TD-3

Advanced types: Compound and collection types

Pierre Cochard and Tanguy Risset

5TC-RUST 2025

Summary

- Compound types
  - Tuples
  - Arrays (primitive type)
  - Ranges & Iterators
- Collections (Vectors & Hash-maps)
- 'string' types (`str` and `String`)
- Slices

Compound types: Tuples

Tuples are fixed-size collections of **arbitrary-typed** values.

```
// Inferred type:
let mut tup = (31, 16, 47.27, "hello!");

// Explicit type:
let mut tup: (i32, i32, f32, &str) = (31, 16, 47.27, "hello!");
```

Accessing individual values within a Tuple can be done by either:

- referring to its *index*
- *destructuring* the tuple and *bind it* to individually-named variables:

```
let mut tup = (31, 16, 47.27, "hello!");

// Access by index:
tup.0 = 22;
tup.3 = "world!";

// 'Destructuring' a tuple:
let (t0, t1, t2, t3) = tup;
println!("y = ({t0}, {t2})");
```

'**Tuples**' are stored on the stack unless `Box<...>` is explicitly used.

Question 1)

Take a mutable reference to the third element (`f32`) of the tuple `tup` and pass it to a function that multiplyies it by `2` :

```
// The function's prototype (to be implemented):
fn mul2(t: &mut f32);

fn main() {
    let mut tup = (31, 16, 47.27, "hello!");

    mul2(...);

    assert!(tup == (31, 16, 94.54, "hello!"));
}
```

Tuples can be conviently used in a function in order to **return multiple values**, which can then be assigned to distinct variables in a same expression:

```rust
// A function returning a pair of signed integers:
fn return_tuple(x: i32) -> (i32, i32) {
    return (x+1, x+2);
}

// Passing a tuple as an argument to a function:
fn print_tuple(x: &(i32, i32)) {
    println!("x = ({:?}, {:?})", x.0, x.1);
}

fn main() {
    // Calling a function, and storing its result:
    let y: (i32, i32) = return_tuple(8);
    println!("y = {:?}", y);
    println!("y = ({:?}, {:?})", y.0, y.1);

    print_tuple(&y);
}
```

Question 2)

Write a function that transforms a `(i32, i32)` tuple by swapping its two values:

```
// The function prototype to be implemented:
fn swap(tup: &mut(i32, i32));

fn main() {
    let mut x = (31, 27);
    swap(&mut x);

    assert!(x == (27, 31));
}
```

Arrays (primitive type)

Arrays are fixed-size groups of values of the **same type**, and can be defined in Rust with the syntax:

- `[Subtype; Length]`, for instance `[i32; 10]`

```
// Inferred type:
let b = [0, 1, 2, 3, 4]; // #[i32; 5]

// Explicit type:
let a: [i32; 3] = [31, 16, 47];
```

'**Arrays**' are stored on the stack unless `Box<...>` is explicitly used.

Arrays can be defined and **initialized** to a specific value in a single statement. They cannot be used without being initialized before.

```
// Create and zero-initialize an array:
let mut a: [usize; 10] = [0; 10];

// same as:
let mut a = [0 as usize; 10];

// same as:
let mut a = [0usize; 10];

// Writing at a specific index:
// Note: in Rust, as in C, array indices start at 0
a[0] = 2;
```

As in most programming languages, **multidimensional/nested arrays** are also supported in Rust, and can be declared as follows:

```
// 2-dimensional array, 2 arrays of `i32` with a length of 10 each:
let mut multi_array = [[0 as i32; 10]; 2];
[...]
multi_array[0][0] = 1;
```

Question 3)

What would be the type of the following arrays?

```
let a1 = [(1, 2), (3, 4), (5, 6)];

let a2 = [(1, 2), (3, 4), (5, (6, 7))];
```

Ranges & Iterators

Arrays are convenient for storing and processing a set of contiguous data on the *stack*, for instance through the use of **loops**, **ranges** and **iterators**.

