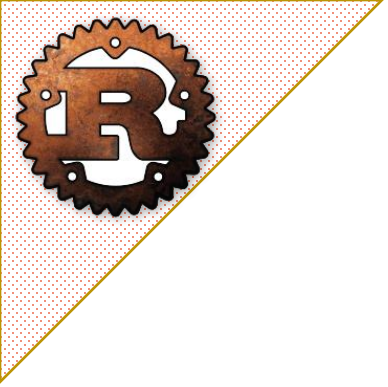
Rust programming

Course – 8

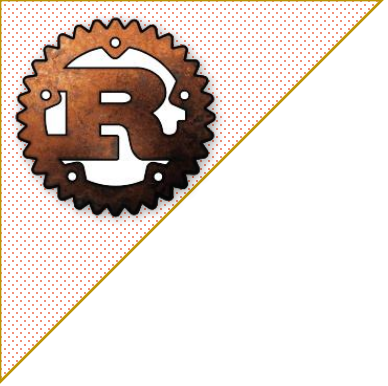
Gavrilut Dragos

Agenda for today

1. Closures
2. Iterators
3. Vectors
4. Sorting data sequences
5. HashMap
6. HashSet
7. BTreeMap
8. BTreeSet
9. Map comparation between C++ and Rust



Closures



Closures

Closures (or lambda functions) are short functions that can be used in different scenarios (e.g. when sorting, or filtering collection of elements).

Closures are widely used with iterators.

The general format of a closure is:

$$|Param_1:Type_1, Param_2:Type_2,\ldots Param_n:Type_n| \rightarrow ReturnType \{ code\text{-}block \}$$

With some observations:

- ***ReturnType*** can be omitted. In this case Rust will try to infer it from the code-block return value; ➔ $|Param_1, Param_2,\ldots Param_n| \{ code\text{-}block \}$
- $\mathbf{Type_1, Type_2 \ldots Type_n}$ can be omitted as well. Rust will try to infer them from the usage.
- The brackets from the code-code block can be omitted (in particular if the code-block is just a simple expression). In this case, the code-block contains just the expression that evaluates the return value; ➔ $|Param_1, Param_2,\ldots Param_n|$ return-value
- If brackets are omitted, the ***ReturnType*** must be omitted as well.

Closures

Let’s see some examples (with a parameter types and return type specified):

Rust

```rust
fn main() {
    let f1 = |x|->i32 { x+1 };
    let cmmdc = |x:i32,y:i32|->i32 {
        let mut a = x;
        let mut b = y;
        while a!=b {
            if a>b { a-=b; } else { b-=a;}
        }
        return a;
    };
    println!("{}",f1(10));
    println!("{}",cmmdc(18,24));
}
```

Output
11 6

Closures

Let’s see some examples (with any type specified):

Rust

```rust
fn main() {
    let f1 = |x| { x+1 };
    let f2 = |x,y| x+y;
    let cmmdc = |x,y| {
        let mut a = x;
        let mut b = y;
        while a!=b {
            if a>b { a-=b; } else { b-=a;}
        }
        a
    };
    println!("{}",f1(10));
    println!("{}",f2(10,20));
    println!("{}",cmmdc(18,24));
}
```

Output

```
11
30
6
```

Closures

Keep in mind that a closure is not a template (even if no type is specified). In the next example, “x” and “y” from f1 are inferred to be of type |x:i32,y:i32|->i32 { x+y } after the first call of printf! Macro.
As such, the usage of a float value will not be allowed.

Rust

```
fn main() {
    let f1 = |x,y| x+y;
    println!("{}",f1(10,20));
    println!("{}",f1(1.2,2.5));
}
```

Error

```
error[E0308]: mismatched types
 --> src\main.rs:4:22
  |
4 |     println!("{}",f1(1.2,2.5));
  |                      ^^^ expected integer, found floating-point number

error[E0308]: mismatched types
 --> src\main.rs:4:26
  |
4 |     println!("{}",f1(1.2,2.5));
  |                          ^^^ expected integer, found floating-point number
```

Closures

A closure does not need to have parameters, it can just be a simple function that prints something on the screen.

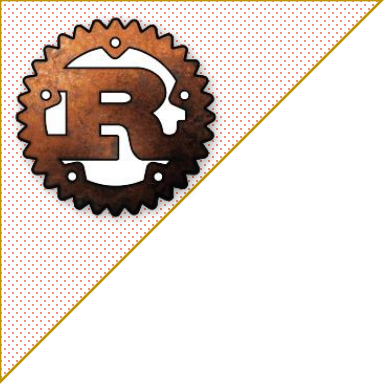
Rust

```rust
fn main() {
    let r = || println!("Rust");
    println!("I like");
    r();
    println!("I like");
    r();
}
```

Output

```
I like
Rust
I like
Rust
```

OBS: *This is in particular useful with captures.*



Closures

A closure can capture local parameters. Let's analyze the following example:

Rust

```
fn main() {
    let x = 1;
    let print_x = || println!("x={}",x);
    print_x();
}
```

Output
X=1

Let's see what's happening in this case (where print_x closure capture the value of "x").

```
mov         dword ptr [x],1

lea         rax,[x]
mov         qword ptr [print_x],rax

lea         rcx,[print_x]
call        main::closure$0
```

Closures

A closure can capture local parameters. Let’s analyze the following example:

Rust

```
fn main() {
    let x = 1;
    let print_x = || println!("x={}",x);
    print_x();
}
```

Output

```
X=1
```

Let’s see what's happening in this case (where print_x closure capture the value of “x”).

```
mov     dword ptr [x],1

lea     rax,[x]
mov     qword ptr [print_x],rax

lea     rcx,[print_x]
```

It’s obvious that a reference to “x” is being stored in print_x.

Closures

A closure can capture local parameters. Let's analyze the following example:

Rust

```
fn main() {
    let x = 1;
    let print_x = || println!("x={}",x);
    print_x();
}
```

Output
X=1

Let's see what's happening in this case (where print_x closure capture the value of "x").

```
mov         dword ptr [x],1

lea         rax,[x]
mov         qword ptr [print_x],rax

lea         rcx,[print_x]
call        main::closure$0
```

The way this call is made, looks like a method call from a class, where RCX register stores this/self.

Closures

A closure can capture local parameters. Let's analyze the following example:

Rust

```
fn main() {
    let x = 1;
    let print_x = || println!("x={}",x);
    print_x();
}
```

Let's see what's happening in this case (where print_x closure capture the value of "x").

Keep in mind that this is an approximation based on how the assembly code looks like.

C++ equivalent

```
struct TempClosure {
    int* x;
    void Run() {
        printf("x=%d",*x);
    }
};
void main() {
    int x = 1;
    TempClosure print_x;
    print_x.x = &x;
    print_x.Run();
}
```

Closures

A closure can capture local parameters. Let's analyze the following example:

Rust

```
fn main() {
    let x = 1;
    let print_x = || println!("x={}",x);
    print_x();
}
```

Let's see what's happening in this case (where print_x closure capture the value of "x").

This is how in reality the code from a closure/lambda function is created.

C++ equivalent (with classes)

```
class TempClosure {
    int& x;
public:
    TempClosure(int& value): x(value) {}
    void operator() () {
        printf("x=%d",x);
    }
};
void main() {
    int x = 1;
    TempClosure print_x(x);
    print_x();
}
```

Closures

So … a closure captures references to variables that are being used in its evaluation.

This also means that every rule that applies to borrowing variables apply here as well.

Rust

```
fn main() {
    let x = 1;
    let print_x = || println!("x={}",x);
    print_x();
    println!("x from main = {}",x);
    print_x();
}
```

Output

```
x=1
x from main = 1
x=1
```

In reality, both print_x and main use immutable references to “x” and as such this code will work just fine.

Closures

Let’s consider this code:

Rust

```rust
fn main() {
    let mut x = 1;
    let print_x = || println!("x={}",x);
    println!("x from main = {}",x);
    print_x();
}
```

Output

```
x from main = 1
x=1
```

“x” is a mutable variable. This code works so the way print_x capture “X” is by an immutable reference (if it were to be a mutable reference, the println! macro would not compile as it would imply the existence of both an immutable and a mutable reference to the same variable).

OBS: *In reality, Rust choses how it borrows references based on how those references are being used in the closure.*

Closures

Let’s consider this code:

Rust

```
fn main() {
    let mut x = 1;
    let print_x = || { println!("x={}",x); x+=1; };
    println!("x from main = {}",x);
    print_x();
}
```

Error

```
borrowed as mutable
 --> src\main.rs:4:33
  |
3 |     let print_x = || { println!("x={}",x);x+=1; };
  |                      --                     - first borrow occurs due
  |                      |                        to use of `x` in closure
  |                      |
  |                      mutable borrow occurs here
4 |     println!("x from main = {}",x);
  |                                 ^ immutable borrow occurs here
5 |     print_x();
  |     ------- mutable borrow later used here
```

In this case, we have modified the closure to increment the value of “x”. For this to happen, “x” must be borrowed as mutable, and as such the println!(…) macro can no longer be used as it implies the existence of both immutable and mutable references to the same variable.

Closures

Let’s consider this code:

Rust

```rust
fn main() {
    let mut x = 1;
    let mut print_x = || { println!("x={}",x);x+=1; };
    print_x();
    print_x();
}
```

Output

```
x=1
x=2
```

Notice that print_x is mutable. This is required as in reality we change the value of one of its data members (the mutable reference to “x”).

Closures

Rust also has a special keyword (move) that can be used to move (assign) the value of the captured elements into the lambda/closure.

Rust

```
fn main() {
    let mut x = 1;
    let mut print_x = move || { println!("x={}",x);x+=1; };
    print_x();
    println!("x from main = {x}");
    print_x();
    println!("x from main = {x}");
}
```

Output

```
x=1
x from main = 1
x=2
x from main = 1
```

Let’s see what happens in this case.

Closures

Rust also has a special keyword (move) that can be used to move (assign) the value of the captured elements into the lambda/closure.

Rust

```
fn main() {
    let mut x = 1;
    let mut print_x = move || { println!("x={}",x);x+=1; };
    print_x();
    println!("x from main = {x}");
    print_x();
    println!("x from main = {x}");
}
```

Output

```
x=1
x from main = 1
x=2
x from main = 1
```

```
mov    eax,dword ptr [x]
mov    dword ptr [print_x],eax
```

Let's see what happens in this case.

Notice the usage of mov instruction, instead of lea. This means that the content of "x" is being transferred to print_x and not a reference towards "x".

Closures

Let’s see a C++ equivalent for this:

```
let mut print_x = || { println!("x={}",x);x+=1; };
```

C++ equivalent (with classes)

```
class TempClosure {
    int& x;
public:
    TempClosure(int& value): x(value) {}
    void operator() () {
        printf("x=%d",x);
        x+=1;
    }
};
void main() {
    int x = 1;
    TempClosure print_x(x);
    print_x();
}
```

```
let mut print_x = move || { println!("x={}",x);x+=1; };
```

C++ equivalent (with classes)

```
class TempClosure {
    int x;
public:
    TempClosure(int value): x(value) {}
    void operator() () {
        printf("x=%d",x);
        x+=1;
    }
};
void main() {
    int x = 1;
    TempClosure print_x(x);
    print_x();
}
```

Closures

Let's see a C++ equivalent for this:

```
let mut print_x = || { println!("x={}",x);x+=1; };
```

C++ equivalent (with classes)

```
class TempClosure {
    int& x;
public:
    TempClosure(int& value): x(value) {}
    void operator() {
        x+=1;
    }
};
void main() {
    int x = 1;
    TempClosure print_x(x);
    print_x();
}
```

In this case a reference is captured.

```
let mut print_x = move || { println!("x={}",x);x+=1; };
```

C++ equivalent (with classes)

```
class TempClosure {
    int x;
public:
    TempClosure(int value): x(value) {}
    void operator() {
        x+=1;
    }
};
void main() {
    int x = 1;
    TempClosure print_x(x);
    print_x();
}
```

In this case the value is captured.

Closures

This means that the previous example that uses move keyword worked (but only because Copy trait is present on i32 type).

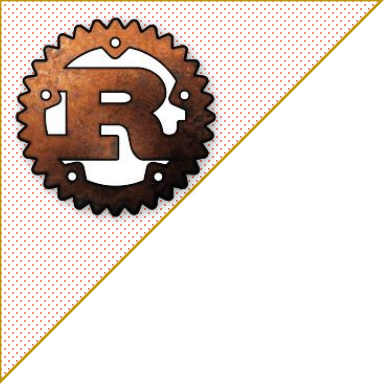
Rust

```
fn main() {
    let mut x = String::from("abc");
    let mut print_x = move || { println!("x={}",x);x.push_str("1"); };
    print_x();
    println!("x from main = {x}");
    print_x();
    println!("x from main = {x}");
}
```

However, if we use a type that does not have a Copy trait (e.g. a String) the code will not compile !

Error

```
error[E0382]: borrow of moved value: `x`
 --> src\main.rs:5:30
  |
2 |     let mut x = String::from("abc");
  |         ----- move occurs because `x` has type `String`, which does not
  |               implement the `Copy` trait
3 |     let mut print_x = move || { println!("x={}",x);x.push_str("1"); };
  |                       -------                    - variable moved due to
  |                       |                            use in closure
  |                       |
  |                       value moved into closure here
4 |     print_x();
5 |     println!("x from main = {x}");
  |                              ^ value borrowed here after move
```



Closures

This means that the previous example that uses move keyword worked (but only because Copy trait is present on i32 type).

Rust

```rust
fn main() {
    let mut x = String::from("abc");
    let mut print_x = move || { println!("x={}",x);x.push_str("1"); };
    print_x();
    print_x();
    print_x();
}
```

Output

```
x=abc
x=abc1
x=abc11
```

Now the code works, but the ownership of “x” has been moved into the print_x closure.

Closures

One solution to move back a value that was captured by a closure is to return it.

Rust

```
fn main() {
    let mut x = String::from("abc");
    let mut print_x = move || { println!("x={}",x);x.push_str("1");return x; };
    x = print_x();
    println!("x = {}",x);
}
```

Output

```
x=abc
x = abc1
```

In this example, first “X” is moved into print_x, then it is moved back.

OBS: *In reality, these type of closures can only be called once (for example in this case, the moment the value of “x” is moved back, the capture print_x can no longer be used).*

Closures

One solution to move back a value that was captured by a closure is to return it.

Rust

```
fn main() {
    let mut x = String::from("abc");
    let mut print_x = move || { println!("x={}",x);x.push_str("1");return x; };
    x = print_x();
    println!("x = {}",x);
    print_x();
}
```

Error

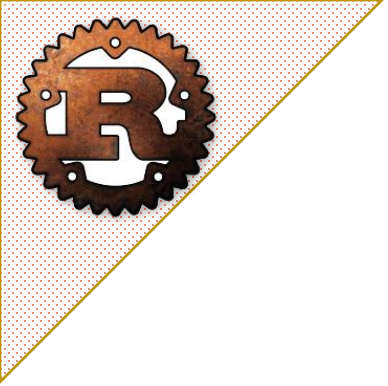
```
error[E0382]: use of moved value: `print_x`
 --> src\main.rs:6:5
  |
4 |     x = print_x();
  |         --------- `print_x` moved due to this call
5 |     println!("x = {}",x);
6 |     print_x();
  |     ^^^^^^^ value used here after move
  |
note: closure cannot be invoked more than once because it moves the variable `x` out of its
environment
 --> src\main.rs:3:75
  |
3 |     let mut print_x = move || { println!("x={}",x);x.push_str("1");return x; };
  |                                                                           ^
note: this value implements `FnOnce`, which causes it to be moved when called
 --> src\main.rs:4:9
4 |     x = print_x();
```

What is **FnOnce** trait ?

Closures

Each closure implicitly implements at least one of the following 3 ***traits***. The decision on what to implement belongs to the compiler, based on the operation and how capture is being used in the closure.

1. FnOnce → closures that can be called only one time (usually a closure that moves a value through the return type out of its context)
2. FnMut → closures that don't move values out of their context but might change the value of a mutable reference that they capture.
3. Fn → closures that don't move values out of their context and don't modify any reference that they capture (they capture immutable references)



Closures

So … what if we want to create a function that returns a closure. Well … the first thing that we need to understand is how to define a pointer/reference to a function (similar to how this is defined in C/C++).

To do this, we will use the keyword fn in the following way:

fn ($Type_1$,$Type_2$, … $Type_n$)->ReturnType

Some examples:

- fn(i32)->i32 ➔ a function that receives a i32 value and returns another i32 value
- fn(&str,usize)->String ➔ a function that receives a &str and an usize value and returns an object of type String
- type name = fn(char)->i32 ➔ creates a type that represents a pointer to a function that receives a parameter of type char and returns an i32

Closures

Let’s see one example that returns a pointer to a function:

Rust

```rust
type MyFunction = fn(i32,i32)->i32;

fn create_add_function() -> MyFunction {
    return |x:i32,y:i32|->i32 { return x+y; }
}
fn main() {
    let add = create_add_function();
    let sub: MyFunction = |x,y| x-y;
    println!("{}, {}",add(1,2),sub(10,4));
}
```

Output
3, 6

In this example, ***MyFunction*** is a type that defines a pointer to a function that takes two i32 parameters and returns an i32 value.

Closures

But what if we want to do something more complex (e.g. to return a closure that captures some variables/parameters):

Rust

```rust
fn create_closure(value: i32)-> fn (i32)->i32 {
    return |x:i32|->i32 { return x / value; };
}

fn main() {
    let f = create_closure(10);
    println!("res = {}",f(50));
}
```

Error

```
error[E0308]: mismatched types
 --> src\main.rs:2:12
  |
1 | fn create_closure(value: i32)-> fn (i32)->i32 {
  |                                  ------------- expected `fn(i32) -> i32` because of
  |                                                return type
2 |     return |x:i32|->i32 { return x / value; };
  |            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected fn pointer, found closure
  |
  = note: expected fn pointer `fn(i32) -> i32`
                found closure `[closure@src\main.rs:2:12: 2:46]`
note: closures can only be coerced to `fn` types if they do not capture any variables
 --> src\main.rs:2:38
  |
2 |     return |x:i32|->i32 { return x / value; };
  |                                      ^^^^^ `value` captured here
```

The short answer is that we can’t.

Closures

That is because a fn(…) is just a pointer while a closure is a struct and its size its unknown (pending on what variables it has captured). The solution is to explain the output based on what it implements: **Fn**, **FnOnce** or **FnMut**

Rust

```rust
fn create_closure(value: i32)-> impl Fn(i32)->i32 {
    return |x:i32|->i32 { return x / value; };
}
fn main() {
    let f = create_closure(10);
    println!("res = {}",f(50));
}
```

Error

```
error[E0373]: closure may outlive the current function, but it borrows
`value`, which is owned by the current function
 --> src\main.rs:2:12
  |
2 |     return |x:i32|->i32 { return x / value; };
  |            ^^^^^^^^^^^^             ----- `value` is borrowed here
  |            |
  |            may outlive borrowed value `value`
```

The approach is ok, but since value is not copied, but only borrowed, when function create_closure ends, “value” lifetime ends and as a result, it can not exist in the returned closure.

