Rust Programming Language: the Basics

Luca Abeni
luca.abeni@santannapisa.it

March 29, 2025

Rust Programming Language Ideas

```
fn main() {
  println!("Hello,␣world!")
}
```

- C-like syntax (see Rust ”hello world”...)
  - But support for higher-level abstractions!
- No heavy runtime (no GC, type/memory checks are mostly static, ...)
  - Without loosing safety...
- Try to provide control to user (do not hide memory allocation/deallocation, ...)
  - Only when needed

C: Control to User

```
struct s {
  int v;
  ...
};

p    = malloc(sizeof(struct s));
p->v = 5;
...
free(p)
```

- Control on the memory layout of data
  - Even better: “`packed`” attribute and “`int`*xx*`_t`” types
- Control on the amount of allocated memory
- Control on when memory is allocated/deallocated

Too Much Control?

- Usual issues: things like

```
    p = malloc(sizeof(int)); ???
    free(p); a = p->v;
...
```

- Control on memory (de)allocations risks to allow errors on `malloc()` and `free()`
- Control on pointers creates issues with aliasing/leaks
- We know a *possible* solution: RAII

Rust and RAII

```
struct S {
   v: i32,
   ...
}

fn WorkOnS() {
   let mut p = Box::new(S {v: 5, ...});

   p.v = ...
     /* use p ... */
     ...
}
```

- When `p` goes out of scope, memory is deallocated!
  - Problem: things like “`let mut p1 = p`” risk to break the thing!
  - Rust has to somehow make sure that there is only an *active* reference/pointer to the structure

Rust Vision of “Control to User”

- In the Rust example, notice:
  - Control on the structure size (“`i32`”)
  - Explicit memory allocation (“`Box::new(S v: 5, ...)`”)
  - No constructors!
  - Control on the variable mutability (“`let` **mut** `p`”)
- The type of “`p`” (pointer to “`struct S`” — `Box<S>`) is not explicitly specified
  - Type inference!

Rust and Assignments (Move Semantics)

- Here, Rust needs to enforce that there is only one pointer to the allocated structure:

```
struct S {
  v: i32
}

fn work_on_s() {
  let mut p = Box::new(S {v: 5});
```

- Assignments have *move semantics*: "`let p1=p`" moves the ownership of the structure from "`p`" to "`p1`" $\Rightarrow$ after this, "`p`" is invalid
- So, this does not build:

```
let mut p = Box::new(S {v: 5});
let p1 = p;
println!("v:{}", p.v);
```

Move and... Borrow?

- Assignment: move the ownership of a data structure
  - Can a value be "borrowed"?
  - Meaning, "`p`" owns a data structure; passes it to "`p1`" and gets it back when "`p1`" goes out of scope
  - While the value is borrowed, "`p`" cannot modify it...
- Yes, we can! Use references ("`&`")

```
let mut p = Box::new(S {v: 5});
{
  let p1 = &p;
  println!("p1.v:{}",p1.v)
}
println!("v:{}", p.v);
```

- "`p.v = 666;`" in the inner block can fail to build

Borrowing: Rules

- A value owned by a variable can be borrowed as mutable or as immutable
  - Mutable reference ("`&mut`") or immutable references ("`&`")
  - Mutable reference: only one; immutable references: can borrow multiple times
- When borrowed, it cannot be modified by the original owner
- `rustc` sometimes does "smart things" (if a variable is not used after a line of code, it is considered dropped there)
- Borrowing is used also for function parameters (passed by reference)

Rust Syntax: the Basics

- C-Like syntax: program written as a set of functions
  - Special "`main`" function invoked when the program is executed
- Function: block of code associated to a name (+ environment + parameters + return value)
  - Syntax: "`fn` *name*`(`*parameters*`) ->` *return type*" followed by a block of code
  - Special case: if the return type is "`()`" (unit type), "`-> ()`" can be avoided
- Block of code: contains variable definitions and expressions
  - As in C, C++, Java, ..., start with "{" and finish with "}"

Rust Syntax: Peculiarities

- Difference with C & friends: meaning of "`;`"
  - No "end of instruction", but separator between expressions
- A block of code is an expression
  - Evaluates to the value of the last expression of the block
  - Special case: if the last expression is "`()`", it can be removed
  - Example: "`{println!("Hi"); ()}`" and "`{println!("Hi");}`" are the same
  - Example: "`{5;}`" and "`{5}`" are different (the first evaluates to "`()`", the second to "`5`"
- Corollary: no need for a "`return`" keyword!