Ranges

A **range** represents an *interval* of values between a *start* and an *end* point.
In Rust, they can be conveniently used with the `start..end` construct (here *excluding* the *end* value), or with `start..=end` (here *including* the *end* value).

```
let a = 0..10;

let b = 1..=10;
```

Question 4)

Examine the following `assert!` statements, will this program compile?

```rust
let a = 0..10;
let b = 1..=10;

// 'a' range:
assert!(a.contains(&0));
assert!(!a.contains(&10));

// 'b' range:
assert!(!b.contains(&0));
assert!(b.contains(&10));

// The 'a' and 'b' ranges have the same number of elements:
assert_eq!(a.count(), b.count());
```

Iterators

Iterators allow to go through an *array*, a *range* or a *collection*, and **access each element one-by-one**.

- An iterator implements the `Iterator` trait (TD4), whose main method is `next()`
- `next()` returns `Some(element)`, or `None` when the iteration is over (`Option`, TD5)
- A `for` loop calls `next()` until it gets `None`

```
let a = [10, 20, 30];
let mut it = a.iter();
assert_eq!(it.next(), Some(&10));
assert_eq!(it.next(), Some(&20));
assert_eq!(it.next(), Some(&30));
assert_eq!(it.next(), None);
```

`iter()`, `iter_mut()` and `into_iter()`

An array or a collection is **not an iterator by itself**, an iterator has to be created from it:

Method	Elements	After the iteration, the collection is…
`c.iter()`	shared references (`&T`)	unchanged, still usable
`c.iter_mut()`	mutable references (`&mut T`)	possibly modified, still usable
`c.into_iter()`	the values themselves (`T`)	**consumed** (moved into the iterator)

`for` loops call `into_iter()`

A `for` loop **automatically calls** `into_iter()` on what it is given (`IntoIterator` trait):

`for` loop	Equivalent to	Type of `x`
`for x in &c`	`for x in c.iter()`	`&T`
`for x in &mut c`	`for x in c.iter_mut()`	`&mut T`
`for x in c`	`for x in c.into_iter()`	`T`

- An **array** of `Copy` elements iterated by value is *copied*: still usable after the loop (**not** the case for a `Vec` !)
- A **range** is already an iterator: no need for `iter()` / `into_iter()` (`for n in 0..10`), it is consumed by the loop

Iterate over an array:

```
let a = [0; 10];

// Basic for-loop iteration:
for x in a {
    println!("{x}");
}

// From a 'range' of the array:
for n in 0..a.len() {
    println!("{}", a[n]);
}
```

Iterate as mutable:

```rust
let mut a = [0; 10];

// As mutable, changing the values of the array:
for x in &mut a {
    *x += 1;
}

// Equivalent to (using a 'closure'):
a.iter_mut().for_each(|x| *x += 1 );

// Iterate with both element and index:
for (i, x) in a.iter_mut().enumerate() {
    *x += i;
}
```

Iterator adapters and consumers

- **adapters** (`map`, `filter`, `enumerate`, `rev`, `zip`, `skip`, `take`, `step_by`, `copied`...) transform an iterator into another one
- **consumers** (`sum`, `count`, `min`, `max`, `collect`, `for_each`...) go through it and produce a result
- many take a **closure**: an anonymous function `|arguments| expression` (TD7)

```
let a = [1, 2, 3, 4, 5, 6];
for x in a.iter().rev().step_by(2) { print!("{x} "); } // -> 6 4 2

let b = ['a', 'b', 'c'];
for (x, c) in a.iter().zip(b.iter()) { print!("{x}{c} "); } // -> 1a 2b 3c

let s: i32 = a.iter().filter(|x| *x % 2 == 0).sum();   // 12
let doubled: Vec<i32> = a.iter().map(|x| x * 2).collect(); // [2, 4, 6, 8, 10, 12]
```