Closures

The solution is to move the value that is being capture into the closure. In this case, there is no concern related to lifetime as the result is copied.

Rust

```rust
fn create_closure(value: i32)-> impl Fn(i32)->i32 {
    return move |x:i32|->i32 { return x / value; };
}
fn main() {
    let f = create_closure(10);
    println!("res = {}",f(50));
}
```

Output
res = 5

The code could be written with FnOnce (as we are using move and FnOnce is also implemented).

Closures

But … what happens when we use the impl keyword ? To answer this, let's analyze the following code:

Rust

```
fn create_closure(value: i32) -> impl Fn(i32)->i32 {
    return move |x: i32| -> i32 { return x / value; };
}
fn create_closure2(value: i32) -> impl Fn(i32)->i32 {
    return move |x: i32| -> i32 { return x / value; };
}
fn main() {
    let x1 = create_closure(10);
    let x2 = create_closure2(10);
    let value = 10;
    let x3 = move |x: i32| -> i32 { return x / value; };
    let y1 = x1(20);
    let y2 = x2(20);
    let y3 = x3(20);
    println!("{},{},{}",y1,y2,y3);
}
```

Output
2,2,2

All of these 3 closures are **identical** in terms of their code.

Does this mean that they have the same type ?

Closures

But … what actually happens when we use the impl keyword ? To answer this, let's analyze the following code:

Rust

```
fn create_closure(value: i32) -> impl Fn(i
    return move |x: i32| -> i32 { return x
}
fn create_closure2(value: i32) -> impl Fn(
    return move |x: i32| -> i32 { return x
}
fn main() {
    let x1 = create_closure(10);
    let x2 = create_closure2(10);
    let value = 10;
    let x3 = move |x: i32| -> i32 { return
    let y1 = x1(20);
    let y2 = x2(20);
    let y3 = x3(20);
    println!("{},{},{}",y1,y2,y3);
}
```

```
mov         edx,20
lea         rcx,[x1]
call        first::create_closure::closure$0 (07FF664C31250h)
mov         dword ptr [y1],eax
```

```
mov         edx,20
lea         rcx,[x2]
call        first::create_closure2::closure$0 (07FF664C312E0h)
mov         dword ptr [y2],eax
```

```
mov         edx,20
lea         rcx,[x3]
call        first::main::closure$0 (07FF664C31370h)
mov         dword ptr [y3],eax
```

Closures

But … what actually happens when we use the impl keyword ? To answer this, let's analyze the following code:

Rust

```
fn create_closure(value: i32) -> impl Fn(i
    return move |x: i32| -> i32 { return x
}
fn create_closure2(value: i32) -> impl Fn(
    return move |x: i32| -> i32 { return x
}
fn main() {
    let x1 = create_closure(10);
    let x2 = create_closure2(10);
    let x3 = move |x: i32| -> i32 { return
    let y1 = x1(20);
    let y2 = x2(20);
    let y3 = x3(20);
    println!("{},{},{}",y1,y2,y3);
}
```

```
mov         edx,20
lea         rcx,[x1]
call        first::create_closure::closure$0 (07FF664C31250h)
mov         dword ptr [y1],eax
```

```
mov         edx,20
lea         rcx,[x2]
call        first::create_closure2::closure$0 (07FF664C312E0h)
mov         dword ptr [y2],eax
```

```
mov         edx,20
lea         rcx,[x3]
call        first::main::closure$0 (07FF664C31370h)
mov         dword ptr [y3],eax
```

Notice that even if all closures are identical, each one of them has a **different address** of the code that needs to be run → even if that code is identical on all 3 closures.

Closures

This actually means that each closure is a separate type. From this point of view, two identical closures (in terms of code, parameters, capture and return value) are different from Rust point of view. This is similar to how C++ implements lambda functions, and it also explains why the next code does not compile !

Rust

```
fn create_closure(value: i32) -> impl Fn(i32) -> i32 {
    if value>10 {
        return move |x: i32| -> i32 { return x / value; };
    } else {
        return move |x: i32| -> i32 { return x / value; };
    }
}
fn main() {
    let c = create_closure(10);
}
```

Error

```
error[E0308]: mismatched types
 --> src\main.rs:5:16
1 | fn create_closure(value: i32) -> impl Fn(i32) -> i32 {
  |
5 |         return move |x: i32| -> i32 { return x / value; };
  |                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |                expected closure, found a different closure
  = note: no two closures, even if identical, have the same type
  = help: consider boxing your closure and/or using it as a trait object
```

Closures

Since we can not have a function that returns two different type, the next code can not be compiled.

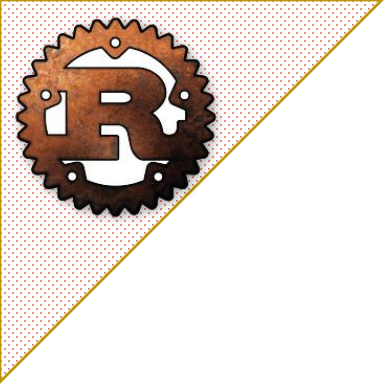
Rust

```rust
fn create_closure(value: i32) -> impl Fn(i32) -> i32 {
    if value>10 {
        return move |x: i32| -> i32 { return x / value; };
    } else {
        return move |x: i32| -> i32 { return x / value; };
    }
}
fn main() {
    let c = create_closure(10);
}
```

Closure of type A

Closure of type B

But… what is the relation between “impl Fn(i32)->i32” and those two closures ?



Closures

Let’s assume the following function: create_closure

```
fn create_closure(value: i32) -> impl Fn(i32) -> i32 {
    return move |x: i32| -> i32 { return x / value; };
}
```

Let’s assume that this closure hase type **ABCD**;

When the compiler sees that the return type uses impl keyword, it search any return type from the function code and assumes that the return type is what the function returns. This means that the previous code will be translated by Rust as follows:

```
fn create_closure(value: i32) -> ABCD {
    return move |x: i32| -> i32 { return x / value; };
}
```

After this, Rust checks to see if **ABCD** implements the trait Fn (with one parameter of type i32) and if it returns an i32 as well. If this is so, then the function is correct, and its return type was inferred from the type of the closure. Furthermore:

```
let c = create_closure(10);
```

Variable “c” will be of type **ABCD** as well.

Closures

The main advantage of this technique is that it allows static linkage of the closure calling method. This means that since we know the type in the compiling phase, we know the memory offset where the calling method of that type lies, and we can call it directly.

However, let's analyze one of the previous errors and see what Rust suggest:

Error

```
error[E0308]: mismatched types
 --> src\main.rs:5:16
1 | fn create_closure(value: i32) -> impl Fn(i32) -> i32 {
  |
5 |         return move |x: i32| -> i32 { return x / value; };
  |                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |                expected closure, found a different closure
  = note: no two closures, even if identical, have the same type
  = help: consider boxing your closure and/or using it as a trait object
```

Closures

So, what does Boxing means in this context ? Well … its like the usage of virtual methods from C++ !

Rust

```rust
fn create_closure(value: i32) -> Box<dyn Fn(i32) -> i32> {
    return Box::new(move |x: i32| -> i32 { return x / value; });
}
fn main() {
    let c = create_closure(10);
    println!("{}",c(50));
}
```

Output
5

Notice the usage of the keyword **dyn** in the definition and the fact that we don't return from the stack but rather allocate a space on heap (a box) from where we will return an object.

We will talk more about **dyn** (short from dynamic ☺) on another course.

Closures

With this change, we can now return two different closure (one that uses multiply, and another one that uses division).

Rust

```rust
fn create_closure(value: i32) -> Box<dyn Fn(i32) -> i32> {
    if value % 2 == 0 {
        return Box::new(move |x: i32| -> i32 { return x / value; });
    } else {
        return Box::new(move |x: i32| -> i32 { return x * value; });
    }
}
fn main() {
    let c1 = create_closure(11);
    let c2 = create_closure(10);
    println!("{},{}",c1(50),c2(50));
}
```

Output
550,5

Iterators

Iterators

Iterators are object that can be used to iterate over an existing collection. They are efficient for cases where index access requires a boundary check, or for collections where index access is not possible (e.g. a linked list – std::collections::LinkedList)

Collection that use iterators:

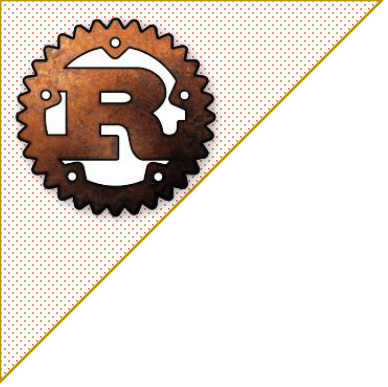
- Arrays
- Vectors
- Maps (BTreeMap, HashMap)
- Sets (BTreeSet, HashSet)

All collections that implement iterators use the trait Iterator defined in **std::iter::Iterator**

Iterators

Basic operation for iterators

Method	Usage
`fn next(&mut self) -> Option<Self::Item>`	Moves to the next element from the collections. This is a virtual method that must be implemented by collection that implements this trait.
`fn count(self) -> usize`	Iterates until the final element and returns the number of iterations.
`fn last(self) -> Option<Self::Item>`	Iterates until the last element from the collections and returns it.
`fn nth(&mut self, n: usize) -> Option<Self::Item>`	Returns the n^{th} items from the current position.
`fn max(self) -> Option<Self::Item>` `fn min(self) -> Option<Self::Item>`	Returns the maximum/minimum number from the current position



Iterators

Let’s see how an iterator works:

Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7,8,9];
    let mut i = a.iter();
    println!("{:?}",i.next());
    println!("{:?}",i.next());
    println!("{:?}",i.nth(3));
    println!("{:?}",i.count());
}
```

Output

```
Some(1)
Some(2)
Some(6)
3
```

Let’s see how iterators work in this case:

When “I” is created, it points to the first element in the array.

Iterators

Let’s see how an iterator works:

Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7,8,9];
    let mut i = a.iter();
    println!("{:?}",i.next());
    println!("{:?}",i.next());
    println!("{:?}",i.nth(3));
    println!("{:?}",i.count());
}
```

Output

```
Some(1)
Some(2)
Some(6)
3
```

Reads the value from current position, and then advances to the next one

Iterators

Let's see how an iterator works:

Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7,8,9];
    let mut i = a.iter();
    println!("{:?}",i.next());
    println!("{:?}",i.next());
    println!("{:?}",i.nth(3));
    println!("{:?}",i.count());
}
```

Output

```
Some(1)
Some(2)
Some(6)
3
```

Reads the value from the 2nd position, and then advances to the next one

Iterators

Let’s see how an iterator works:

Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7,8,9];
    let mut i = a.iter();
    println!("{:?}",i.next());
    println!("{:?}",i.next());
    println!("{:?}",i.nth(3));
    println!("{:?}",i.count());
}
```

Output

```
Some(1)
Some(2)
Some(6)
3
```

Advances 3 positions and reads the value, then move to the next position.

Iterators

Let’s see how an iterator works:

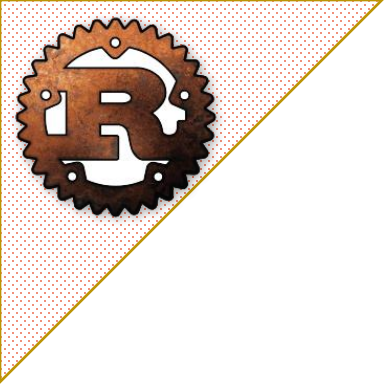
Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7,8,9];
    let mut i = a.iter();
    println!("{:?}",i.next());
    println!("{:?}",i.next());
    println!("{:?}",i.nth(3));
    println!("{:?}",i.count());
}
```

Output

```
Some(1)
Some(2)
Some(6)
3
```

Counts how many elements are until the final !



Iterators

Keep in mind that some methods (like **count**, **last**, **min** or **max**) consume the iterator after using it (notice that count need self and not a reference to self : `fn count(self) -> usize`). This means that the iterator can not be used anymore after calling these methods.

Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7,8,9];
    let mut i = a.iter();
    println!("{:?}",i.count());
    i.next();
}
```

Error

```
error[E0382]: borrow of moved value: `i`
   --> src\main.rs:5:5
    |
3   |     let mut i = a.iter();
    |         ----- move occurs because `i` has type `std::slice::Iter<'_, i32>`,
    |               which does not implement the `Copy` trait
4   |     println!("{:?}",i.count());
    |                     ------- `i` moved due to this method call
5   |     i.next();
    |     ^^^^^^^^ value borrowed here after move
    |
note: this function takes ownership of the receiver `self`, which moves `i`
```

Iterators

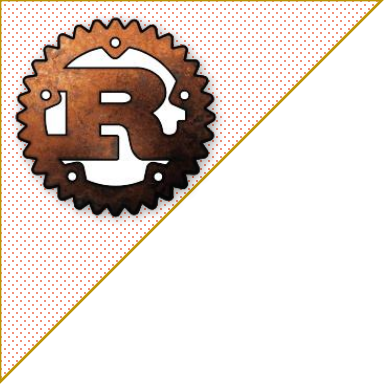
This behavior is different than next() method that does not consume the iterator (even after it reaches the end of the sequence of elements).

Rust

```rust
fn main() {
    let a = vec![1,2,3];
    let mut i = a.iter();
    for _ in 0..10 {
        println!("{:?}",i.next());
    }
}
```

Output

```
Some(1)
Some(2)
Some(3)
None
None
None
None
None
None
None
```

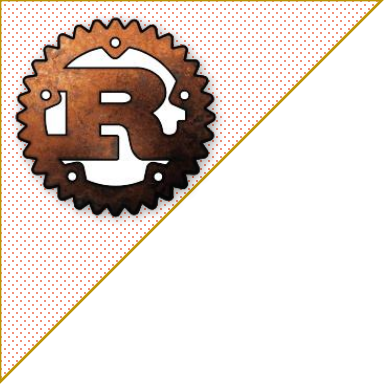


Iterators

Iterators are often used in a for loop. There are 3 forms of iterators that are usually used in such a context:

Method	Usage
`fn iter(&self) -> Iter<T>`	Creates an iterator that return a reference to each element from a collection
`fn iter_mut(&mut self) -> IterMut<T>`	Similar to the previous one, but the reference is mutable, and the value can be modified.
`fn into_iter(self) -> Self::IntoIter`	Notice that this iterator has a parameter of type self (and not &self). This means that this iterator consumes the content of the collection.

OBS: *Keep in mind that without any explicit specification, the for loop will use the into_iter form (e.g `for x in a {...}`). This means that the for loop will consume the element, and "a" will not be available anymore after the for-loop ends.*



Iterators

Let's see some cases where .iter() , .iter_mut() and .into_iter() are used.

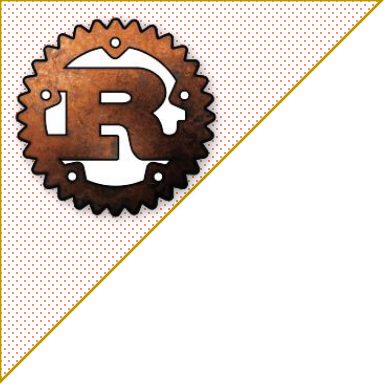
Rust

```rust
fn main() {
    let a = [String::from("abc"),String::from("xyz")];
    for x in a.iter() {
        println!("{x}"); // x is a &String
    }
    println!("{a:?}");
}
```

Output

```
abc
xyz
["abc", "xyz"]
```

In this case x is an immutable reference to every String element from array “a”.



Iterators

Let's see some cases where .iter() , .iter_mut() and .into_iter() are used.

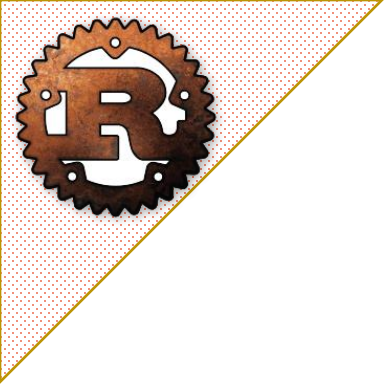
Rust

```rust
fn main() {
    let mut a = [String::from("abc"),String::from("xyz")];
    for x in a.iter_mut() {
        println!("{x}"); // x is a &mut String
        x.push_str("+++");
    }
    println!("{a:?}");
}
```

Output

```
abc
xyz
["abc+++", "xyz+++"]
```

Notice that since we have used iter_mut for this example, we can modify each element from the array “a”.



Iterators

Let's see some cases where .iter() , .iter_mut() and .into_iter() are used.

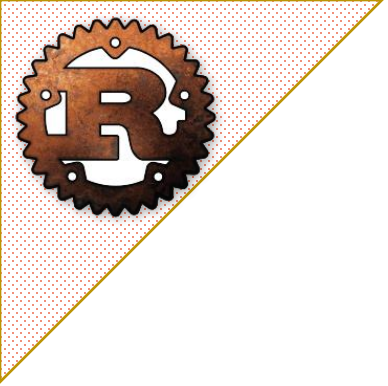
Rust

```rust
fn main() {
    let a = [String::from("abc"),String::from("xyz")];
    for x in a.into_iter() {
        println!("{x}"); // x is a String (takes ownership)
    }
    println!("{a:?}");
}
```

Error

```
error[E0382]: borrow of moved value: `a`
   --> src\main.rs:6:16
    |
2   |     let a = [String::from("abc"),String::from("xyz")];
    |         - move occurs because `a` has type `[String; 2]`, which does not
    |           implement the `Copy` trait
3   |     for x in a.into_iter() {
    |                ---------- `a` moved due to this method call
...
6   |     println!("{a:?}");
    |                ^ value borrowed here after move
```

In this case, each element from array “a” is moved. As a result, the last println!(…) can not work, as “a” was moved.



Iterators

Let's see some cases where .iter() , .iter_mut() and .into_iter() are used.

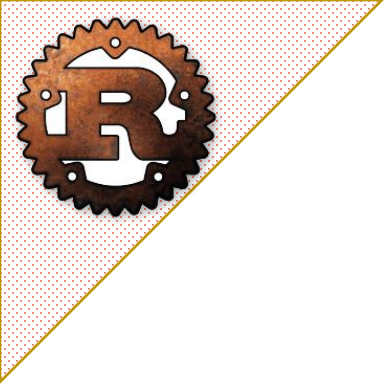
Rust

```rust
fn main() {
    let a = [1,2,3];
    for x in a.into_iter() {
        println!("{x}"); // x is a i32 (a copy !)
    }
    println!("{a:?}");
}
```

Output

```
1
2
3
[1, 2, 3]
```

Keep in mind that into_iter tries uses assignment for each element in the collection. If the element has the Copy trait, it will be copied, otherwise it will be moved. This means that for these cases, the code will compile as the element is not moved !!!