So... Hello!!!

- Let's start with a "hello world" program...
  - "`main`" function taking no arguments and returning no value
    - "returning no value" means "returning a value of unit type"
    - Unit type: type having only one value: "`()`"
  - Remember: "`-> ()`" can be avoided
- To print values on stdout, use the "`println!()`" macro

```
fn main() {
     println!("Hello,␣world!")
}
```

- Notice: no "`;`" at the end... Why?

Slightly More Interesting Example

```
fn mult2(v: i32) -> i32 {
   v * 2
}

fn main() {
   let number = 5;
   let number2 = mult2(number);

   println!("{} multipled by 2 is {}",
                 number, number2)
}
```

- Notice how "`mult2`" returns its result
- To print the content of a variable, use "{}" in the format string
  - As convenient as C's `printf()`...
  - ...But safer! The compiler can actually check the type of each printed variable

The Rust Type System

- Set of predefined types
  - The usual scalar types (will see in next slides)
- Set of mechanisms for building new types (based on existing ones)
  - Based on algebraic data types
  - Product types (structures and tupes) and sum types (enums)
- Set of rules for working with types
  - Rust is statically typed
    - Types of variables known at build time
  - Strict compatibility rules
  - Type inference by default

Type Inference

- The compiler tries to *infer* the type of variables
  - No need to always specify variable types...
  - ...But, sometimes, the compiler might use some help!
- Example: this fails to build:

```
let s = "123".to_string();
let n = s.parse().unwrap();
```

- "`parse()`" returns a type encapsulating the result...
  - But, which type is the result? (integer? floating point? ...?)
- Type annotations are needed, here!

```
let n = s.parse::<f64>().unwrap();
let n1: i32 = s.parse.unwrap();
```

Scalar, Compound, and Custom Types

- Different ways to classify types...
- ...But a distinction between scalar types and compound types is generally recognized
  - Again, various definitions (of “scalar”, in this case!)
  - Rust also introduce custom types (structures and enumerations)
- Primitive (predefined) types are generally scalar
- In Rust, 4 classes of scalar types: integers, floating point, boolean, and character
- Debatable thing: the unit type “`()`”
  - Is it a scalar type (with only one value “`()`”)...
  - Or is it a tuple with 0 elements?

Rust Never Type and Unit Type

- Never: type "`!`" with no possible values
  - What? How is it useful?
  - Return value of functions that never return...
  - Considered compatible with every other type...
- Unit: type "`()`" with one single value "`()`"
  - Similar to the "`void`" type of other languages
  - Used for functions returning no values
- Is it a tupe (compound type) or a scalar type?
  - Official Rust documentation is not clear about this: `https://doc.rust-lang.org/rust-by-example/primitives.html`

    `https://doc.rust-lang.org/reference/types/tuple.html`

Rust Boolean Type

- Type `bool`, encoded on 1 byte, with only two values
  - `true`, `false`
- Used for boolean predicates (in `if`, etc...)
- Big difference with C: `bool` is not compatible with integer types
  - "`if (d) res = n / d; else res = 0;`" is valid C
  - "`if d {res = n / d;} else {res = 0;}`" is not valid Rust
  - Should be "`if d != 0 {res = n / d;} else {res = 0;}`"
  - More rusty: "`res = if d != 0 {n / d} else {0}`"