Two common pitfalls

- **Iterators are lazy**: adapters do nothing until the iterator is consumed

```
a.iter().map(|x| println!("{x}")); // prints nothing! (warning: unused `Map` that must be used)
a.iter().for_each(|x| println!("{x}")); // OK
```

- **`iter()` yields references**: the closure of `filter()` receives a `&&i32` here
  - `.copied()` (or `.cloned()`) right after `iter()` gives values instead:

```
let s: i32 = a.iter().copied().filter(|x| x % 2 == 0).sum();
```

Question 5)

Using **ranges** and/or **iterators**, write in the following *multidimensional array*'s first *sub-array* values that incrementally go from **1 to 10**, and in the second, decrement the values from **10 to 1**, as shown below:

```
let mut multi_array = [[0 as i32; 10]; 2];

// The following must be true:
assert_eq!(multi_array, [
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
]);
```

Question 6)

Without writing any `for` or `while` loop (only iterator methods):

- compute the sum of the squares of the odd numbers between `1` and `9` (included);
- build a `Vec<i32>` containing the elements of `a` greater than `10`, in reverse order;
- count the even elements of `a`.

```
let a = [4, 8, 15, 16, 23, 42];
let s: i32 = (...);
assert_eq!(s, 165);
let v: Vec<i32> = (...);
assert_eq!(v, vec![42, 23, 16, 15]);
let n = (...);
assert_eq!(n, 4);
```

Collections

In addition to primitive compound types, the Rust **standard library** includes a number of very useful data structures called **collections**. Unlike the built-in *array* and *tuple* types, the data these collections point to is stored **on the heap**, which means the amount of data does not need to be known at compile time and can grow or shrink as the program runs.

Vectors

- Collection of multiple values of **a same type** stored on the **heap**.
- **Dynamic size**: they can grow, or shrink.
- *Ownership* over the data located in its underlying *heap-allocated buffer*.

```rust
// Using the 'vec!()' macro:
let mut v = vec![0, 1, 2, 3, 4, 5]; // Vec<i32>

// 'Pushing' (appending) a new value at the end:
v.push(6);

// 'Popping' (removing) its last value:
let last = v.pop();

// Removing the value at index 0:
v.remove(0);
```

Question 7)

Examine the following `v0`, `v1` and `v2` vectors and their underlying heap buffer pointers.

```
let mut v0: Vec<i32> = vec![0, 1, 2, 3, 4];

// get a pointer to the underlying heap memory buffer:
let v0_ptr: *const i32 = v0.as_ptr();

// Create another vec 'v1' from 'v0', and get its heap pointer again:
// i.e. move v0 to v1
let mut v1: Vec<i32> = v0;
let v1_ptr = v1.as_ptr();

// Create another vec 'v2' from 'v1' using clone():
let mut v2: Vec<i32> = v1.clone();
let v2_ptr = v2.as_ptr();
```

Which of the following assertions are `true`:

```
// Assertion A: the address of pointer 'v0' is the same as pointer 'v1'
assert_eq!(v0_ptr.addr(), v1_ptr.addr());

// Assertion B: the address of pointer 'v1' is the same as pointer 'v2'
assert_eq!(v1_ptr.addr(), v2_ptr.addr());

// Assertion C: the address of pointer 'v0' is the same as pointer 'v2'
assert_eq!(v0_ptr.addr(), v2_ptr.addr());
```

Iterate over a vector

Iterating over a vector is the same process as for an array (most operations are inter-compatible!).

```
// Initializing from a range and iterator:
let mut v = Vec::from_iter((0..6).map(|i| i+1 ));

// Iterate/increment:
for x in &mut v {
    *x += 1;
}

// General operations:
v.rotate_left(1);
```

Iterating over a vector by value moves it

A `Vec` is **not** `Copy`: `for x in v` (i.e. `v.into_iter()`) moves `v` into the loop

```
let v = vec![1, 2, 3];
for x in v {            // 'v' is moved here (into_iter)
    println!("{x}");
}
println!("{:?}", v);    // error[E0382]: borrow of moved value: `v`
```