Iterators

If none of these forms are being used, a for-loop uses .into_iter() !

Rust

```rust
fn main() {
    let a = [String::from("abc"),String::from("xyz")];
    for x in a {
        println!("{x}"); // x is a String (takes ownership)
    }
    println!("{a:?}");
}
```

Error

```
error[E0382]: borrow of moved value: `a`
   --> src\main.rs:6:16
    |
2   |     let a = [String::from("abc"),String::from("xyz")];
    |         - move occurs because `a` has type `[String; 2]`, which does not
    |           implement the `Copy` trait
3   |     for x in a {
    |              - `a` moved due to this implicit call to `.into_iter()`
...
6   |     println!("{a:?}");
    |                ^ value borrowed here after move
```

Because of this, elements from “a” will be moved and will no longer be available when println!(…) macro is being called.

Iterators

If none of these forms are being used, a for-loop uses .into_iter() !

Rust

```rust
fn main() {
    let a = [String::from("abc"),String::from("xyz")];
    for x in &a {
        println!("{x}"); // x is a &String
    }
    println!("{a:?}");
}
```

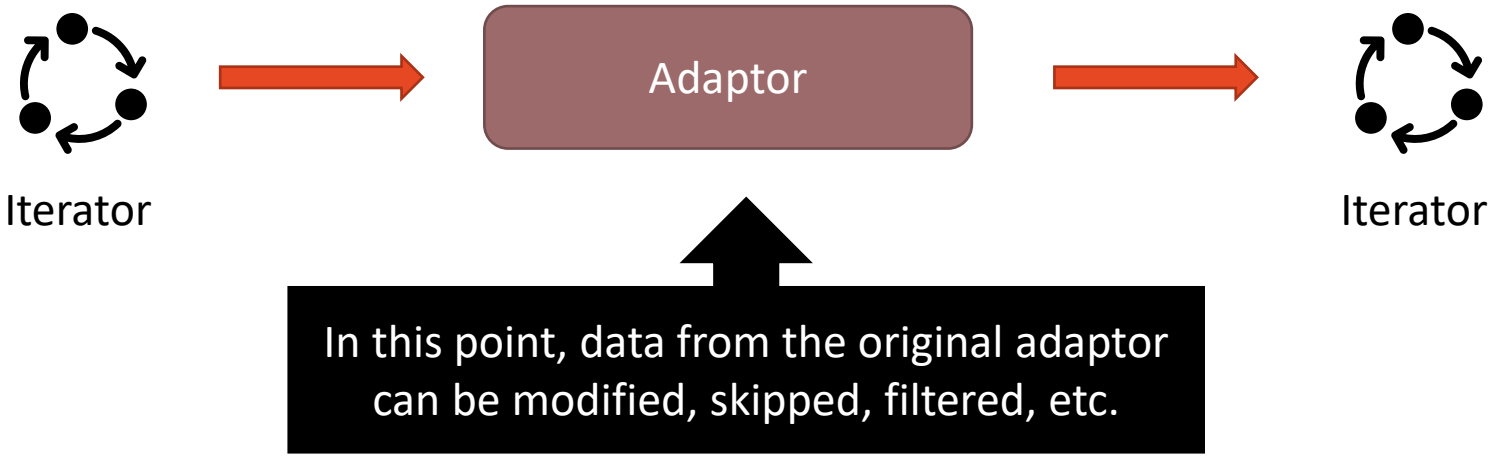
Output

```
abc
xyz
["abc", "xyz"]
```

One solution for this cases is to use the & operator to indicate the for loop to use references instead of moving the entire value.

Iterators

Rust also support various adaptors over an existing iterator, that can allow one to perform quick actions over a data set. Such a construct usually translates into another iterator that can in turn be further used with a different set of adaptors.



Iterators

Adaptors:

Method	Usage
`fn step_by(self, step: usize) -> StepBy<Self>`	Creates a new iterator where every elements will be read from the original use using a step
`fn filter(self, predicate: P) -> Filter<Self, P>`	Filters all elements from the original iterator and returns a new one where only the elements that pass the filter are present
`fn map (self, function: F) -> Map<Self, F>`	Maps all items from an existing iterator into another one, applying a conversion over each element.
`fn skip_while(self, predicate: P) -> SkipWhile<Self, P>` `fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>`	Skips/Takes a number of elements based on a predicated
`fn skip(self, n: usize) -> Skip<Self>` `fn take(self, n: usize) -> Take<Self>`	Skips/takes a number of “n” elements from the original iterator
`fn inspect(self, f: F) -> Inspect<Self, F>`	Runs function “f” for each element, and then pass it on. Useful for debugging purposes.

Iterators

Iterate over a list with step 2:

Rust

```rust
fn main()
{
    let a = vec![1,2,3,4,5,6,7,8,9];
    for i in a.iter().step_by(2)
    {
        println!("{}",*i);
    }
}
```

or

Rust

```rust
fn main() {
    let a = vec![1,2,3,4,5,6,7,8,9];
    let i = a.iter()
             .step_by(2)
             .inspect(|x| println!{"{:?}",*x});
    for _ in i {}
}
```

Output

```
1
3
5
7
9
```

Notice that we have chained several iterators to obtain the same result.

Iterators

Rust also has a **for_each** adaptor that iterates over all elements. While it has the same purpose as a regular for keyword, it can also be used in large chains of adaptors as the final one to trigger the iteration.

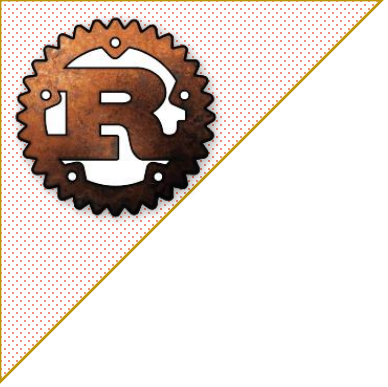
Method	Usage
`fn for_each(self, f: F)`	Iterates over a collection and calls function “f” for each element.

Rust

```rust
fn main() {
    let a = vec![1, 2, 3, 4, 5];
    a.iter()
     .inspect(|x| println! {"Before step: {:?}",*x})
     .step_by(2)
     .for_each(|x| println! {"After step: {:?}",*x});
}
```

Output

```
Before step: 1
After step: 1
Before step: 2
Before step: 3
After step: 3
Before step: 4
Before step: 5
After step: 5
```



Iterators

Rust also has a **for_each** adaptor that iterates over all elements. While it has the same purpose as a regular for keyword, it can also be used in large chains of adaptors as the final one to trigger the iteration.

Method	Usage
`fn for_each(self, f: F)`	Iterates over a collection and calls function “f” for each element.

Rust

```
fn main() {
    let a = vec![1,2,3,4,5,6,7,8,9];
    a.iter()
     .step_by(2)
     .inspect(|x| println!("{:?}", *x))
     .for_each(|_|{});
}
```

Output

```
1
3
5
7
```

Notice the usage of `|_|{}` ➔ this is called an empty closure (a function that does nothing)

Iterators

Let’s see a more complex example that takes a vector, filters out all even elements, multiply the rest of the elements by 2 and then sums them all up.

Rust

```rust
fn main() {
    let a = vec![1,2,3,4,5,6,7,8,9];
    let s:i32 = a.iter()
                 .filter(|x| *x % 2 == 0)
                 .inspect(|x| println!{"Filtered {:?}",*x})
                 .map(|x| x*2)
                 .inspect(|x| println!{"Mapped to {:?}",*x})
                 .sum();
    println!("sum is {}",s);
}
```

Output

```
Filtered 2
Mapped to 4
Filtered 4
Mapped to 8
Filtered 6
Mapped to 12
Filtered 8
Mapped to 16
sum is 40
```

Iterators

You can also use .skip and .take to perform operations over a continuous sub-set from a collection. The next example sums up the next four elements from the 3rd element in a collection:

Rust

```
fn main() {
    let a = vec![1,2,3,4,5,6,7,8,9];
    let s:i32 = a.iter()
                 .skip(3)
                 .take(4)
                 .inspect(|x| println!{"Value {:?}",*x})
                 .sum();
    println!("sum is {}",s);
}
```

Output

```
Value 4
Value 5
Value 6
Value 7
sum is 22
```

Iterators

Another wildly use adaptor is .collect(). This adaptor allows transforming a collection into another one.

Method	Usage
`fn collect<B>(self) -> B`	Iterates over a collection and converts it into another collection.

How to use .collect():

1. let var:*type* = …iterators chain.. .collect();
2. let var = …iterators chain.. .collect::<*type*>();

Usually, the first version is preferred as it avoids the *turbo-fish* format.

Iterators

Let’s take the previous example and build a new vector instead of computing a sum:

Rust

```rust
fn main() {
    let a = vec![1,2,3,4,5,6,7,8,9];
    let s = a.iter()
             .skip(2)
             .take(3)
             .collect::<Vec<_>>();
    println!("result is {:?}",s);
}
```

Rust

```rust
fn main() {
    let a = vec![1,2,3,4,5,6,7,8,9];
    let s: Vec<_> = a.iter()
                     .skip(2)
                     .take(3)
                     .collect();
    println!("result is {:?}",s);
}
```

Output

```
result is [3, 4, 5]
```

Iterators

This method is often used to convert an array into a vector:

Rust

```rust
fn main() {
    let a = [1,2,3,4,5];
    let b: Vec<_> = a.iter().collect();
    println!("a =  {:?}",a);
    println!("b =  {:?}",b);
}
```

Output

```
a =  [1, 2, 3, 4, 5]
b =  [1, 2, 3, 4, 5]
```

Notice the Vec<_> notation. The underline (_) tells Rust that the type of the vector must be inferred from the result. We should also mention that since .iter() uses references, vector b will be of type Vec<&i32> !

Iterators

Another adaptor is partition. It has a similar purpose as collect , but in this case, it tries to split an existing collection into two partitions. The closure function serves this purpose (elements where it returns true will be added to the first partition, and the rest of them to the second partition).

Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7];
    let (p1,p2): (Vec<_>,Vec<_>) = a.into_iter().partition(|x| *x>4);
    println!("Partition 1 = {:?}",p1);
    println!("Partition 2 = {:?}",p2);
}
```

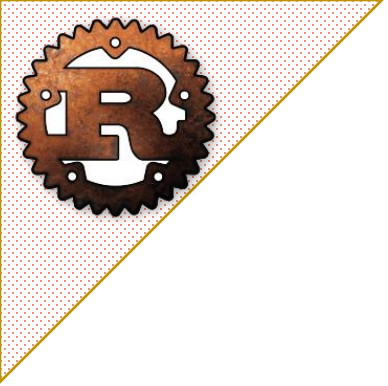
Output

```
Partition 1 = [5, 6, 7]
Partition 2 = [1, 2, 3, 4]
```

Iterators

You may have noticed that for the previous example we have used **into_iter** instead of the regular **iter**. So … what is the difference.

1. into_iter() is an iterator that moves the element (meaning that after you iterate over it, all elements from the sequence are no longer available
2. iter() uses references (meaning that after you iterate over a sequence of data, that sequence is still available).



Iterators

Let’s consider the following example:

Rust

```
fn main() {
    let a = [String::from("ABC"),String::from("123")];
    for i in a.into_iter() {
        println!("{:?}",i);
    }
    println!("a = {:?}",a);
}
```

Error

```
error[E0382]: borrow of moved value: `a`
   --> src\main.rs:7:25
    |
3   |     let a = [String::from("ABC"),String::from("123")];
    |         - move occurs because `a` has type `[String; 2]`, which does not implement the
    |           `Copy` trait
4   |     for i in a.into_iter() {
    |                ---------- `a` moved due to this method call
...
7   |     println!("a = {:?}",a);
    |                         ^ value borrowed here after move
    |
note: this function takes ownership of the receiver `self`, which moves `a`
    |
267 |     fn into_iter(self) -> Self::IntoIter;
```

Iterators

However, if we change the previous example from using iter() instead of into_iter() it works as we will no longer move the object when iterating but use a reference instead.

Rust

```rust
fn main() {
    let a = [String::from("ABC"),String::from("123")];
    for i in a.iter() {
        println!("{:?}",i);
    }
    println!("a = {:?}",a);
}
```

Output

```
"ABC"
"123"
a = ["ABC", "123"]
```

Iterators

Another useful adaptor is .find() that can be used to search for a specific item that matches a criteria.

Method	Usage
`fn find<P>(&mut self, predicate: P) -> Option<Self::Item>`	Finds an element that is matched by the predicate P

Rust

```
fn main() {
    let a = [1,2,3,4,5,6,7];
    let b = a.iter().find(|&&x| x==4);
    println!("{:?}",b);
}
```

Output
Some(4)

***OBS**: .find(f) is equivalent to filter(f).next()*
***OBS**: find predicate is defined as P: FnMut(&Self::Item) -> bool . This means that if an iterator uses references the closer will have to use a double reference (a reference over the reference provided by the original iterator).*

Other functionalities

All of the iterators and adaptors previously described solve some problems.

However, there are some cases that require a different type of functionality.

1. Peekable
2. enumerate
3. DoubleEndedIterator
4. ExactSizeIterator
5. Infinite iterator loops

Iterators (Peekable)

One problem with iterators is that once .next() is called you can not go back. This in fact is a problem as you need the value that if you need the value you get from .next() to decide if you want to iterate further or not.

To solve this, Rust added a new adaptor called Peekable (that allows one to read the next value, but not move to the next position).

Let's analyze the following problem:

- We have a list of numbers: 1,2,3,….
- We want to find number 3, but we don't want to move next to it (to number 4).

Iterators (Peekable)

Let’s analyze the following problem:

- We have a list of numbers: 1,2,3,….
- We want to find number 3, but we don’t want to move next to it (to number 4).

Rust

```rust
fn main() {
    let a = [1,2,3,4,5,6,7];
    let mut i = a.iter().peekable();
    loop {
        if i.peek().is_none() { break; }
        let v = **i.peek().unwrap();
        if v == 3 { break; }
        i.next();
    }
    println!("{}",i.next().unwrap());
}
```

Output
3

Iterators (enumerate)

There are situation where while during iteration an index is required. While these sort of scenarios can easily be solved by creating an external index, and incrementing it after each iteration, Rust also provides an adaptor (called enumerate) that does the same thing.

Rust

```rust
fn main() {
    let a = ["John", "Mary", "Mike", "George"];
    for i in a.iter().enumerate() {
        println!("{:?}",i);
    }
}
```

Output

```
(0, "John")
(1, "Mary")
(2, "Mike")
(3, "George")
```

Iterators (Infinite loops)

Iterators can be used to create infinite loops via .cycle() method. This method creates an iterator that when it reaches the last element will reset itself to point to the first one, thus creating an infinite cycle.

Rust

```rust
fn main() {
    let a = [1,2,3,4,5];
    for x in a.iter().cycle() {
        print!("{x},");
    }
}
```

Output

```
1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,
4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,
2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,
5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1, .....
```

Iterators (DoubleEndedIterator)

There are cases where you might need to read elements from both ends of a collection. For this cases, there is a special trait (called DoubleEndedIterator) that if implemented allows one to also read elements from the end of the collection via:

Method	Usage
`fn next_back(&mut self) -> Option<Self::Item>`	Moves to the previous element from the collections starting from its end.
`fn nth_back(&mut self, n: usize) -> Option<Self::Item>`	Returns the previous n^{th} items from the current position.

OBS: *It is important to notice that back and forward iterator can not meet (one can not go beyond the other one).*

Iterators (DoubleEndedIterator)

Let's see some example:

Rust

```rust
fn main() {
    let a = [1,2,3,4,5,6];
    let mut it = a.iter();
    while let Some(x) = it.next_back() {
        print!("{x},");
    }
}
```

Output
6,5,4,3,2,1

Keep in mind that this is not an **iterator**, but a trait (meaning that not all types that implement ***Iterator*** trait also implement ***DoubleEndedIterator*** trait). That depends on how the data is store in that collection. For example, a single linked list can not implement such a trait, while a vector, array or a double linked list can.

Iterators (DoubleEndedIterator)

Let’s see some example:

Rust

```rust
fn main() {
    let a = [1,2,3,4,5,6];
    let mut it = a.iter();
    for i in 0..5 {
        println!("Next from Front => {:?}",it.next());
        println!("Previous from Back => {:?}",it.next_back());
        println!("----------------------------");
    }
}
```

Output

```
Next from Front => Some(1)
Previous from Back => Some(6)
----------------------------
Next from Front => Some(2)
Previous from Back => Some(5)
----------------------------
Next from Front => Some(3)
Previous from Back => Some(4)
----------------------------
Next from Front => None
Previous from Back => None
----------------------------
Next from Front => None
Previous from Back => None
```

Notice that once back and front iteration reach the middle of the array, the result of both .next and .next_back methods is None ! This is because back and next iteration can not go beyond the other one.

Iterators (ExactSizeIterator)

Another interesting trait is **ExactSizeIterator**. This trait provides a function (**len**) that returns the number of steps until the end of the iteration (the moment from when .next() method will start returning None instead of Some.

Rust

```
fn main() {
    let a = ["A","B","C","D"];
    let mut it = a.iter();
    while let Some(i) = it.next() {
        println!("Element: {i}, {} steps until end", it.len());
    }
}
```

Output

```
Element: A, 3 steps until end
Element: B, 2 steps until end
Element: C, 1 steps until end
Element: D, 0 steps until end
```

Vectors

Vectors

Vectors are sequences of elements of the same type that can increase or decrease in size dynamically. Just like **std::vector** from C++ standard, a vector in Rust is a template/generic object.

To create a vector, use one of the following forms:

a) `let mut a: Vec<type> = Vec::new()`
b) `let mut a = Vec::<type>::new()`
c) `let mut a: Vec<type> = Vec::with_capacity(capacity)`
d) `let mut a = Vec::<type>::with_capacity(capacity)`
e) `let mut a = Vec::from(array)`
f) or use the macro vec! for quick initialization of a vector.

Vectors

Let’s see some examples on how to build a vector.

Rust

```rust
fn main() {
    let mut v1 = Vec::<i32>::new();
    let v2 = Vec::<u32>::with_capacity(100);
    let v3 = vec![1u8, 2,3,4];          // type is Vec<u8>
    let v4 = vec!["123","abc","xyz"];  // type is Vec<&str>
    let v5 = Vec::from([1,2,3]); // type is Vec<i32>
    println!("{:?}",v1);
    println!("{:?}",v2);
    println!("{:?}",v3);
    println!("{:?}",v4);
    println!("{:?}",v5);
}
```

Output

