Practical Programming

Rust : Arrays and Vectors

David Bouchet

david.bouchet.epita@gmail.com

Arrays

- Every element has the same type.
- The length is fixed.
- Data is allocated on the stack.
- Elements cannot be added or removed.
- If mutable, elements can change.

Creating Arrays

```rust
let a = [1, 2, 3, 4];
let b = [true, false];
let c = ["Hello", "World", "!"];
let d = [13; 5];

dbg!(a);
dbg!(b);
dbg!(c);
dbg!(d);
```

```
a = [
    1,
    2,
    3,
    4
]
b = [
    true,
    false
]
c = [
    "Hello",
    "World",
    "!"
]
```

```
d = [
    13,
    13,
    13,
    13,
    13
]
```

Indexing Arrays

```
let a = [1, 2, 3, 4];
let b = [true, false];
let c = ["Hello", "World", "!"];

dbg!(a[2]);
dbg!(b[0]);
dbg!(c[1]);
```

```
a[2] = 3
b[0] = true
c[1] = "World"
```

Mutable and Immutable Arrays

Elements cannot be added or removed.

Elements can change in mutable arrays only.

```rust
let mut a = [1, 2, 3];
let b = [1, 2, 3];

a[1] = 55;        // OK
// b[1] = 55;     // ERROR

dbg!(a);
dbg!(b);
```

```
a = [
    1,
    55,
    3
]
b = [
    1,
    2,
    3
]
```

Array Types

[T; n] →

- **T**: Type of the elements
- **n**: Number of elements

```
let a = [1, 2, 3];            // [i32; 3]
let b: [u8; 3] = [1, 2, 3]; // [u8; 3]
let c = [1_u8, 2, 3];         // [u8; 3]

let x: u8 = 8;
let d = [x, 2, 3];            // [u8; 3]

let mut e = [1, 2, 3];        // [i32; 3]
e = [2, 3, 4];                // [i32; 3]
e = [5, 6, 7, 8];             // [i32; 4] -> ERROR
```

The length belongs to the type.

Vectors

- Every element has the same type.
- The length is variable.
- Data is allocated on the heap.
- If mutable, elements can be added or removed.
- If mutable, elements can change.

Creating Vectors

```rust
let a = vec![1, 2, 3, 4];
let b = vec![true, false];
let c = vec!["Hello", "World", "!"];
let d = vec![13; 5];

dbg!(a);
dbg!(b);
dbg!(c);
dbg!(d);
```

```
a = [
    1,
    2,
    3,
    4
]
b = [
    true,
    false
]
c = [
    "Hello",
    "World",
    "!"
]
```

```
d = [
    13,
    13,
    13,
    13,
    13
]
```

Indexing Vectors

```rust
let a = vec![1, 2, 3, 4];
let b = vec![true, false];
let c = vec!["Hello", "World", "!"];

dbg!(a[2]);
dbg!(b[0]);
dbg!(c[1]);
```

```
a[2] = 3
b[0] = true
c[1] = "World"
```

Mutable Vectors

Elements can be added or removed.

Elements can change

```rust
let mut v = vec!["Hello", "World", "!"];

v.insert(0, "Good");
v[1] = "bye";
v.remove(3);
v.push("!!!!!");
v.insert(2, ",");

dbg!(v);
```

```
v = [
    "Good",
    "bye",
    ",",
    "World",
    "!!!!!"
]
```

Vector Types

Vec<T> → **T**: Type of the elements

```
let a = vec![1, 2, 3];            // Vec<i32>
let b: Vec<u8> = vec![1, 2, 3]; // Vec<u8>
let c = vec![1_u8, 2, 3];         // Vec<u8>

let x: u8 = 8;
let d = vec![x, 2, 3];            // Vec<u8>

let mut e = vec![1, 2, 3];        // Vec<i32>
e = vec![2, 3, 4];                // Vec<i32>
e = vec![5, 6, 7, 8];             // Vec<i32> -> OK
```

The length does not belong to the type.