- To only read the elements: `for x in &v` (or `v.iter()`)
- Iterating by value is useful to **take ownership** of the elements (no copy):

```
let names = vec![String::from("Alice"), String::from("Bob")];
let mut greetings = Vec::new();
for name in names {             // each 'String' is moved out of 'names'...
    greetings.push(name + "!"); // ...modified and moved again: no copy
}
```

Modifying a collection while iterating

The borrowing rules (TD2) also apply to iterators:

```
let mut v = vec![1, 2, 3];
for x in &v {      // 'v' is borrowed (immutably) by the loop...
    v.push(*x);    // error[E0502]: cannot borrow `v` as mutable
}                  //               because it is also borrowed as immutable
```

- Iterate over a range of indices instead (`for i in 0..v.len()`), or build a new collection

Question 8)

Using a *single loop*, move the contents of vector `v` to array `a` such as vector `v` is equal to `vec![]` (empty vector) and array `a` is equal to `[5, 4, 3, 2, 1, 0]`

```
let mut v = vec![0, 1, 2, 3, 4, 5];
let mut a = [0; 6];

(...)

assert_eq!(v, vec![]);
assert_eq!(a, [5, 4, 3, 2, 1, 0]);
```

Question 9)

For each of the following loops, give the type of `x`, and tell whether `v` can still be used after the loop:

```
let mut v = vec![String::from("a"), String::from("b")];

// Loop 1:
for x in v.iter() { (...) }
// Loop 2:
for x in &mut v { (...) }
// Loop 3:
for x in v.clone() { (...) }
// Loop 4:
for x in v { (...) }
```

Question 10)

The following program does not compile: explain why, and fix it (try to find two different solutions).

```
let words = vec![String::from("iterators"), String::from("are"), String::from("lazy")];
let mut total = 0;
for w in words {
    total += w.len();
}
println!("{} words, {total} characters", words.len());
```

Collections: Hash-maps

- Heap-allocated collections of same-type values **indexed by a *unique* key**.
- Like vectors, they can grow, or shrink.

```
// Unlike Vec, the HashMap data structure need to be explicitly included!
use std::collections::HashMap;

fn main() {
    // Inferred type:
    let mut departments = HashMap::new(); // HashMap<i32, &str>

    departments.insert(85, "Vendée");
    departments.insert(31, "Haute-Garonne");
    departments.insert(44, "Loire-Atlantique");
}
```

Manipulating Hash-maps

```
[...]

let d31 = departments[&31];
assert_eq!(d31, "Haute-Garonne");

// Removing a key:
departments.remove(&85);

// Iterating over all values:
for department in departments {
    // We get a tuple!
    println!("Key: {}, Value: {}", department.0, department.1);
}
```

Question 11)

Move the contents of the following `Vec` object into a `BTreeMap` in order to get these athlete names sorted by their score in points. *Note*: some of them have the same score, which should appear in the same `key`.

```
use std::collections::BTreeMap;

fn main() {
    let vec = vec![
        ("Y. Horigome", 281), ("N. Huston", 279), ("M. Dell", 153),
        ("J. Eaton", 281), ("S. Shirai", 278), ("K. Hoefler", 270),
        ("C. Russell", 211), ("R. Tury", 273)];

    let mut map = BTreeMap::new();

    [...]
}
```

Hints:

- build a BTreeMap with the key being the score
- use the `entry(key)` method of `BTreeMap`, which returns the map entry corresponding to `key`
- apply `or_default()` on the result of `entry` (it creates an entry with a default value, here an empty `Vec`, if the key does not exist in the map)

'string' types: `str`

- Can be used to represent a **string literal**
- Has a **static lifetime** which can be also explicitly stated in its type declaration.
- It means that the object is valid throughout **the entire duration of the program**.

```
// Here, the three syntaxes are equivalent:

let s = "Hello, World!"; // Inferred type & lifetime

let s: &str = "Hello, World!"; // Explicit type, inferred lifetime

let s: &'static str = "Hello, World!"; // Explicit type & lifetime
```

