[2],ip[3]);
}
```

Output

```
IP = 0.0.0.0
IP = 192.168.0.1
```

Operators

You can also add multiple indexes:

Rust

```
use std::ops::Index;
struct IPv4 {
    values: [u8;4]
}
impl Index<usize> for IPv4 {
    type Output = u8;
    fn index(&self, index: usize) -> &Self::Output {
        if index<4 { return &(self.values[index]); }
        panic!("Out of bounds !");
    }
}
impl Index<&str> for IPv4 {
    type Output = u8;
    fn index(&self, index: &str) -> &Self::Output {
        match index {
            "first" => { return &(self.values[0]); }
            "second" => { return &(self.values[1]); }
            "third" => { return &(self.values[2]); }
            "forth" => { return &(self.values[3]); }
            _ => { panic!("Invalid index"); }
        }
    }
}
```

Main function

```
fn main() {
    let ip = IPv4{values: [192u8,168,1,123]};
    println!("IP = {}.{}.{}.{}",ip[0],ip[1],ip[2],ip[3]);
    println!("{}",ip["first"]);
    println!("{}",ip["second"]);
}
```

Output

```
IP = 192.168.1.123
192
168
```

Operators

Finally, keep in mind that assignment ('=') **can not be overwritten**.

This is because assignment is used for ownership transfer or Copy semantics (pending on what trait is present).

As such, this operator has to be handled by the compiler itself (as it is part of the move/copy semantics logic that Rust uses internally).

Q & A