Rust Integer Types

- Rust allows to control both size and encoding
- Can be signed or unsigned
  - Signed: two's complement (difference with C: the encoding is specified) $\in [-(2^{b-1}), 2^{b-1} - 1)]$
  - Unsigned: $\in [0, 2^b - 1]$
- Represented on 8, 16, 32, 64 or 128 bits
- `i8`, `i16`, `i32`, `i64`, `i128` and `u8`, `u16`, `u32`, `u64`, `u128`
  - "`isize`" and "`usize`" types: represented on an architecture-dependent number of bits

Integer Overflow in Rust

- No C-like UBs, but behaviour dependent on compilation options
  - Program compiled in debug mode (default) $\rightarrow$ mathematical operations causing overflows crash (`panic()`)
  - Program compiled in release mode ("`rustc -O`") $\rightarrow$ mathematical operation causing overflows use modular arithmetic
- Notice: both these behaviours are safe!

Rust Floating Point Types

- Represented on 32 or 64 bits
  - Using the IEEE 754 standard
  - 32 bits is single precision
  - 64 bits is double precision
- `f32` and `f64`
- `f64` is default ("`let f = 3.14`" gives an `f64` variable)

Rust Characters

- Type `char`, similar to C characters
  - Same syntax ("`c = 'a'`")
- Big difference: stored on 4 bytes, encode Unicode Scalar Values
  - Whatever they are...

Compound Types

- Tuples and arrays
  - Both can be seen as product types
  - Tuple: elements can have different types; generally accessed through pattern matching
  - Arrays: uniform (all elements have the same type); can be accessed through an index
- Tuple: list of comma-separated values, inside parentheses
  - Example: “`(3.14, "pi")`”
  - Also possible to give hints about the types: “`let t: (f32, &str) = (3.14, "pi")`”

Compound Types — Arrays

- Array: list of comma-separated values, inside square brackets
  - Example: "`[3.14, 6.28]`"
  - Things like "`[2, 3.14]`" are not OK
- Array of "`n`" elements initialized to "`v`": "`[v; n]`"
- Random access to single elements is possible
  - And array bounds are checked!
- Rust arrays are not vectors (fixed size, cannot grow)
- Rust introduces some complications due to "*slices*"... Will see later!

Custom Types

- Built using structures and enumerations
- Based on algebraic data type: product and sum
- Structures: C-like "`struct`" syntax
  - This is a simplification; tuple-like structures and empty structures also exist
- Enumerations: "`enum`" keyword, followed by a comma-separated list of variants (inside "`{ }`")
  - Single-value variants: similar to C-style enums
  - Variants generated by a constructor with parameters... Rust uses structures (mainly tuple-style, but C-style could be used too)
- Method and functions can also be attached to structures and enumerations...

Compound Types and Custom Types

- Compound types: tuples and arrays
  - Products of other types
  - Array: all elements have the same type; elements accessible through an index
- Custom types: structures and enumerations
  - Products and sums (disjoint unions) of other types
  - Allow to *define new types* and give them a name
- A value of a compound or custom type is composed by multiple elements
  - How to access the single elements?
  - *Destructure* the type, or *unwrap* the contained values

Arrays

- Collections of n elements of the same type `T`: "`[T; n]`"
  - Random access is possible; out-of-bound accesses are checked
  - The array size n is part of the type: possible checks at build time and at runtime
- Dynamically sized view of an array: slice
  - A slice can be seen as a reference "`&[T]`"
  - The slice size is stored somewhere in a "fat pointer" data structure
- Slices are initialized from arrays: whole array (`let s = &v[..]`) or part of an array (`let s = &v[2..6]`)

Tuples

- Products of different types
- Elements can be accessed through pattern matching
  - Destructuring the tuple
  - No accesses through index variables
- Special “structure-like” syntax

```
let t = ("Hi!", 2);
let (v1, v2) = t;

println!("{}␣{}", v1, v2);
println!("{}␣{}", t.0, t.1);
```