```
[]
[]
[1, 2, 3, 4]
["123", "abc", "xyz"]
[1, 2, 3]
```

Vectors

Basic operations (insert/add/remove) for vectors

Method	Usage
`fn push(&mut self, value: T)`	Adds a new elements at the end of the vector
`fn insert(&mut self, index:usize, element: T)`	Inserts an element at a specific position in the vector. This function panics If index is outside vector boundaries.
`fn append(&mut self, other: &mut Self)`	Appends the element of another vector of the same type to the current one.
`fn pop(&mut self) -> Option<T>`	Returns the last element in the vector (if any) or None for empty vectors
`fn remove(&mut self, index: usize) -> T` `fn swap_remove(&mut self, index: usize) -> T`	Removes the element from a specific index in the vector. This function panics If index is outside vector boundaries.
`fn clear(&mut self)`	Clears the content of the vector leaving the capacity of the vector un-affected.

Vectors

Let’s see some examples on how to build a vector.

Rust

```rust
fn main() {
    let mut v = Vec::<i32>::new();
    for i in 1..10 {
        v.push(i);
    }
    println!("remove from index #2 => {}",v.remove(2));
    while let Some(i) = v.pop() {
        println!("Pop element: {i}")
    }
}
```

Output

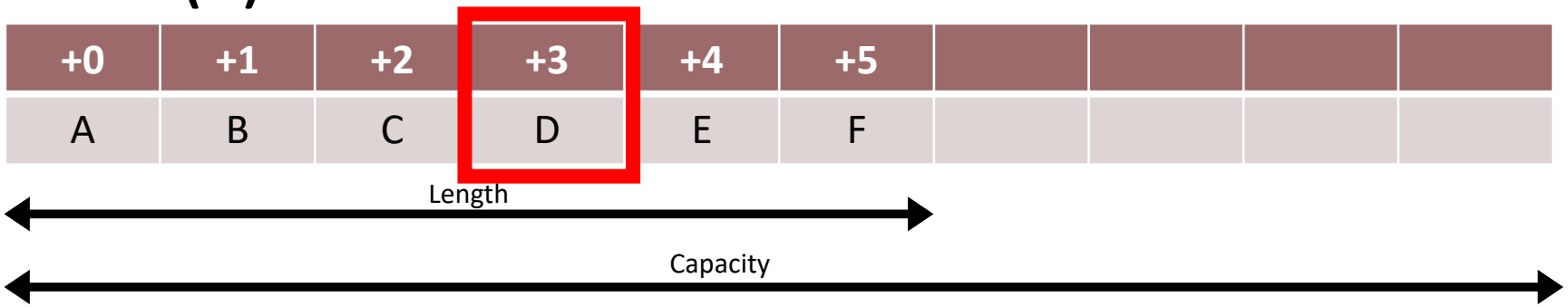
```
remove from index #2 => 3
Pop element: 9
Pop element: 8
Pop element: 7
Pop element: 6
Pop element: 5
Pop element: 4
Pop element: 2
Pop element: 1
```

Vectors

Vector has 2 remove methods to remove an element from a specific position:

1) remove(…)
2) swap_remove()

remove(…)

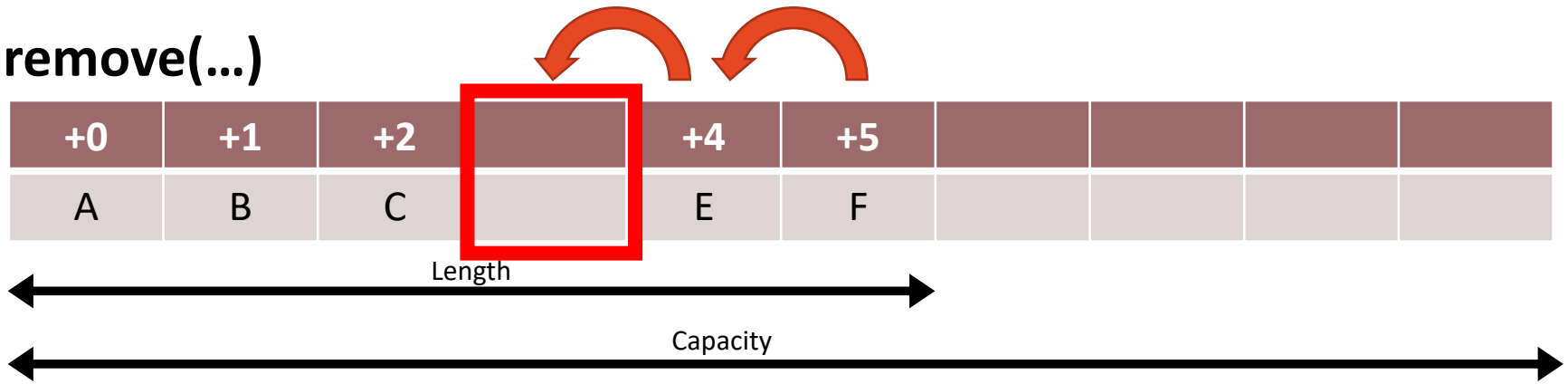


*** We want to remove element with index 3 (the 4th element)**

Vectors

Vector has 2 remove methods to remove an element from a specific position:

1) remove(…)
2) swap_remove()



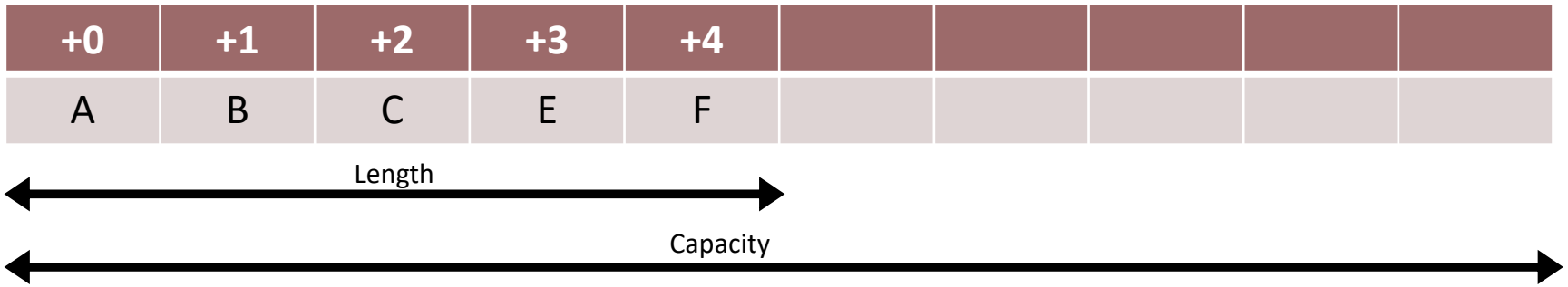
*** After we delete the element, we will move all of the existing elements after index 3 one position to the left**

Vectors

Vector has 2 remove methods to remove an element from a specific position:

1) remove(…)
2) swap_remove()

remove(…)



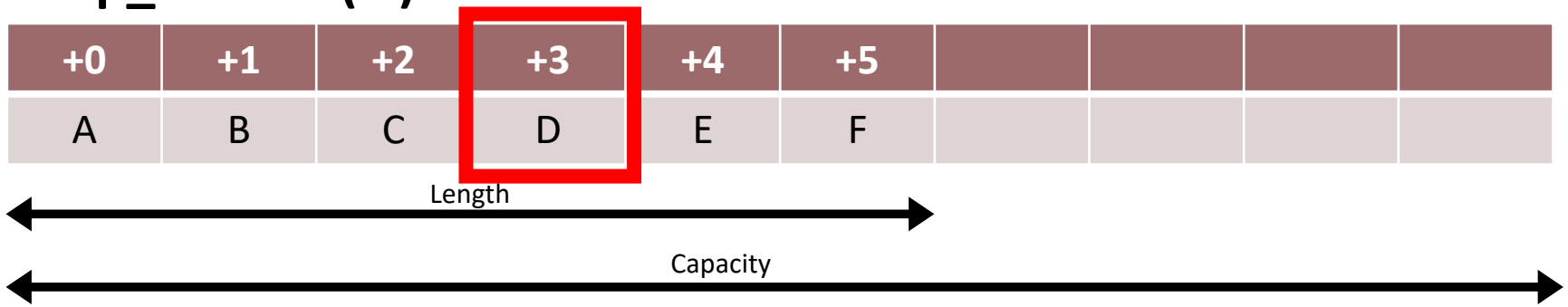
*** Length is decreased by one, the capacity remains the same**
*** Operation cost: O(n)**

Vectors

Vector has 2 remove methods to remove an element from a specific position:

1) remove(…)
2) swap_remove()

swap_remove(…)



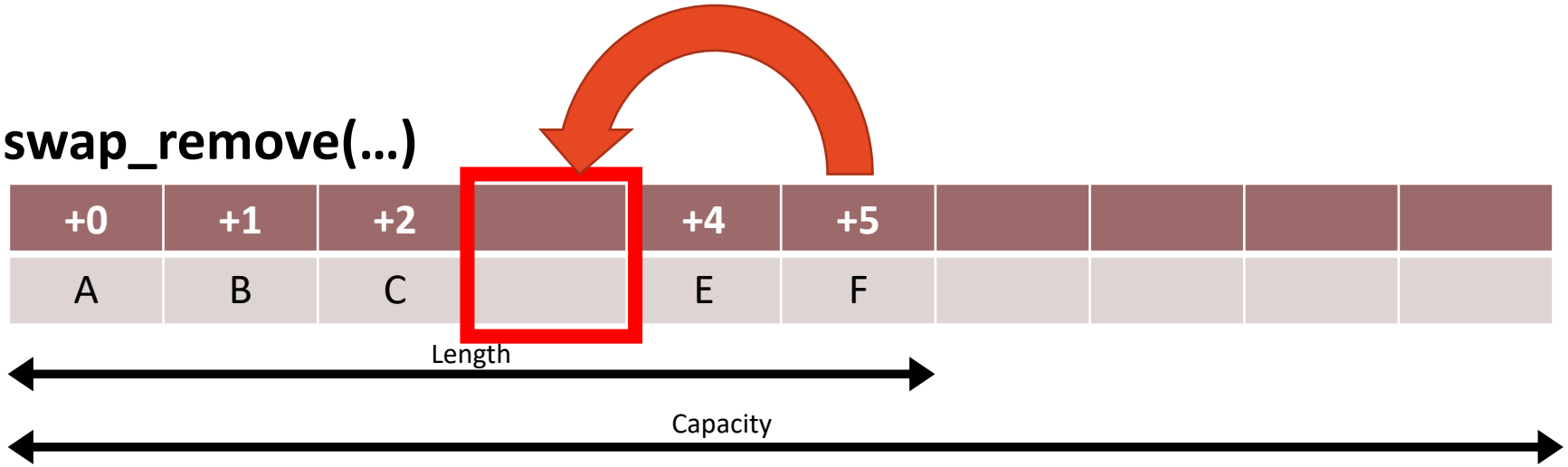
*** We want to remove element with index 3 (the 4th element)**

Vectors

Vector has 2 remove methods to remove an element from a specific position:

1) remove(…)
2) swap_remove()

swap_remove(…)



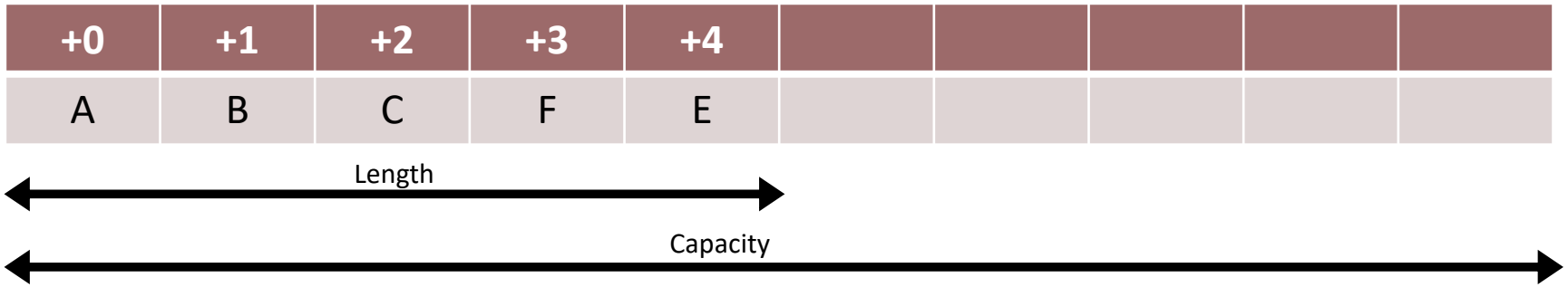
*** After we delete the element, we swap the last element with the one that we have just removed**

Vectors

Vector has 2 remove methods to remove an element from a specific position:

1) remove(…)
2) swap_remove()

swap_remove(…)



*** Length is decreased by one, the capacity remains the same**
*** Operation cost: O(1) ; notice that the order has changed !**

Vectors

Vector has 2 remove methods to remove an element from a specific position:

1) remove(…)
2) swap_remove()

Overview:

1) Use swap_remove if **you are not interested in the order of the elements** from the vector, otherwise use remove
2) **swap_remove** has a complexity of O(1)
3) **remove** has a complexity of O(n)

Vectors

Generic allocation/resize and infos for vectors

Method	Usage
`fn len(&self) -> usize`	Returns the length of the vector
`fn capacity(&self) -> usize`	Returns the capacity of the vector
`fn is_empty(&self) -> bool`	True if vector length is 0, false otherwise
`fn reserve(&mut self, additional: usize)`	Reserve additional elements (on top of the existing one)
`fn truncate(&mut self, len: usize)`	Truncates a vector to a specific len, dropping the extra elements.
`fn try_reserve(&mut self, additional: usize) -> Result<(), TryReserveError>`	Tries to reserve some space (if allocation fails it does not panic like reserve method does); instead, it returns Err.
`fn shrink_to_fit(&mut self)`	Reduces the capacity of the vector to match the exact number of elements from the vector.

Vectors

Example using previous methods.

Rust

```rust
fn add_range(v:&mut Vec<i32>,start:i32, end:i32) {
    for i in start..end+1 { v.push(i); }
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
}
fn main() {
    let mut v = Vec::<i32>::new();
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
    add_range(&mut v, 1, 2);
    v.reserve(32);
    add_range(&mut v, 3, 8);
    v.shrink_to_fit();
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
    v.truncate(4);
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
}
```

Output

```
size=0, capacity=0, v=[]
size=2, capacity=4, v=[1, 2]
size=8, capacity=34, v=[1, 2, 3, 4, 5, 6, 7, 8]
size=8, capacity=8, v=[1, 2, 3, 4, 5, 6, 7, 8]
size=4, capacity=8, v=[1, 2, 3, 4]
```

Vectors

Example using previous methods.

Rust

```
fn add_range(v:&mut Vec<i32>,start:i32, end:i32) {
    for i in start..end+1 { v.push(i); }
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
}
fn mai
    let mut v = Vec::<i32>::new();
    ),v.capacity(),v);
    v.reserve(32);
    add_range(&mut v, 3, 8);
    v.shrink_to_fit();
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
    v.truncate(4);
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
}
```

A close range can be enforced by using ..=

```
for i in start..=end { v.push(i); }
```

Output

```
size=0, capacity=0, v=[]
size=2, capacity=4, v=[1, 2]
size=8, capacity=34, v=[1, 2, 3, 4, 5, 6, 7, 8]
size=8, capacity=8, v=[1, 2, 3, 4, 5, 6, 7, 8]
size=4, capacity=8, v=[1, 2, 3, 4]
```

Vectors

Keep in mind that transferring a variable into a vector might imply change of ownership (in this case after v.push(t) is executed, variable "t" lifetime has ended as a result of "t" being moved into the vector).

Rust

```
#[derive(Debug)]
struct Test { v1: i32, v2: f32, v3: char }
fn main() {
    let mut v: Vec<Test> = Vec::new();
    let t = Test{v1:5,v2:1.3,v3:'A'};
    v.push(t);
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
    println!("t = {:?}",t);
}
```

Error

```
error[E0382]: borrow of moved value: `t`
  --> src\main.rs:12:25
   |
9  |     let t = Test{v1:5,v2:1.3,v3:'A'};
   |         - move occurs because `t` has type `Test`, which does
   |           not implement the `Copy` trait
10 |     v.push(t);
   |            - value moved here
12 |     println!("t = {:?}",t);
   |                         ^ value borrowed here after move
```

Vectors

However, if we implement the Copy trait, the code will compile.

Rust

```rust
#[derive(Debug,Copy,Clone)]
struct Test { v1: i32, v2: f32, v3: char }
fn main() {
    let mut v: Vec<Test> = Vec::new();
    let t = Test{v1:5,v2:1.3,v3:'A'};
    v.push(t);
    println!("size={}, capacity={}, v={:?}",v.len(),v.capacity(),v);
    println!("t = {:?}",t);
}
```

Output

```
size=1, capacity=4, v=[Test { v1: 5, v2: 1.3, v3: 'A' }]
t = Test { v1: 5, v2: 1.3, v3: 'A' }
```

Vectors

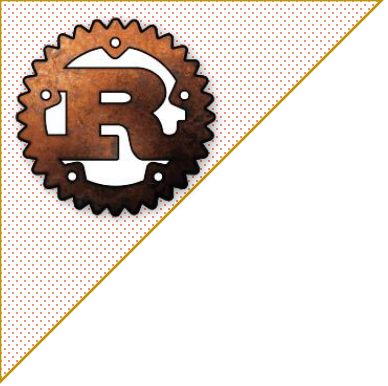
Let’s compare how efficient vector push method is for both C++ and Rust.

Rust

```rust
extern "system" {
        fn GetTickCount64 () -> u64;
}
fn get_time () -> u64 {
        unsafe { GetTickCount64() }
}
#[derive(Debug,Copy,Clone)]
struct Test { v1: i32, v2: f32, v3: char, v4: [u8;256] }
fn main() {
    let mut v: Vec<Test> = Vec::new();
    let t = Test{v1:5,v2:1.3,v3:'A',v4:[48u8;256]};
    let start = get_time();
    for i in 0..10_000_000 {
        v.push(t);
    }
    let end = get_time();
    println!("{}",end-start);
}
```

C++

```cpp
#include <Windows.h>
#include <vector>
struct Test {
    int v1;
    float v2;
    char32_t v3;
    uint8_t v4[256];
};
void main() {
    std::vector<Test> v;
    Test t;
    auto start = GetTickCount64();
    for (auto i = 0; i < 10000000; i++) {
        v.push_back(t);
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```



Vectors

Both codes were teste in the same environment, for 10 times and the average was recorded. All tests were run on x64 architecture (Debug and Release). Times are measures in milliseconds.
Keep in mind that GetTickCount function has an error margin of 16ms.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Average
C++ (Debug)	2562	2562	2640	2578	2578	2515	2562	2578	2562	2547	**2568**
Rust (Debug)	1922	1844	1860	1843	1812	1813	1797	1781	1828	1813	**1831**
C++ (Release)	1828	1781	1797	1781	1781	1781	1797	1797	1821	1797	**1796**
Rust (Release)	1750	1750	1688	1719	1687	1703	1719	1687	1703	1687	**1709**

Vectors

As a general conclusion, when it comes to vectors (and copying object not moving them), Rust is faster than C/C++ (in both debug and release modes).