- Unlike `const char*` in the **C programming language**, `&str` in Rust is **not null-terminated**.
- It relies on a **slice**, which is composed of a pointer and a size in bytes:

```
let s = "Hello, World!";
println!("Pointer: {:?}, Length: {} bytes", s.as_ptr(), s.len());

for (n, char) in s.chars().enumerate() {
    println!("Char {n}: {char}");
}
```

- They're also **always immutable!**

```
let s: &mut str = "Hello, World!"; // Does not compile!
```

Question 12)

```
let s1 = "It's not about the bunny      \t";

// trim() Returns a string slice with leading and trailing whitespace removed.
let s2 = s1.trim();
println!("{s1}");
println!("Address: {:?}, Length: {}", s1.as_ptr(), s1.len());
println!();
println!("{s2}");
println!("Address: {:?}, Length: {}", s2.as_ptr(), s2.len());
```

Since Rust forbids modifying the contents of a `str` literal, why are we in this case allowed to use the `.trim()` function? What is truly happening in this code?

What would happen if we modified `s1` as follows?

```
let s1 = "     It's not about the bunny      \t";
let s2 = s1.trim();
```

'string' types: `String`

- **Standard library collection** type that can be basically seen as a vector of UTF-8 encoded bytes (`Vec<u8>`), dynamically stored *on the heap*.
- It can grow, shrink, and has ownership over its own underlying buffer, which makes it an easier object to manipulate than `str` , while inheriting all of its methods.
- Does not have a static lifetime.

```rust
// Create from a string literal:
let mut s = String::from("Owls are not what they seem");

// Append a 'char':
s.push('!');
println!("Value: {:?}", s);

// Append another string:
s.push_str(" Really?");
println!("Value: {:?}", s);

// Other ways of appending to the String:
s = s + " Yes, ";
s += "really!";
```

Iterating and manipulating `String`:

```
// Iterate over every 'char'
for c in s.chars() {
    print!("{c} ");
}
println!("");

// Example of transformation:
s = s.chars().rev().collect();

println!("Value: {:?}", s);
```

Question 13)

Examine the following code:

```rust
let s0: &str = "That gum you like is going to come back in style";

// Build a 'String' object from the previous '&str':
let mut s1: String = String::from(s0);

// Now modify 'string':
s1 = s1.to_ascii_uppercase();
println!("{string}");
```

Is `s0` still accessible? If yes, what is now its value? Is it the same as `s1` and why?

We now call the `as_str()` method on `s1`, and store the resulting `&str` value in a new variable called `s2`. Can you guess what is the **lifetime** of `s2`?

```
let s0: &str = "That gum you like is going to come back in style";

// Build a 'String' object from the previous '&str':
let mut s1: String = String::from(s0);
let s2: &str = s1.as_str();
```

Slices

- **Bounded pointer or reference** to a contiguous sequence of elements in an array, a collection, or a string. It **does not have ownership** over its contents.

```
// Create a byte buffer:
let mut buffer = [0 as u8; 16];

// Get a slice on half the buffer, using the 'range syntax':
// (notice how the slice itself is not mutable,
// but instead points to a mutable sequence in the buffer
let slice: &mut[u8] = &mut buffer[0..8];

// Iterate on the slice to change values:
for (i, n) in slice.iter_mut().enumerate() {
    *n = i as u8;
}
```

Question 14)

- Take a slice out of the `string` object, starting from character `25` until the end, and use the `.make_ascii_lowercase()` method on the captured slice.

```
let mut string = String::from(
    "YOU REMIND ME TODAY OF A SMALL MEXICAN CHIHUAHUA"
);

let slice = ...;

slice.make_ascii_lowercase();

println!("{string}");
```

- What is the inferred type of the `slice` variable?

End of TD3