Structures

- Product type, with a name
- Difference with tuples: each field has a name
- Fields accessible through their names, using C-like dot notation
- Can also be quickly destructured using pattern matching

```
struct Point {
  x: f32,
  y: f32,
  z: f32
}

  let s = Point {x: 1.0, y: 1.0, z: 1.0};
  let Point {x: x1, y: y1, z: z1} = s;

  println!("{}␣{}␣{}", x1, y1, z1);
  println!("{}␣{}␣{}", s.x, s.y, s.z)
```

Strange Structures

- Unit structure: no fields
  - Unit-like type (type with a single value) with a name
  - "`struct EmptyStruct;`"
  - Will be useful for building enumerations (variants with single value)
- Tuple-like structures
  - Again, will be useful for building enumerations
  - "`let s = StrangeStruct(1.0, 2.0, 3.0);`"
  - Pattern matching will be useful here, too

Enumerations

- Sum type, with a name
- Comma-separated list of structures
  - These structures are the constructors
  - Unit structures: constructors with no argument (C enums)
  - Tuple structures: constructors with arguments (an argument per field); C unions
  - C-like structures can be used too, but are less useful
- Pattern matching is the only way to destructure them/unwrap the contained values
  - For enumerations, there is no other way to access the type elements

Rust Variables

- Variables are defined using the “`let`” keyword
  - Typically defined and initialized at the same time
  - The compiler can generally infer the type of a variable
- As usual, can be mutable or immutable
  - Rust variables are immutable by default
  - Mutable variables must be explicitly defined as so (“`let mut`”)
  - If a variable is defined as mutable without apparent reasons, the compiler complains!
- Assignments can be performed only on mutable variables

Example

This does not compile

```
fn main() {
  let x = 5;
  println!("The␣value␣of␣x␣is:␣{}", x);
  x = 6;
  println!("The␣value␣of␣x␣is:␣{}", x);
}
```

Changing “`let x = 5;`” into “`let mut x = 5;`” fixes the issue.

Shadowing

- Shadowing: the same name can be associated to multiple variables
  - The last "active" (in scope) binding is used
  - Something like this is valid:

```
fn main() {
  let x = 5;
  println!("The␣value␣of␣x␣is:␣{}", x);
  let x = 6;
  println!("The␣value␣of␣x␣is:␣{}", x);
}
```

- "`let x = 6;`" is the definition of a new variable, not an assignment

Shadowing

- To better understand shadowing, try this:

```
fn main()
{
  let x = 5;

  println!("The␣value␣of␣x␣is:␣{}", x);
  {
    let x = 6;

    println!("The␣value␣of␣x␣is␣now:␣{}", x);
  }
  println!("The␣value␣of␣x␣is␣now:␣{}", x);
}
```

- The second “`let x`” defines a new variable; when it goes out of scope, the first “`x`” is used

Pattern Matching

- Rust provides a powerful pattern matching mechanism, that can be used to:
  - Implement “case-like” switches
  - Define variables, assigning values to them
  - Destructure complex data types
  - Unwrap values contained in algebraic data types
- Pattern matching is used in various construcs, such as:
  - `match`, `if let`, and similar
  - Variables definitions (`let` statements)
  - Parameters passing

Rust Patterns

- A pattern can be:
  - A value (literal, constant)
  - A variable
  - A compound or custom type (tuple, structure, enumeration, ...)
  - The “wildcard pattern”
- A pattern is “matched” by comparing it with some value
  - A constant/literal obviously matches with its value
  - A variable matches with any value of the same type
    - Example: “`let pi = 3.14`”

More Complex Matching Rules

- A compound/custom value matches if all the elements match
  - For tuples, it is simple:

    ```
    let t = ("Hi!", 2);
    let (v1, v2) = t;
    ```

  - Here, "`(v1, v2)`" matches "`("Hi!", 2)`" because "`v1`" matches "`"Hi!"`" and because "`v2`" matches "`2`"