We should point out that the build that was tested for C++ was compiled with Microsoft compiler (cl.exe) and it does not reflect results for gcc or clang (that might optimize the C++ code in a different way).

However, the question still remains on what's different in Rust vs C++ in terms of how vector works ?

Vectors

Let’s see how Rust allocates memory for the previous case.

Rust

```rust
struct Test {
        v1: i32,
        v2: f32,
        v3: char,
        v4: [u8;256]
}
fn main() {
    let mut v: Vec<Test> = Vec::new();
    let t = Test{v1:5,v2:1.3,v3:'A',v4:[48u8;256]};
    let mut capacity = v.capacity();
    for i in 0..10_000_000 {
        v.push(t);
        let c = v.capacity();
        if c>capacity {
            println!("Size={:08X},Capacity={:08X}",v.len(), c);
            capacity = c;
        }
    }
}
```

Output

```
Size=00000001,Capacity=00000004
Size=00000005,Capacity=00000008
Size=00000009,Capacity=00000010
Size=00000011,Capacity=00000020
Size=00000021,Capacity=00000040
...
Size=00040001,Capacity=00080000
Size=00080001,Capacity=00100000
Size=00100001,Capacity=00200000
Size=00200001,Capacity=00400000
Size=00400001,Capacity=00800000
Size=00800001,Capacity=01000000
```

Vectors

Let’s see how Rust allocates memory for the previous case.

Rust

```rust
struct Test {
        v1: i32,
        v2: f32,
        v3: char,
        v4: [u8;256]
}
fn main() {
    let mut v: Vec<Test> = Vec::new();
    let t = Test{v1:5,v2:1.3,v3:'A',v4:[48u8;256]};
    let mut capacity = v.capacity();
    for i in 0..10_000_000 {
        v.push(t);
        let c = v.capacity();
        if c>capacity {
            println!("Size={:08X},Capacity={:08X}",v.len(), c);
            capacity = c;
        }
    }
}
```

Rust uses an allocator that **doubles** the capacity, with the start capacity of 4 elements.

Output

```
Size=00000001,Capacity=00000004
Size=00000005,Capacity=00000008
Size=00000009,Capacity=00000010
Size=00000011,Capacity=00000020
Size=00000021,Capacity=00000040
…
Size=00040001,Capacity=00080000
Size=00080001,Capacity=00100000
Size=00100001,Capacity=00200000
Size=00200001,Capacity=00400000
Size=00400001,Capacity=00800000
Size=00800001,Capacity=01000000
```

Vectors

Let’s see how C++ allocates memory for the previous case.

C++

```cpp
#include <vector>
struct Test {
    int v1;
    float v2;
    char32_t v3;
    uint8_t v4[256];
};
void main() {
    std::vector<Test> v;
    Test t;
    auto capacity = v.capacity();
    for (auto i = 0; i < 10000000; i++) {
        v.push_back(t);
        auto c = v.capacity();
        if (c > capacity) {
            printf("Size=%08X, Capacity=%08X\n", (uint32_t)v.size(), (uint32_t)c);
            capacity = c;
        }
    }
}
```

Output

```
Size=00000001, Capacity=00000001
Size=00000002, Capacity=00000002
Size=00000003, Capacity=00000003
Size=00000004, Capacity=00000004
Size=00000005, Capacity=00000006
Size=00000007, Capacity=00000009
Size=0000000A, Capacity=0000000D
Size=0000000E, Capacity=00000013
Size=00000014, Capacity=0000001C
Size=0000001D, Capacity=0000002A
Size=0000002B, Capacity=0000003F
…
Size=00240B5D, Capacity=0036110A
Size=0036110B, Capacity=0051198F
Size=00511990, Capacity=0079A656
Size=0079A657, Capacity=00B67981
```

Vectors

Let’s see how C++ allocates memory for the previous case.

C++

```cpp
#include <vector>
struct Test {
    int v1;
    float v2;
    char32_t v3;
    uint8_t v4[256];
};
void main() {
    std::vector<Test> v;
    Test t;
    auto capacity = v.capacity();
    for (auto i = 0; i < 10000000; i++) {
        v.push_back(t);
        auto c = v.capacity();
        if (c > capacity) {
            printf("Size=%08X, Capacity=%08X\n", (uint32_t)v.size(), (uint32_t)c);
            capacity = c;
        }
    }
}
```

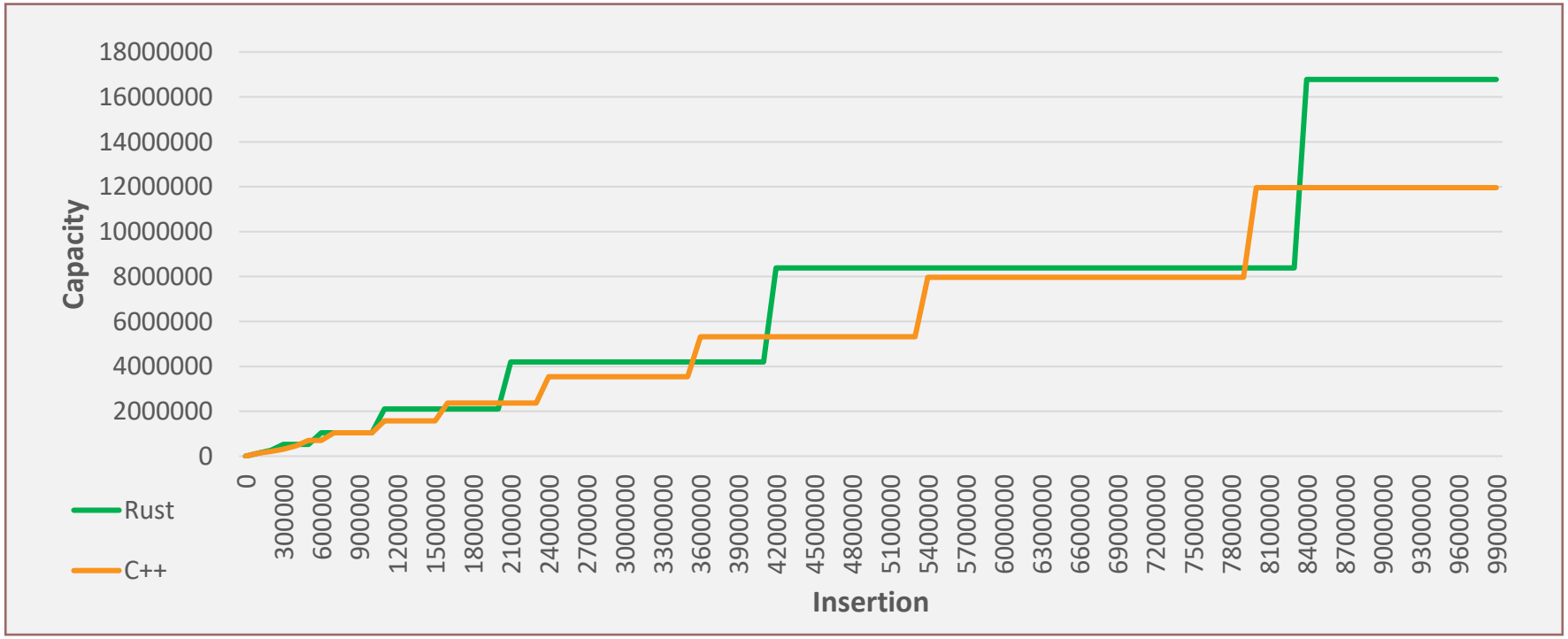
C++ has a different strategy where the growth factor is **1.5** (for the cl.exe/MS implementation)

Output

```
Size=00000001, Capacity=00000001
Size=00000002, Capacity=00000002
Size=00000003, Capacity=00000003
Size=00000004, Capacity=00000004
Size=00000005, Capacity=00000006
Size=00000007, Capacity=00000009
Size=0000000A, Capacity=0000000D
Size=0000000E, Capacity=00000013
Size=00000014, Capacity=0000001C
Size=0000001D, Capacity=0000002A
Size=0000002B, Capacity=0000003F
…
Size=00240B5D, Capacity=0036110A
Size=0036110B, Capacity=0051198F
Size=00511990, Capacity=0079A656
Size=0079A657, Capacity=00B67981
```

Vectors

So … the difference lies in how growth algorithm works for those two cases (Rust and C++).



Vectors

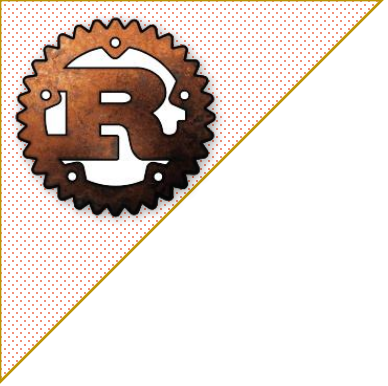
So … lets see the behavior if we reserve the memory from the start.

Rust

```rust
extern "system" {
         fn GetTickCount64 () -> u64;
}
fn get_time () -> u64 {
         unsafe { GetTickCount64() }
}
#[derive(Debug,Copy,Clone)]
struct Test { v1: i32, v2: f32, v3: char, v4: [u8;256] }
fn main() {
    let mut v: Vec<Test> = Vec::with_capacity(10_000_000);
    let t = Test{v1:5,v2:1.3,v3:'A',v4:[48u8;256]};
    let start = get_time();
    for i in 0..10_000_000 {
        v.push(t);
    }
    let end = get_time();
    println!("{}",end-start);
}
```

C++

```cpp
#include <Windows.h>
#include <vector>
struct Test {
    int v1;
    float v2;
    char32_t v3;
    uint8_t v4[256];
};
void main() {
    std::vector<Test> v;
    Test t;
    v.reserve(10000000);
    auto start = GetTickCount64();
    for (auto i = 0; i < 10000000; i++) {
        v.push_back(t);
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```



Vectors

Test were performed in a similar manner like the previous ones (Debug and Release, 10 iterations and we compute the average).

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Average
C++ (Debug)	782	766	781	766	781	766	797	797	828	781	**784**
Rust (Debug)	984	1094	1063	1031	875	1032	1016	860	906	859	**972**
C++ (Release)	547	532	531	515	500	515	516	500	531	531	**521**
Rust (Release)	532	500	531	516	562	531	547	547	547	515	**532**

Keep in mind that there is an error margin of 16 ms for GetTickCount API. This means that the difference between C++ and Rust is insignificant (we can consider both at the same level).

Vectors

To access an element from an index in the vector use the […] index operator. if the index is out of range, the code will panic.

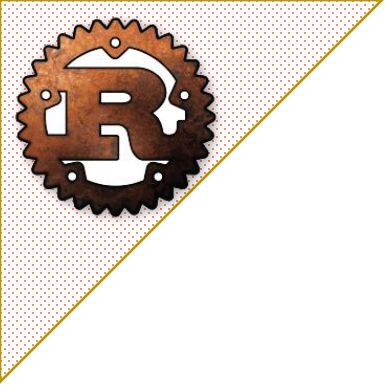
Rust

```rust
#[derive(Debug)]
struct Test {
        v1: i32,
        v2: f32,
        v3: char,
        v4: [u8;256]
}
fn main() {
    let mut v = Vec::<Test>::new();
    v.push(Test{v1:1,v2:1.2,v3:'A',v4:[15;256]});
    let b = v[0];
    println!("{:?}",b);
}
```

Error

```
error[E0507]: cannot move out of index of `Vec<Test>`
 --> src\main.rs:6:13
  |
6 |     let b = v[0];
  |             ^^^^
  |             |
  |             move occurs because value has type `Test`, which
  |             does not implement the `Copy` trait
  |             help: consider borrowing here: `&v[0]`
```

In this particular case, the code will not compile because the assignment is equivalent to moving an element from the vector.



Vectors

There are two solution to the previous problem:
1. borrow the value of the element from index 0

Rust

```rust
#[derive(Debug)]
struct Test {
        v1: i32,
        v2: f32,
        v3: char,
        v4: [u8;256]
}
fn main() {
    let mut v = Vec::<Test>::new();
    v.push(Test{v1:1,v2:1.2,v3:'A',v4:[15;256]});
    let b = &v[0];
    println!("{:?}",b);
}
```

Output

```
Test { v1: 1, v2: 1.2, v3: 'A',
v4: [15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
…
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15] }
```

Vectors

There are two solution to the previous problem:
2. implement the Copy trait for structure Test

Rust

```rust
#[derive(Debug,Copy,Clone)]
struct Test {
        v1: i32,
        v2: f32,
        v3: char,
        v4: [u8;256]
}
fn main() {
    let mut v = Vec::<Test>::new();
    v.push(Test{v1:1,v2:1.2,v3:'A',v4:[15;256]});
    let b = v[0];
    println!("{:?}",b);
}
```

Output

```
Test { v1: 1, v2: 1.2, v3: 'A',
v4: [15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
…
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15,
15] }
```

Vectors

A vector is iterable (for read and write):

Rust

```rust
fn main() {
    let v = vec![1,2,3,4,5];
    let mut s = 0;
    for i in v {
        s+=i;
    }
    println!("{}",s);
}
```

Output

```
15
```

Read

Rust

```rust
fn main() {
    let mut v = vec![1,2,3,4,5];
    let mut s = 0;
    for i in &mut v {
        *i = (*i) * 2;
    }
    for i in v {
        s+=i;
    }
    println!("{}",s);
}
```

Output

```
30
```

Write

Vectors

A vector has a special function (call retain) that can be used to keep only some elements that have a specific characteristics:

Method	Usage
`fn retain<F>(&mut self, mut f: Function)`	Retains all elements that for which a function replies with **true**
`fn retain_mut<F>(&mut self, mut f: Function)`	

Rust

```rust
fn odd(value: &i32)->bool {
    return value % 2 == 1;
}
fn main() {
    let mut v = vec![1,2,3,4,5];
    v.retain(odd);
    println!("{:?}",v);
}
```

Output

```
[1,3,5]
```

Vectors

Vectors can be easily converted into slices (much like an array can). We can do this via the range operator .. or via as_slice() / as_mut_slice() methods.

Rust

```
fn sum(list: &[i32])->i32 {
    let mut s = 0;
    for i in list {
        s+=*i;
    }
    s
}
fn main() {
    let mut v = vec![1,8,3,9,10,6,4];
    println!("Sum = {}",sum(v.as_slice()));
    let slice = &v[1..4];
    println!("Slice = {:?}, sum = {}",slice,sum(slice));
    println!("Vector = {:?}",v);
}
```

Output

```
Sum = 41
Slice = [8, 3, 9], sum = 20
Vector = [1, 8, 3, 9, 10, 6, 4]
```

Vectors

A vector can also be split off into two parts resulting two vectors. For this use the method .split_off(index).

Rust

```rust
fn main() {
    let mut v = vec![1,2,3,4,5];
    let mut b = v.split_off(2);
    println!("v={:?}, capacity={}",v,v.capacity());
    println!("b={:?}, capacity={}",b,b.capacity());
}
```

Output

```
v=[1, 2], capacity=5
b=[3, 4, 5], capacity=3
```

In this case, we split from the index 2 (meaning that the first two elements will remain in the original vector, and the rest of them will be transferred to another vector).

The first vector capacity remains untouched (in this case 5).

Keep in mind that this method creates another vector (and allocates memory for it).

Vectors

A vector also has a set of methods called drain that can be used to remove some elements from the vector based on a specific logic or range.

Method	Usage
`fn drain<R>(&mut self, range: R) -> Drain<'_, T, A>`	Removes all elements from a vector within a specific range
`fn drain_filter<F>(&mut self, filter: F) -> DrainFilter<'_, T, F, A>`	Removes all element from a vector that are filtered by a given function.

Note that drain methods return an iterator over the elements that need to be removed → and if used in conjunction with the .collect() method from the iterator, these methods can be used to split a vector in a different way.

** **drain_filter** is considered an **unstable feature** (we will not discuss about this method)*

Vectors

Let’s see some examples:

Rust

```rust
fn main() {
    let mut v = vec![1,2,3,4,5];
    v.drain(3..);
    println!("{:?}",v)
}
```

Output

```
[1, 2, 3]
```

Rust

```rust
fn main() {
    let mut v = vec![1,2,3,4,5];
    let mut b: Vec<i32> = v.drain(3..).collect();
    println!("{:?}",v);
    println!("{:?}",b);
}
```

Output

```
[1, 2, 3]
[4, 5]
```

Vectors

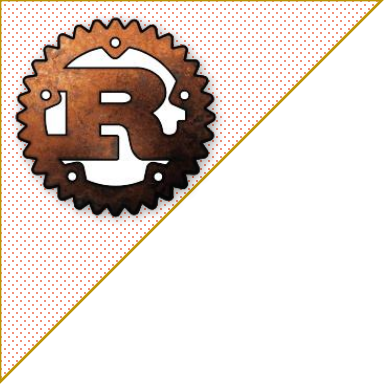
Its also important to notice that drain(Range) method keeps a mutable reference to the vector. This is more efficient as it does not allocate extra space for the elements that are being drained. It also means that if you obtain this iterator, you can not modify the existing vector until you consume the drain or you drop it.

Rust

```
fn main() {
    let mut v = vec![1,2,3,4,5];
    let d = v.drain(3..);
    println!("{:?}",v);
    let mut s = 0;
    for i in d { s+= i; }
}
```

Error