Array and Vector Lengths

```rust
let a = [0; 17];
let v = vec![0; 24];

dbg!(a.len());
dbg!(v.len());
```

```
a.len() = 17
v.len() = 24
```

Iterating over Immutable References

```rust
let a = [1, 2, 3];
let v = vec![4, 5, 6];

for i in a.iter()
{
    dbg!(i);
}

for i in v.iter()
{
    dbg!(i);
}
```

```
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
```

- The *i* variable is an **immutable reference**.
- By default, Rust prints the value that is pointed to by the reference.
- `dbg!(*i)` would give the same result.

Iterating over Mutable References

```rust
let mut a = [1, 2, 3];
let mut v = vec![4, 5, 6];

for i in a.iter_mut()
{
    *i *= 2;
}

for i in v.iter_mut()
{
    *i *= 2;
}

dbg!(a);
dbg!(v);
```

```
a = [
    2,
    4,
    6
]
v = [
    8,
    10,
    12
]
```

The *i* variable is a **mutable reference**.

Printing Arrays and Vectors

There is no default formatter for arrays and vectors.

```rust
fn main()
{
    let a: [u8; 3] = [1, 2, 3];
    println!("{}", a);
}
```

```
error[E0277]: `[u8; 3]` doesn't implement `std::fmt::Display`
 --> printing_default.rs:4:20
  |
4 |     println!("{}", a);
  |                    ^ `[u8; 3]` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `[u8; 3]`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
  = note: required by `std::fmt::Display::fmt`
```

Debug Printing

To print arrays and vectors with *println!()*, we must use the debug formatter:

```
let a = [1, 2, 3];
let v = vec![4, 5, 6];

println!("a = {:?}", a);
println!("v = {:?}", v);
```

```
a = [1, 2, 3]
v = [4, 5, 6]
```

Implicit Dereferencing

```rust
let a = [1, 2, 3];
let v = vec![4, 5, 6];

let ref_a = &a;
let ref_v = &v;

dbg!(a[1]);
dbg!(v[1]);

dbg!((*ref_a)[1]);
dbg!((*ref_v)[1]);

dbg!(ref_a[1]);
dbg!(ref_v[1]);
```

```
a[1] = 2
v[1] = 5
(*ref_a)[1] = 2
(*ref_v)[1] = 5
ref_a[1] = 2
ref_v[1] = 5
```

Slice Reference

&[T] → **T**: Type of the elements

```rust
fn average(slice: &[i32]) -> f64
{
    let mut sum = 0;

    for i in slice.iter()
    {
        sum += i;
    }

    sum as f64 / slice.len() as f64
}
```

```rust
let a = [1, 2, 3];
let v = vec![4, 5, 6];

dbg!(average(&a));
dbg!(average(&v));
```

```
average(&a) = 2.0
average(&v) = 5.0
```

Slicing

Array and vector slices can be referenced in the same way as string slices.

```
let a = [1, 2, 3, 4, 5];
let v = vec![6, 7, 8, 9, 10];

dbg!(average(&a[1..]));
dbg!(average(&v[..2]));
dbg!(average(&a[1..=3]));
dbg!(average(&v[2..4]));
```

```
average(&a[1..]) = 3.5
average(&v[..2]) = 6.5
average(&a[1..=3]) = 3.0
average(&v[2..4]) = 8.5
```

Converting Slices into Vectors

```rust
let a = [1, 2, 3];
let s = &a[1..3];
let mut v = s.to_vec();

println!("a = {:?}", a);
println!("s = {:?}", s);
println!("v = {:?}", v);

v.push(18);
println!("v = {:?}", v);
```

```
a = [1, 2, 3]
s = [2, 3]
v = [2, 3]
v = [2, 3, 18]
```

- The slice is copied into another memory space, which is associated with a vector.
- In the example, ***s*** and ***v*** are independent.

Mutable Slices

&mut [T] → **T**: Type of the elements

```
let mut a = [1, 2, 3, 4, 5, 6];

println!("a = {:?}", a);
set_to_zero(&mut a[1..=4]);
println!("a = {:?}", a);
```

```
a = [1, 2, 3, 4, 5, 6]
a = [1, 0, 0, 0, 0, 6]
```

```
fn set_to_zero(slice: &mut [i32])
{
    for i in slice.iter_mut()
    {
        *i = 0;
    }
}
```