```
error[E0502]: cannot borrow `v` as immutable because it is also
borrowed as mutable
 --> src\main.rs:5:21
  |
4 |     let d = v.drain(3..);
  |             ------------ mutable borrow occurs here
5 |     println!("{:?}",v);
  |                     ^ immutable borrow occurs here
6 |     let mut s = 0;
7 |     for i in d { s+= i; }
  |              - mutable borrow later used here
```



Sorting

Sorting

One of the most common problem when dealing with data sequences (e.g. a vector, an array, a slice) is to be able to sort them.

Rust has several sort algorithms in place that take into consideration:

- If the sort is stable or not
- Worst case
- Memory consumption
- Sort using a key

Sorting

Any mutable vector , array or slice have several ***sort*** related methods:

Method (Vector/Slice/Array)	Usage

`fn sort(&mut self)`
`fn sort_by<F>(&mut self, mut compare: F)`
`fn sort_by_key<K, F>(&mut self, mut f: F)`
`fn sort_by_cached_key<K, F>(&mut self, f: F)`
	Stable sort (keeps the order of the equal elements).

`fn sort_unstable(&mut self)`
`fn sort_unstable_by<F>(&mut self, mut compare: F)`
`fn sort_unstable_by_key<K, F>(&mut self, mut f: F)`
	Unstable sort (may reorder equal elements).

`fn is_sorted(&self) -> bool`
`fn is_sorted_by<F>(&self, mut compare: F) -> bool`
`fn is_sorted_by_key<F, K>(&self, f: F) -> bool`
	Checks if elements are already sorted.

Sorting

Sort algorithms:

Methods	Infos:
sort sort_by	• **Algorithm**: iterative merge sort inspired by timsort • **Worst case**: O(n*log(n)) • **Memory**: for large vectors allocates extra memory (half the size of the vector) • **Stable**: does not change the order of equal elements • **Best for**: nearly sorted sequences
sort_by_key	*Similar with sort and sort_by, except for complexity* • **Worst case**: O(m*n + n*log(n)), O(m) = time needed to compute the key
sort_by_cached_key	• **Algorithm**: pattern defeating quick sort • **Worst case**: O(m*n + n*log(n)), O(m) = time needed to compute the key • **Memory**: in worst case it allocates the size of the vector/slice • **Stable**: does not change the order of equal elements • **Guarantees**: A key is computed at most one time

Sorting

Sort algorithms:

Methods	Infos:
sort_unstable sort_unstable_by	• **Algorithm**: pattern defeating quick sort • **Worst case**: O(n*log(n)) • **Memory**: Swap in done in-place (no extra allocation) • **Unstable**: it may change the order of equal elements
sort_unstable_by_key	*Similar with sort and sort_by, except for complexity* • **Worst case**: O(m*n + n*log(n)), O(m) = time needed to compute the key

OBS: *if elements order is not at issue, unstable sorts are generally faster and require less memory than a regular sort. The only cases where stable sort is recommended is if the sequence of data contains elements that are already partially sorted.*

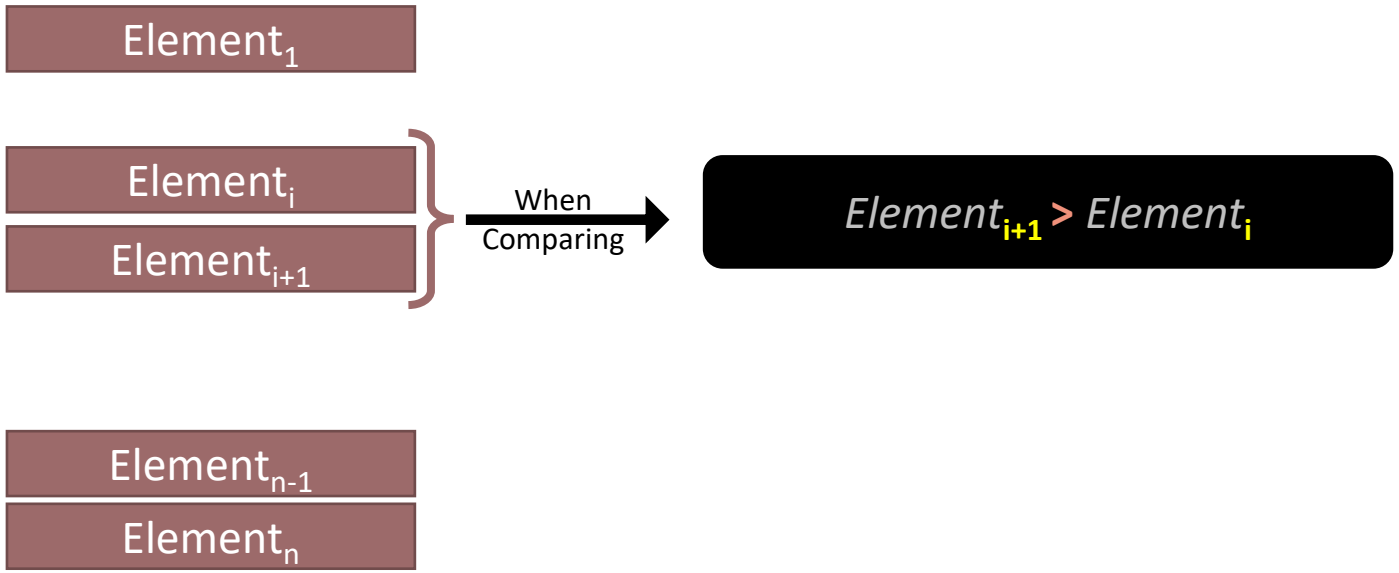
Sorting

What is the difference between regular sort, sort by and sort by a key (or the caching form of sort by key) ?

Sorting

What is the difference between regular sort, sort by and sort by a key (or the caching form of sort by key) ?

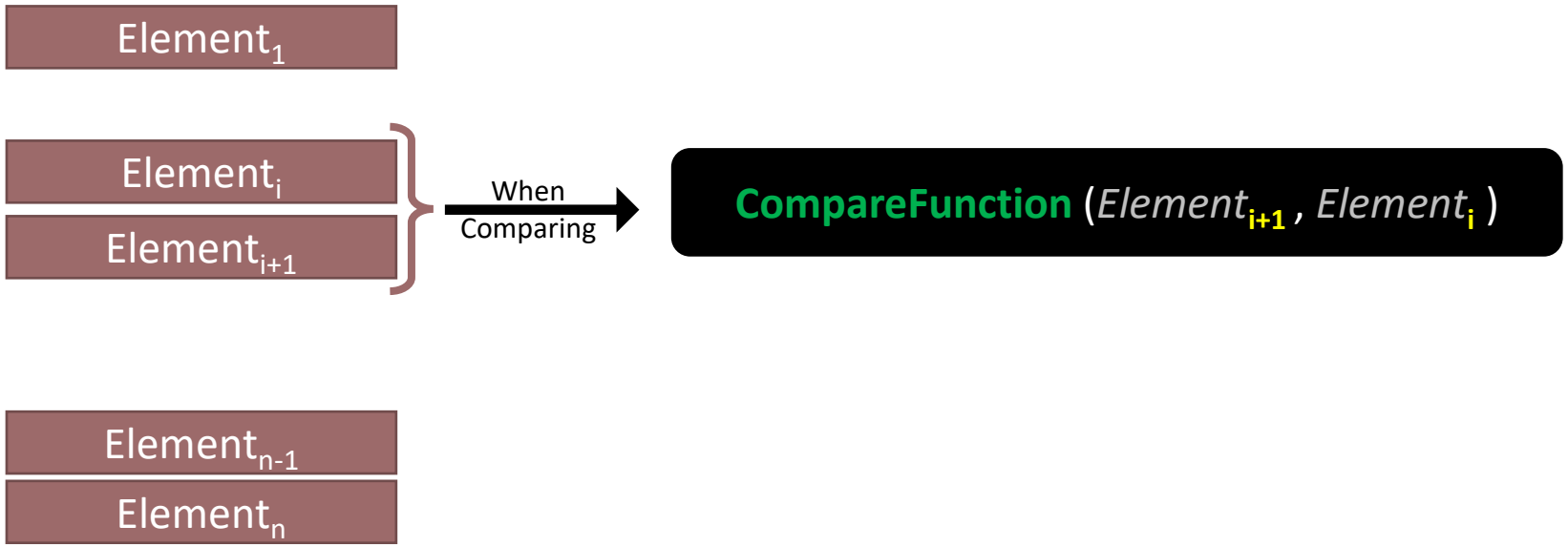
Regular sort:



Sorting

What is the difference between regular sort, sort by and sort by a key (or the caching form of sort by key) ?

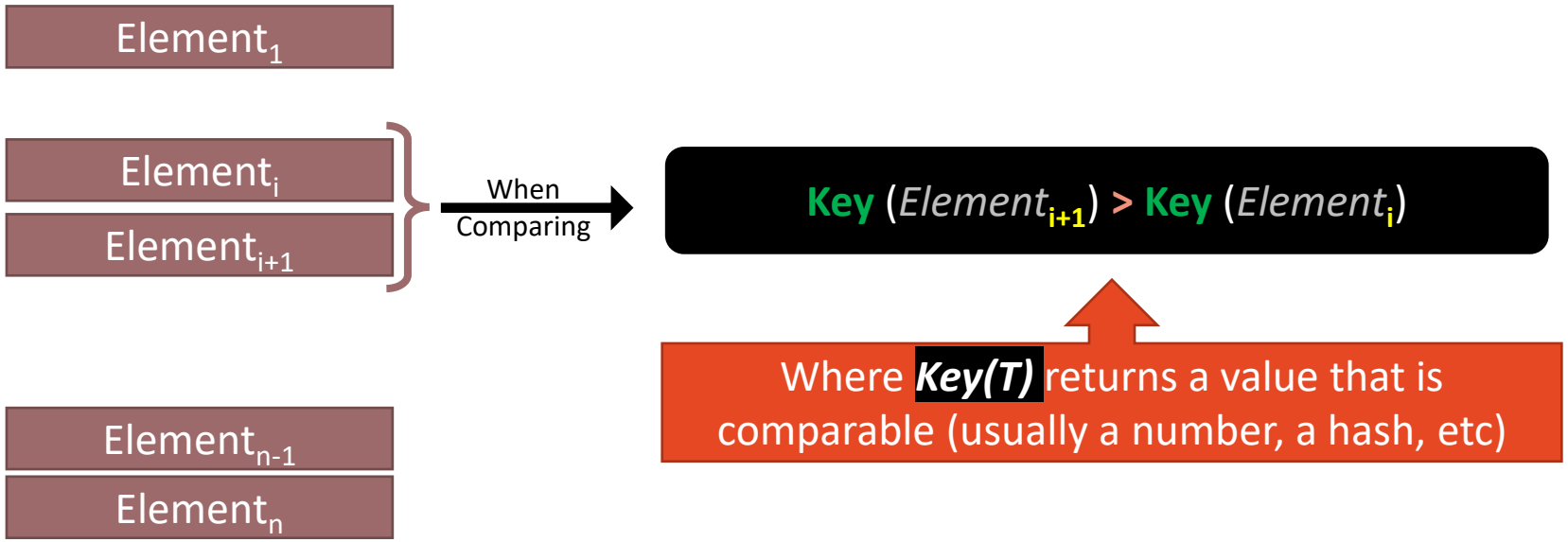
Sort by:



Sorting

What is the difference between regular sort, sort by and sort by a key (or the caching form of sort by key) ?

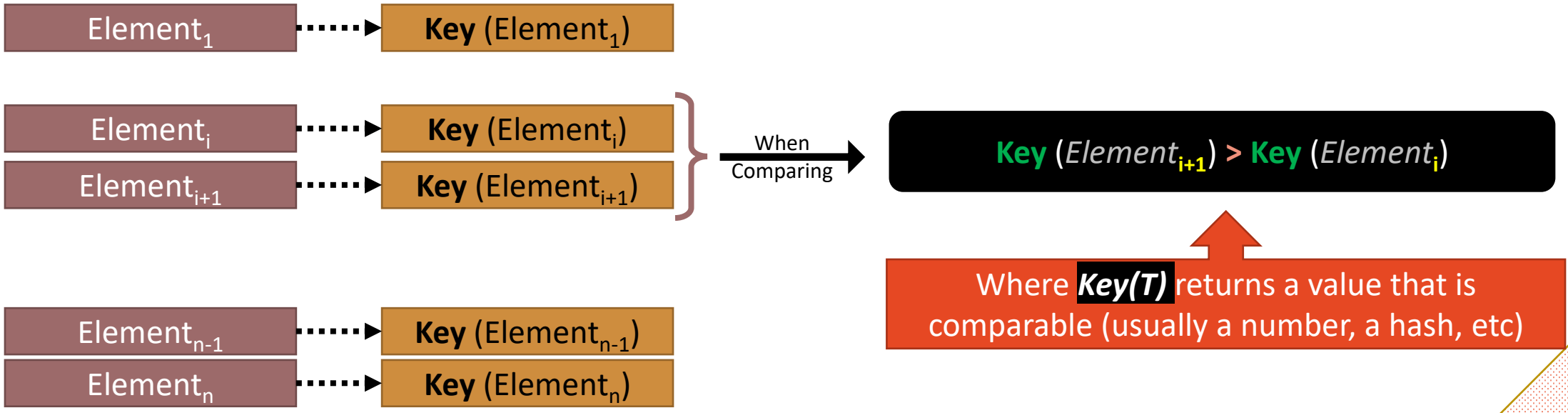
Sort by KEY:



Sorting

What is the difference between regular sort, sort by and sort by a key (or the caching form of sort by key) ?

Sort by KEY (cached):



Sorting

Let’s see some examples:

Rust

```rust
fn main() {
    let mut v = vec![1,9,6,2,9,3,6,8,3,6,1,3,7,8];
    v.sort_unstable();
    println!("{:?}",v);
}
```

Output

```
[1, 1, 2, 3, 3, 3, 6, 6, 6, 7, 8, 8, 9, 9]
```

and

Rust

```rust
fn absolute_value(value:&i32) -> i32 {
    if *value < 0 { -(*value) } else  { *value }
}
fn main() {
    let mut v = vec![1,-9,6,-2,9,-3,-6,-8,3,-6,-1,3,7,8];
    v.sort_by_key(absolute_value);
    println!("{:?}",v);
}
```

Output

```
[1, -1, -2, -3, 3, 3, 6, -6, -6, 7, -8, 8, -9, 9]
```

Sorting

Let’s discuss an even more complex example. We will start by defining the structure Student and a function that can be used to create such an object that will further be used in our examples.

Rust

```rust
#[derive(Debug)]
struct Student {
    math: u8,
    english: u8,
    name: String
}
impl Student
{
    fn new(studentName: &str, mathGrade: u8, englishGrad: u8) -> Student {
        Student {
            name: String::from(studentName),
            math: mathGrade,
            english: englishGrad
        }
    }
}
```

We will discuss more about constructors for structs in another course !

Sorting

Let’s try to sort a list of students:

Rust

```
fn main() {
    let mut v = vec![
        Student::new("Andrei",10,8),
        Student::new("Dragos",8,10),
        Student::new("Bogdan",7,7),
        Student::new("Clara",9,10)
    ];
    v.sort();
}
```

Error

```
error[E0277]: the trait bound `Student: Ord` is not satisfied
   --> src\main.rs:23:7
    |
23  |     v.sort();
    |       ^^^^ the trait `Ord` is not implemented for `Student`
    |
note: required by a bound in `slice::<impl [T]>::sort`
    |
275 |         T: Ord,
    |            ^^^ required by this bound in `slice::<impl [T]>::sort`
```

A sort implies the ability to compare two elements. Right now, there is no such method that describes how to compare two Students, and as such, sorting can not be done. The solution is to implement several traits, called Ord , PartialOrd, Eq and PartialEq to Student

Sorting

Let’s see what implementing this traits means.

Rust

```rust
use core::cmp::Ordering;

#[derive(Debug)]
struct Student { math: u8, english: u8, name: String }

impl Ord for Student {
    fn cmp(&self, other:&Self) -> Ordering {
        return self.name.cmp(&other.name);
    }
}
impl PartialOrd for Student {
    fn partial_cmp(&self, other:&Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}
impl PartialEq for Student {
    fn eq(&self, other: &Self) -> bool {
        self.cmp(other) == Ordering::Equal
    }
}
impl Eq for Student {}
```

Compare function (based on the name)

We will discuss more about this traits and how to use them with structs in another course.

Sorting

After we implement these traits, the code compile.

Rust

```rust
fn main() {
    let mut v = vec![
        Student::new("Andrei",10,8),
        Student::new("Dragos",8,10),
        Student::new("Bogdan",7,7),
        Student::new("Clara",9,10)
    ];
    v.sort();
    for i in v {
        println!("{:?}",i)
    }
}
```

Output

```
Student { math: 10, english: 8, name: "Andrei" }
Student { math: 7, english: 7, name: "Bogdan" }
Student { math: 9, english: 10, name: "Clara" }
Student { math: 8, english: 10, name: "Dragos" }
```

Sorting

But what if we don’t want to implement all of these traits for a simple sort ?

Rust

```rust
#[derive(Debug)]
struct Student { math: u8, english: u8, name: String }
impl Student {
    fn new(studentName: &str, mathGrade: u8, englishGrad: u8) -> Student { … }
}

fn main() {
    let mut v = vec![
        Student::new("Andrei",10,8),
        Student::new("Dragos",8,10),
        Student::new("Bogdan",7,7),
        Student::new("Clara",9,10)
    ];
    v.sort_by_key(|i| i.math);
    for i in v {
        println!("{:?}",i)
    }
}
```

Output

```
Student { math: 7, english: 7, name: "Bogdan" }
Student { math: 8, english: 10, name: "Dragos" }
Student { math: 9, english: 10, name: "Clara" }
Student { math: 10, english: 8, name: "Andrei" }
```

In this cases, using **sort_by_key** combined with a lambda function is ideal, especially if we want to sort based on a field from the structure

Sorting

Other methods related to sort methods:

Method (Vector)	Usage
`fn dedup(&mut self)`	Removes all consecutive elements that are equals.
`fn dedup_by<F>(&mut self, mut same_bucket: F)`	Removes all consecutive elements that belong to the same bucket (based on a function that determines if an element is part of a bucket or not).
`pub fn dedup_by_key<F, K>(&mut self, mut key: F)`	Removes all consecutive elements that have the same key.

All of these methods imply that the element in the vector is comparable (has the **PartialEq** trait).

Dedup methods are in particular useful when used after a sort command.

Sorting

Dedup methods (Vectors):

Rust

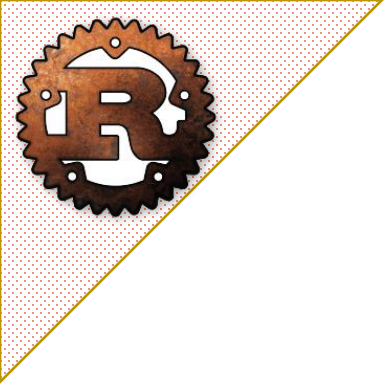
```rust
fn main() {
    let mut v = vec![1,2,7,2,5,1,2,5,7];
    v.sort();
    println!("Sorted vector: {:?}",v);
    v.dedup();
    println!("Deduped vector: {:?}",v);
}
```

Output

```
Sorted vector: [1, 1, 2, 2, 2, 5, 5, 7, 7]
Deduped vector: [1, 2, 5, 7]
```

In this case, the following buckets were reduced:

- [1, 1, 2, 2, 2, 5, 5, 7, 7] ➔ [1, 2, 5, 7]
- [1, 1, 2, 2, 2, 5, 5, 7, 7] ➔ [1, 2, 5, 7]
- [1, 1, 2, 2, 2, 5, 5, 7, 7] ➔ [1, 2, 5, 7]
- [1, 1, 2, 2, 2, 5, 5, 7, 7] ➔ [1, 2, 5, 7]



Sorting

Dedup methods (Vectors):

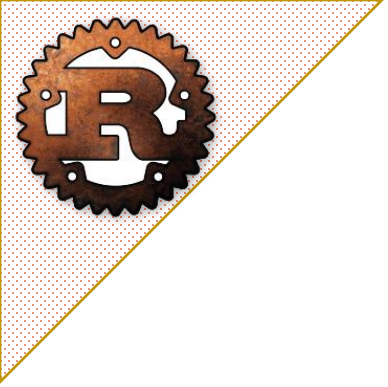
Rust

```rust
fn sort_modulo_3(value:&i32)->i32 {
    (*value) % 3
}
fn dedup_modulo_3(value:&mut i32)->i32 {
    (*value) % 3
}
fn main() {
    let mut v = vec![1,2,7,2,5,1,2,5,7];
    v.sort_by_key(sort_modulo_3);
    println!("Sorted vector: {:?}",v);
    v.dedup_by_key(dedup_modulo_3);
    println!("Deduped vector: {:?}",v);
}
```

Output

```
Sorted vector: [1, 7, 1, 7, 2, 2, 5, 2, 5]
Deduped vector: [1, 2]
```

Vector	1	7	1	7	2	2	5	2	6
Modulo 3	1	1	1	1	2	2	2	2	2



Sorting

Dedup methods (Vectors):

Rust

```
fn main() {
    let mut v = vec![1,2,7,2,5,1,2,5,7];
    v.sort_by_key(|i| (*i) % 3);
        println!("Sorted vector: {:?}",v);
    v.dedup_by_key(|i| (*i) % 3);
    println!("Deduped vector: {:?}",v);
}
```

Output

```
Sorted vector: [1, 7, 1, 7, 2, 2, 5, 2, 5]
Deduped vector: [1, 2]
```

Vector	1	7	1	7	2	2	5	2	6
Modulo 3	1	1	1	1	2	2	2	2	2

The same result can also be obtained via usage of lambda functions/closures (like in the previous example).

We will talk more about closures in another course.

Sorting

Another interesting method (related to a sorted sequence of elements) is the ability to use a binary search to quickly find an item (in $O(\log_{(n)})$ complexity). This methods can be used for Vectors, Arrays or slices.

Methods
`fn binary_search(&self, x: &T) -> Result<usize, usize>`
`fn binary_search_by<F>(&self, mut f: F) -> Result<usize, usize>`
`fn binary_search_by_key<B, F>(&self, b: &B, mut f: F) -> Result<usize, usize>`

All of these methods should be used together with the similar sort functions (e.g. use a ***binary_search_by*** with the ***sort_by*** or ***sort_unstable_by***) and with the same function as parameter.

Sorting

Let’s see some examples on how to use binary search.

Rust

```rust
fn main() {
    let v = vec![1,2,3,4,5,6,7,8];
    println!("{:?}",v.binary_search(&4));
    println!("{:?}",v.binary_search(&400));
    println!("{:?}",v.binary_search(&0));
}
```

Output

```
Ok(3)
Err(8)
Err(0)
```

The binary search function returns:

- Ok (with the value the index where the exact match was found)
- Err (with the value of the closest index to the value that was searched).

***Obs**: Notice that &4, &400 or &0. In Rust, a constant value is not implicitly converted into a constant reference like in C++ (you have to explicitly say you want to do this).*

Sorting

Let’s see some examples on how to use binary search.

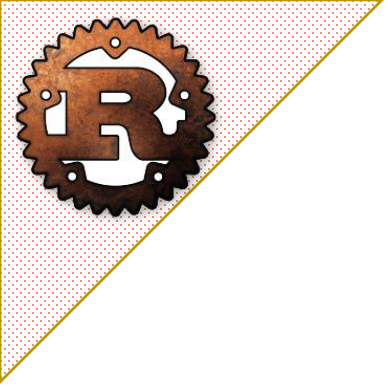
Rust

```
fn main() {
    let mut a = [1,2,3,4,5,6,7,8];
    a.sort_by_key(|i| (*i) % 3);
    println!("{:?}",a);
    println!("{:?}",a.binary_search_by_key(&0,|i| (*i) % 3));
    println!("{:?}",a.binary_search_by_key(&1,|i| (*i) % 3));
    println!("{:?}",a.binary_search_by_key(&2,|i| (*i) % 3));
    println!("{:?}",a.binary_search_by_key(&3,|i| (*i) % 3));
}
```

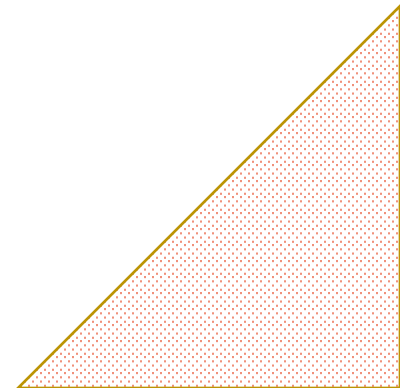
Output

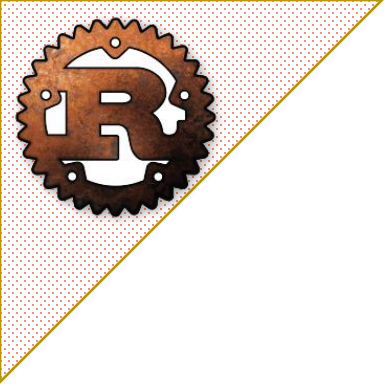
```
[3, 6, 1, 4, 7, 2, 5, 8]
Ok(1)
Ok(4)
Ok(6)
Err(8)
```

Keep in mind that **binary_search_by_key** receives for the first parameter a key and not a value. In this case, possible keys are 0,1 and 2 (everything that module 3 can obtained). That is why, the first 3 searches will end up with Ok, while search no 4 (for the value 3) will return Err as any value module 3 will never result in 3!!!



Hash maps





Hash maps

Hash maps collection of elements, where every element can quickly (ideally $O_{(1)}$) be access via a key. There are several implementation possible for a Hash maps; Rust has a specific object called HashMap (like **std::unordered_map**) from C++ standard.

Rust implementation (a variation of https://abseil.io/blog/20180927-swisstables)

To create a map, use one of the following forms:

a) `let mut a: HashMap<key,value> = HashMap::new()`
b) `let mut a = HashMap::<key,value>::new()`
c) `let mut a: HashMap<key,value> = HashMap::with_capacity(capacity)`
d) `let mut a: HashMap<key,value> = HashMap::from([(key,vector);count])`

Hash maps

Let’s see some examples on how to build a hash map.

Rust

```rust
use std::collections::HashMap;

fn main() {
    let m1 = HashMap::<i32,i32>::new();
    let m2 = HashMap::<&str,i32>::with_capacity(100);
    let m3 = HashMap::from([
        ("John",10), ("Mike",20), ("George",30)
    ]);
    println!("{:?}, Capacity={}, items={}",m1,m1.capacity(),m1.len());
    println!("{:?}, Capacity={}, items={}",m2,m3.capacity(),m2.len());
    println!("{:?}, Capacity={}, items={}",m3,m3.capacity(),m3.len());
}
```

Output

```
{}, Capacity=0, items=0
{}, Capacity=3, items=0
{"John": 10, "Mike": 20, "George": 30}, Capacity=3, items=3
```

***Obs**: Notice the use of **std::collections::HashMap** (this is required to use a hash map).*

Hash maps

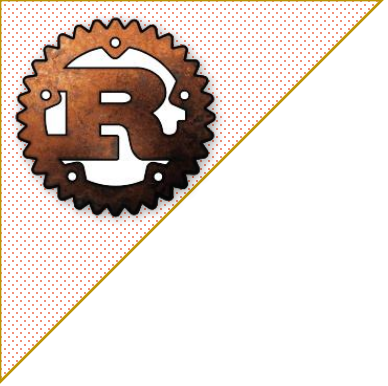
It is important to notice that the hashing algorithm use rely on a randomized seed. This mean that consecutive execution of the same code will result in a different order of elements in memory.

Rust

```rust
fn main() {
    for _ in 0..7 {
        let m3 = HashMap::from([("John", 10), ("Mike", 20), ("George", 30)]);
        println!("{:?}, Capacity={}, items={}", m3, m3.capacity(), m3.len());
    }
}
```

Output (possible).

```
{"Mike": 20, "George": 30, "John": 10}, Capacity=3, items=3
{"George": 30, "John": 10, "Mike": 20}, Capacity=3, items=3
{"John": 10, "George": 30, "Mike": 20}, Capacity=3, items=3
{"George": 30, "Mike": 20, "John": 10}, Capacity=3, items=3
{"George": 30, "Mike": 20, "John": 10}, Capacity=3, items=3
{"John": 10, "Mike": 20, "George": 30}, Capacity=3, items=3
{"George": 30, "John": 10, "Mike": 20}, Capacity=3, items=3
```



Hash maps

Basic operations for hash maps

Method	Usage
`fn insert(&mut self, k: K, v: V) -> Option<V>`	Inserts a key/value pair into the hash map. If the key exists in the map, None is return. If the key exists, it is updated, and the old value is returned.
`fn get (&self, k: &K) -> Option<&V>` `fn get_mut (&mut self, k: &K) -> Option<&mut V>`	Get a reference to the value associated with a key
`fn contains_key (&self, k: &K) -> bool`	True if a key exists in the hash map
`fn remove (&mut self, k: &Q) -> Option<V>`	Removes an element from the map an returns its value.
`fn clear(&mut self)`	Clears all key/value pairs (but keeps the allocated memory for future usage).

Hash maps

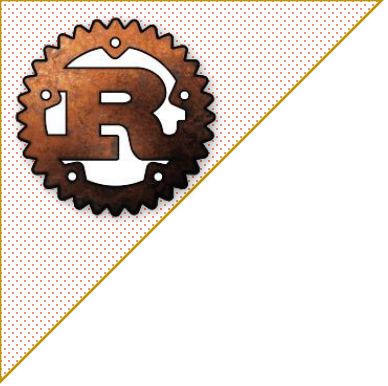
Let’s see an example:

Rust

```rust
fn main() {
    let mut m = HashMap::from([
        ("John",10), ("Mike",20), ("George",30)
    ]);
    println!("John is in m: {}",m.contains_key("John"));
    m.insert("Vincent", 20);
    println!("{:?}",m);
    println!("Value for 'Mike' is : {:?}",m.get("Mike"));
    *m.get_mut("George").unwrap() = 50;
    println!("{:?}",m);
}
```

Output

```
John is in m: true
{"John": 10, "Vincent": 20, "George": 30, "Mike": 20}
Value for 'Mike' is : Some(20)
{"John": 10, "Vincent": 20, "George": 50, "Mike": 20}
```



Hash maps

One of the most used method for hash maps is .entry():

Method	Usage
`fn entry(&mut self, key: K) -> Entry<K, V>`	Returns a structure (Entry) that can be used to modify the value of a key.

Entry struct has the following methods:

Method	Usage
`fn and_modify(self, f: Function) -> Self`	Change the value associated with a specific key with a value returned from a function F.
`fn or_insert(self, default: V) -> &mut V`	Returns a mutable reference to a value of a specific key. If that key is not present, it will be inserted and set up with a default value, and then the reference to that default value will be returned.
`fn or_insert_with(self, default: F) -> & mut V` `fn or_insert_with_key-> V>(self, default: F) -> &mut V`	Similar to ***or_insert***, but uses a function **default** to return a value

Hash maps

Let’s see some examples on how to use .entry() method:

Rust

```rust
fn main() {
    let mut m = HashMap::from([("John",10), ("Mike",2
    m.entry("John").and_modify(|x| { *x = *x + 10; });
    println!("{m:?}");
    m.entry("John2").and_modify(|x| { *x = *x + 10; });
    println!("{m:?}");
}
```

Output

```
{"George": 30, "Mike": 20, "John": 20}
{"George": 30, "Mike": 20, "John": 20}
```

Notice that if key is not present than .and_modify(..) has no effect !

Hash maps

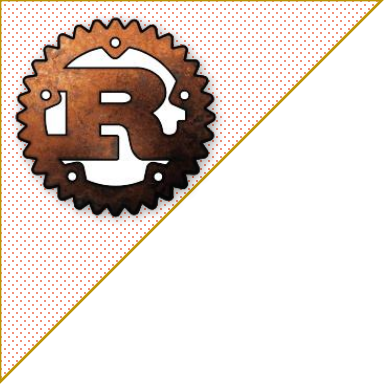
Let’s see some examples on how to use .entry() method:

Rust

```rust
fn main() {
    let mut m = HashMap::from([("John",10), ("Mike",20), ("George",30)]);
    let john_value = m.entry("John").or_insert(200);
    println!("john = {john_value}");
    let alice_value = m.entry("Alice").or_insert(200);
    println!("alice = {alice_value}");
    println!("{:?}",m);
}
```

Output

```
john = 10
alice = 200
{"John": 10, "George": 30, "Alice": 200, "Mike": 20}
```



Hash maps

Keep in mind that the value returned by .or_insert(…) method is a mutable reference. This means that for example you can not use the hashmap as an immutable reference while that reference still exists.

Rust

```rust
fn main() {
    let mut m = HashMap::from([("John",10), ("Mike",20), ("George",30)]);
    let john_value = m.entry("John").or_insert(200);
    println!("{:?}",m);
    println!("john = {john_value}");
}
```

Error

```
error[E0502]: cannot borrow `m` as immutable because it is also borrowed as
mutable
 --> src\main.rs:6:21
  |
5 |     let john_value = m.entry("John").or_insert(200);
  |                      --------------- mutable borrow occurs here
6 |     println!("{:?}",m);
  |                     ^ immutable borrow occurs here
7 |     println!("john = {john_value}");
  |                       ---------- mutable borrow later used here
```

Hash maps

Keep in mind that the value returned by .or_insert(…) method is a mutable reference. This means that for example you can not use the hashmap as an immutable reference while that reference still exists.

Rust

```rust
fn main() {
    let mut m = HashMap::from([("John",10), ("Mike",20), ("George",30)]);
    let john_value = m.entry("John").or_insert(200);
    println!("john = {john_value}");
    println!("{:?}",m);
}
```

Output

```
john = 10
{"George": 30, "Mike": 20, "John": 10}
```

In this case we have reversed the order of calls (first we print ***john_value*** and then ***m***). This will work as the lifetime of ***john_value*** ends after println! Macro.

Hash maps

Let’s see some examples on how to use .entry() method:

Rust

```rust
fn main() {
    let mut m = HashMap::from([("John",10), ("Mike",20), ("George",30)]);
    m.entry("John").or_insert_with(|| 100);
    println!("{m:?}");
    m.entry("Alice").or_insert_with(|| 100);
    println!("{m:?}");
    m.entry("Liam").or_insert_with_key(|key| key.len());
    println!("{m:?}");
}
```

Output

```
{"John": 10, "Mike": 20, "George": 30}
{"John": 10, "George": 30, "Alice": 100, "Mike": 20}
{"John": 10, "George": 30, "Alice": 100, "Mike": 20, "Liam": 4}
```

***OBS**: .or_insert_with_key(…) uses a function that receives the key name and returns a value (in our case “Liam” has a size of 4 chars).*

Hash maps

One of the most common usage for .entry() is to count elements from a vector / array. The solution is to use .entry(…).or_insert(…) to first insert and initialize a string in the map, and then increment the value.

Rust

```rust
fn main() {
    let mut m = HashMap::new();
    let v = ["John","Alice","John","Mike","Alice","John","John"];
    for k in v {
        *m.entry(k).or_insert(0)+=1;
    }
    println!("{m:?}");
}
```

Output

```
{"John": 4, "Alice": 2, "Mike": 1}
```

Hash maps

Hash maps are iterable:

Rust

```rust
fn main() {
    let mut m = HashMap::from([("John",10), ("Mike",20), ("George",30)]);
    for i in m {
        println!("{:?}",i);
    }
}
```

Output

```
("George", 30)
("John", 10)
("Mike", 20)
```

You can also use .keys() to enumerate directly through keys:

Rust

```rust
fn main() {
    let mut m = HashMap::from([("John",10), ("Mike",20), ("George",30)]);
    for i in m.keys() {
        println!("{:?}",i);
    }
}
```

Output

```
"John"
"Mike"
"George"
```

Hash maps

Keep in mind that iterating through an object moves the key/value pair instead of returning a reference:

Rust

```
fn main() {
    let m = HashMap::from([
        (String::from("key-1"),String::from("John")),
        (String::from("key-2"),String::from("Mike")),
        (String::from("key-3"),String::from("Marry")),
    ]);
    for i in m {
        println!("{:?}",i);
    }
    println!("{:?}",m);
}
```

Error

```
error[E0382]: borrow of moved value: `m`
   --> src\main.rs:12:21
    |
4   |     let m = HashMap::from([
    |         - move occurs because `m` has type `HashMap<String, String>`,
    |           which does not implement the `Copy` trait
...
9   |     for i in m {
    |              - `m` moved due to this implicit call to `.into_iter()`
...
12  |     println!("{:?}",m);
    |                     ^ value borrowed here after move
```

Hash maps

The solution is to iterate over a reference of that object instead of the object.

Rust

```rust
fn main() {
    let m = HashMap::from([
        (String::from("key-1"),String::from("John")),
        (String::from("key-2"),String::from("Mike")),
        (String::from("key-3"),String::from("Marry")),
    ]);
    for i in &m {
        println!("{:?}",i); // "i" is of type (&String,&string)
    }
    println!("{:?}",m);
}
```

Output

```
("key-1", "John")
("key-2", "Mike")
("key-3", "Marry")
{"key-1": "John", "key-2": "Mike", "key-3": "Marry"}
```

Hash maps

To get the map capacity and length use .len() and .capacity() methods

Rust

```rust
fn main() {
    let m = HashMap::from([
        (String::from("key-1"),String::from("John")),
        (String::from("key-2"),String::from("Mike")),
        (String::from("key-3"),String::from("Marry")),
        (String::from("key-4"),String::from("Andy")),
        (String::from("key-5"),String::from("Andrei")),
        (String::from("key-6"),String::from("Dragos")),
        (String::from("key-7"),String::from("Carlos")),
        (String::from("key-8"),String::from("Terry")),
        (String::from("key-9"),String::from("Ana")),
    ]);
    println!("Capacity={}, len={}",m.capacity(), m.len());
}
```

Output

```
Capacity=14, len=9
```

Hash maps

Use .remove(key) to remove a key from the hash map.

Rust

```rust
fn main() {
    let mut m= HashMap::from([
        (String::from("John"),10),
        (String::from("Mike"),8),
        (String::from("Marry"),4),
        (String::from("Andy"),9),
        (String::from("Andrei"),5),
    ]);
    println!("Remove Mike -> with value: {:?}",m.remove("Mike"));
    println!("Remove Dragos -> with value: {:?}",m.remove("Dragos"));
    println!("Hashmap = {:?}",m);
}
```

Output

```
Remove Mike -> with value: Some(8)
Remove Dragos -> with value: None
Hashmap = {"Marry": 4, "Andy": 9, "John": 10, "Andrei": 5}
```

Hash maps

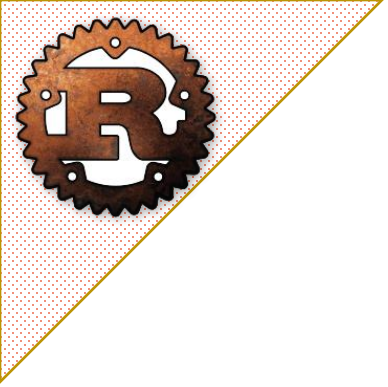
You can also use .retain(predicate) to keep in the map only the elements that match a specific criteria.

Rust

```rust
fn bigger_than_8(key: &String,value: &mut i32)->bool {
    (*value)>8
}
fn main() {
    let mut m= HashMap::from([
        (String::from("John"),10),
        (String::from("Mike"),8),
        (String::from("Marry"),4),
        (String::from("Andy"),9),
        (String::from("Andrei"),5),
    ]);
    m.retain(bigger_than_8);
    println!("{:?}",m);
}
```

Output

```
{"John": 10, "Andy": 9}
```



Hash maps

You can also use .retain(predicate) to keep in the map only the elements that match a specific criteria.

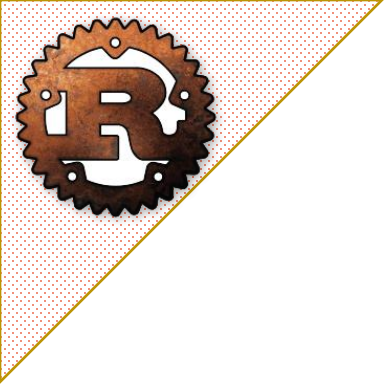
Rust

```rust
fn main() {
    let mut m= HashMap::from([
        (String::from("John"),10),
        (String::from("Mike"),8),
        (String::from("Marry"),4),
        (String::from("Andy"),9),
        (String::from("Andrei"),5),
    ]);
    m.retain(|k,v| *v>8);
    println!("{:?}",m);
}
```

Output

```
{"John": 10, "Andy": 9}
```

The same result can also be obtained via a closure/lambda function.



HashSet

Hash sets

HashSet type in Rust is implemented over a HashMap with a value of type () → a ZST type thus making sure that there is no extra memory allocated for values.

To create a set, use one of the following forms:

a) `let mut a: HashSet<type> = HashSet::new()`
b) `let mut a = HashSet::<type>::new()`
c) `let mut a: HashSet<type> = HashSet::with_capacity(capacity)`
d) `let mut a: HashSet<type> = HashSet::from([<type>;count])`

Hash sets

Let’s see some examples on how to build a hash set.

Rust

```rust
use std::collections::HashSet;

fn main() {
    let s1 = HashSet::from([1,2,3,4,5]);
    let s2 = HashSet::<i32>::new();
    let s3 = HashSet::from([1,1,2,2,3,3,3,4,5]);
    println!("s1 = {:?}",s1);
    println!("s2 = {:?}",s2);
    println!("s3 = {:?}",s3);
}
```

Output

```
s1 = {2, 1, 4, 3, 5}
s2 = {}
s3 = {1, 5, 2, 4, 3}
```

***Obs**: Notice the use of **std::collections::HashSet** (this is required to use a hash set).*

Keep in mind that the order of the elements is not guarantee to be the insertion order. Also … using several elements with the same value, will strip down equal elements

Hash sets

Basic operations for hash sets

Method	Usage
`fn insert(&mut self, value: T) -> bool`	Inserts a value in a set. Returns false if the value already exists in the set, true otherwise.
`fn get (&self, v: &T) -> Option<&T>`	Get a reference to a value if exists in the set
`fn contains (&self, value: &T) -> bool`	True if a value exists in the set
`fn remove (&mut self, value: &T) -> bool`	If value exists in the set, removes it and return true. Otherwise returns false.
`fn clear(&mut self)`	Removes all elements from the set (but keeps the allocated memory for future usage).

Hash sets

Let’s see some examples on how to use a set:

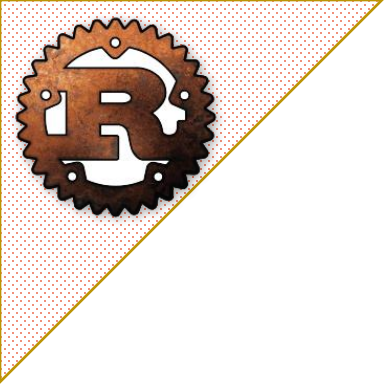
Rust

```rust
use std::collections::HashSet;

fn main() {
    let mut s = HashSet::from([1,2,3,4,5]);
    println!("{:?}",s);
    println!("Add 3 -> {} => s = {:?}",s.insert(3),s);
    println!("Add 7 -> {} => s = {:?}",s.insert(7),s);
    println!("Remove 1 -> {} => s = {:?}",s.remove(&1),s);
    println!("Remove 9 -> {} => s = {:?}",s.remove(&9),s);
    println!("Is 4 in the set -> {}",s.contains(&7));
    println!("Get 5 from set -> {:?}",s.get(&5));
    println!("Get 8 from set -> {:?}",s.get(&8));
}
```

Output

```
{1, 2, 4, 5, 3}
Add 3 -> false => s = {1, 2, 4, 5, 3}
Add 7 -> true => s = {1, 2, 4, 7, 5, 3}
Remove 1 -> true => s = {2, 4, 7, 5, 3}
Remove 9 -> false => s = {2, 4, 7, 5, 3}
Is 4 in the set -> true
Get 5 from set -> Some(5)
Get 8 from set -> None
```



Hash sets

There are however, some methods specific to sets:

Method
`fn intersection(&self, other: &HashSet<T>) -> Intersection<T>`
`fn union(&self, other: &HashSet<T>) -> Union<T>`
`fn symmetric_difference(&self, other: &HashSet<T>) -> SymmetricDifference<T>`
`fn difference(&self, other: &HashSet<T>) -> Difference<T>`
`fn is_disjoint(&self, other: &HashSet<T>) -> bool`
`fn is_subset(&self, other: &HashSet<T>) -> bool`
`fn is_superset(&self, other: &HashSet<T>) -> bool`

Methods for union, intersection, difference and symmetric difference return an iterator.

Hash sets

Let’s see some examples on how set specific methods:

Rust

```
fn main() {
    let s1 = HashSet::from([1,2,3,4,5]);
    let s2 = HashSet::from([3,4,5,6,7]);
    let u:HashSet<_> = s1.union(&s2).collect();
    let i:HashSet<_> = s1.intersection(&s2).collect();
    let sd:HashSet<_> = s1.symmetric_difference(&s2).collect();
    let d1:HashSet<_> = s1.difference(&s2).collect();
    let d2:HashSet<_> = s2.difference(&s1).collect();
    println!("Union = {:?}",u);
    println!("Intersection = {:?}",i);
    println!("Sym.diff = {:?}",sd);
    println!("s1-s2={:?}   s2-s1={:?}",d1,d2);
}
```

Output

```
Union = {4, 3, 6, 7, 5, 1, 2}
Intersection = {5, 4, 3}
Sym.diff = {7, 6, 1, 2}
s1-s2={2, 1}   s2-s1={6, 7}
```

Btree Map

Btree Map

A Btree map is an ordered map based on a binary tree algorithm (more on binary trees : https://en.wikipedia.org/wiki/B-tree). The closest equivalence from C++ space is std::map (even though they use different algorithms under the hood).

Keep in mind that current Rust implementation is slightly different than the classical one as it tries to optimize search for small sets of data and use as much of the processor cache as possible.

To create a b-tree map, use one of the following forms:

a) `let mut a: BTreeMap<key,value> = BTreeMap ::new()`
b) `let mut a = BTreeMap::<key,value>::new()`
c) `let mut a: BTreeMap<key,value> = BTreeMap ::with_capacity(capacity)`
d) `let mut a: BTreeMap <key,value> = BTreeMap ::from([(key,vector);count])`

Btree Map

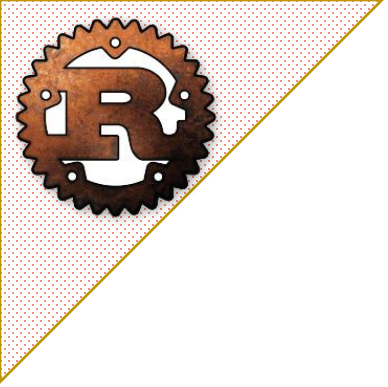
Let’s see some examples:

Rust

```rust
fn main() {
    for _i in 0..3 {
        let m = BTreeMap::from([("John",10),("Ana",20),("Mike",5),("Bugsy",10)]);
        println!("{:?}",m);
    }
    println!("---------------------------------");
    for _i in 0..3 {
        let m = HashMap::from([("John",10),("Ana",20),("Mike",5),("Bugsy",10)]);
        println!("{:?}",m);
    }
}
```

Output

```
{"Ana": 20, "Bugsy": 10, "John": 10, "Mike": 5}
{"Ana": 20, "Bugsy": 10, "John": 10, "Mike": 5}
{"Ana": 20, "Bugsy": 10, "John": 10, "Mike": 5}
---------------------------------
{"Mike": 5, "John": 10, "Bugsy": 10, "Ana": 20}
{"Bugsy": 10, "Ana": 20, "Mike": 5, "John": 10}
{"Mike": 5, "John": 10, "Ana": 20, "Bugsy": 10}
```



Btree Map

Let’s see some examples on how set specific methods:

Rust

```rust
fn main() {
    for _i in 0..3 {
        let m = BTreeMap::from([("John",10),("Ana",20),("Mike",5),("Bugsy",10)]);
        println!("{:?}",m);
    }
    println!("---------------------------------");
    for _i in 0..3 {
        let m = HashMap::from([("John",10),("Ana",20),("Mike",5),("Bugsy",10)]);
        println!("{:?}",m);
    }
```

Output

```
{"Ana": 20, "Bugsy": 10, "John": 10, "Mike": 5}
{"Ana": 20, "Bugsy": 10, "John": 10, "Mike": 5}
{"Ana": 20, "Bugsy": 10, "John": 10, "Mike": 5}
---------------------------------
{"Mike": 5, "John": 10, "Bugsy": 10, "Ana": 20}
{"Bugsy": 10, "Ana": 20, "Mike": 5, "John": 10}
{"Mike": 5, "John": 10, "Ana": 20, "Bugsy": 10}
```

Notice that BTreeMap sorts all elements based on their key. This is different than what HashMap is doing (HashMap uses a random seed and as such the order is different almost every time).

Btree Map

Btree map has the same methods as Hash maps (e.g. insert, get, entry, contains, etc). However, since the keys in a btree map are sorted, there are other methods available only on BTreeMap objects:

Method	Usage
`fn append(&mut self, other: &mut Self)`	Appends all elements from a specific Btree map into another one.
`fn pop_first(&mut self) -> Option<(K, V)>`	These 4 methods are experimental and are considered **unstable**. While in the future it is possible for these methods to be available, right now they are not part of the stable SDK.
`fn pop_last(&mut self) -> Option<(K, V)>`	
`fn first_entry(&mut self) -> Option<OccupiedEntry<K,V>>`	
`fn last_entry(&mut self) -> Option<OccupiedEntry<K,V>>`	

Btree Map

Let’s see an example that uses .append(…) method:

Rust

```rust
fn main() {
    let mut m1 = BTreeMap::from([("John",10),("Ana",20),("Mike",5),("Bugsy",10)]);
    let mut m2 = BTreeMap::from([("Andra",10),("Ana",10),("Loyd",15),("Erik",12)]);
    m1.append(&mut m2);
    println!("{:?}",m1);
}
```

Output

```
{"Ana": 10, "Andra": 10, "Bugsy": 10, "Erik": 12, "John": 10, "Loyd": 15, "Mike": 5}
```

Notice that if a key already exists, its value is updated with after the append method is called (key “Ana” had initially value 20, after update it has a value of 10).

Btree Map

Btree map is a well-suited choice for problems where a priority queue is required. The most common usage in this case is by using iterators and their method .next() to advance to the next element. The result is that you can extract / iterate over each elements in their order.

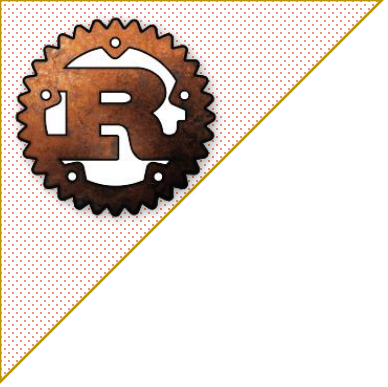
Rust

```rust
fn main() {
    let m = BTreeMap::from([("John",10),("Ana",20),("Mike",5),("Bugsy",10)]);
    let mut i = m.iter();
    while let Some(x) = i.next() {
        println!("Extract {} width value: {}",x.0,x.1);
    }
}
```

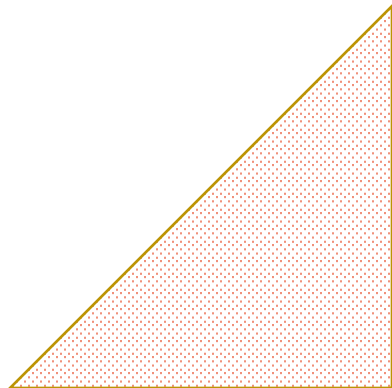
Output

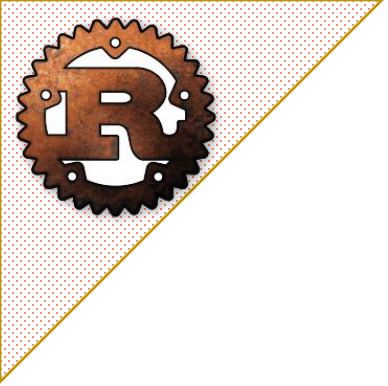
```
Extract Ana width value: 20
Extract Bugsy width value: 10
Extract John width value: 10
Extract Mike width value: 5
```

OBS: *We will discuss more about iterators in another course.*



BTreeSet





BTreeSet

BTreeSet type in Rust is implemented over a BTreeMap with a value of type () → a ZST type thus making sure that there is no extra memory allocated for values.

To create a set, use one of the following forms:

a) `let mut a: BTreeSet<type> = BTreeSet::new()`
b) `let mut a = BTreeSet::<type>::new()`
c) `let mut a: BTreeSet<type> = BTreeSet::with_capacity(capacity)`
d) `let mut a: BTreeSet<type> = BTreeSet::from([<type>;count])`

The logic and methods are similar to the ones from HashSet.

The similar class from C++ is std::set

Btree Set

Let’s see an example on how to use a BTreeSet:

Rust

```rust
fn main() {
    let s = BTreeSet::from([10,2,7,4,9,11,3,6,7]);
    println!("{s:?}");
}
```

Output

{2, 3, 4, 6, 7, 9, 10, 11}

Similar to BTreeMap, there is an .append(…) method:

Rust

```rust
fn main() {
    let mut s1 = BTreeSet::from([10,2,7,4,9,11,3,6,7]);
    let mut s2 = BTreeSet::from([1,8,3,6,5]);
    s1.append(&mut s2);
    println!("{s1:?}");
}
```

Output

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

Map comparation between C++ and Rust

C++ vs Rust (on maps)

Let’s compare how various types of maps work on Rust and C++.

For this we will use:

- std::map (C++)
- std::unordered_map (C++)
- HashMap (Rust)
- BTreeMap (Rust)

The same algorithm will be written in both Rust and C++ and tested in Debug and Release mode. We will use GetTickCount API to measure time. Each variation of the build will be executed for 10 times and the average will be compute.

C++ vs Rust (on maps)

So … lets see the testing algorithm:

Rust

```rust
extern "system" { fn GetTickCount64() -> u64; }
fn get_time() -> u64 { unsafe { GetTickCount64() } }

use std::collections::{BTreeMap, HashMap};

#[derive(Debug, Copy, Clone)]
struct Test { v1: u64, v2: f32, v3: bool }

fn main() {
    let mut m: HashMap<u32, Test> = HashMap::new();
    let start = get_time();
    for i in 0..1_000_000 {
        let t = Test { v1: i as u64, v2: 1.5,v3: i%2==0};
        m.insert(i, t);
    }
    let end = get_time();
    println!("{}", end - start);
}
```

C++

```cpp
#include <Windows.h>
#include <map>
#include <unordered_map>
struct Test {
    unsigned long long v1;
    float v2;
    bool v3;
};
void main() {
    std::unordered_map<unsigned int, Test> m;
    auto start = GetTickCount64();
    for (auto i = 0; i < 1000000; i++) {
        m[i] = Test{ (unsigned long long)i,
                     1.5,i % 2 == 0 };
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```

C++ vs Rust (on maps)

So … lets see the testing algorithm:

Rust

```rust
extern "system" { fn GetTickCount64() -> u64; }
fn get_time() -> u64 { unsafe { GetTickCount64() } }

use s

#[der
struc

fn main() {
    let mut m: HashMap<u32, Test> = HashMap::new();
    let start = get_time();
    for i in 0..1_000_000 {
        let t = Test { v1: i as u64, v2: 1.5,v3: i%2==0};
        m.insert(i, t);
    }
    let end = get_time();
    println!("{}", end - start);
}
```

We will run the same algorithm using:

- HashMap
- BTreeMap

C++

```cpp
#include <Windows.h>
#include <map>
#include <unordered_map>
struc
    u
    f
    b
};
void main() {
    std::unordered_map<unsigned int, Test> m;
    auto start = GetTickCount64();
    for (auto i = 0; i < 1000000; i++) {
        m[i] = Test{ (unsigned long long)i,
                     1.5,i % 2 == 0 };
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```

We will run the same algorithm using:

- std::unordered_map
- std::map

C++ vs Rust (on maps)

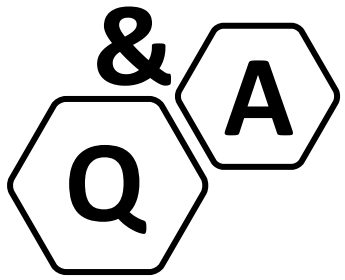
So … lets see the testing algorithm:

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Average
C++ (Debug) std:unordered_map	938	1266	1390	1328	1250	1297	1250	1344	1485	1437	**1298**
C++ (Release) std:unordered_map	297	234	235	281	266	266	234	235	234	234	**251**
C++ (Debug) std::map	1312	1765	1953	1875	1875	1859	1813	1812	1797	1828	**1788**
C++ (Release) std::map	156	141	172	157	141	171	172	156	172	156	**159**
Rust (Debug) HashMap	1141	1297	1265	1250	1281	1312	1359	1297	1343	1297	**1284**
Rust (Release) HashMap	78	78	63	78	78	94	93	94	94	94	**84**
Rust (Debug) BTreeMap	2703	3156	3078	2906	2765	2875	2937	2844	2860	2781	**2890**
Rust (Release) BTreeMap	93	93	109	110	125	110	125	141	125	125	**115**

C++ vs Rust (on maps)

The general conclusion after these tests is:

- Rust is slower the C++ when it comes to debug mode (due to many checks)
- In terms of Release mode, Rust is faster (however, it should be noted that we are not comparing the same algorithms and as such these tests might NOT be correct). However, since we've compared the standard algorithms from each (Rust and C++) libraries, the results are however relevant.
- The tests were performed on Windows 11 (using Microsoft compiler). To produce accurate results, other C++ compilers (such as clang and gcc) should be tested as well.


&
Q
A