Rust Data Types

Andreea Costea

Delft University of Technology

2025-09-04

(slides adapted from Jana Dönszelmann)

TU Delft

Data Types in General

Question:

What is a Data Type?

Data Types in General

Question:

What is a Data Type?

A Data Type classifies values and determines:

- What values exist (the **domain**)
- How are they represented in **memory**
- What **operations** are allowed on those values
- How the values **behave** when operations are applied

Rust Data Types

Scalar Type

- Integers: `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `u128`, `i128`, `usize`, `isize`
- Floating point: `f32`, `f64`
- Boolean: `bool` (true, false)
- Character: `char` (a Unicode Scalar Value, e.g., ‘a’, ‘α’, ‘∞’)

Compound Type

- Tuple
- Array
- Structs

Scalar Types - Integer example

- `u8` is just one byte
- `u32` is 4 bytes

Rust

```rust
// compiler, when I call `foo` somewhere in my code
fn foo() {
  // please make some room for me to use 4 bytes for something
  // I'll use the name `a` when I want to use it
  let a: u32 = 3;

  // and 16 more here, I'll call it b
  let b: u128 = 100_000;
}
```

Scalar Types - Integer in General

- What values exist (the **domain**)?

Scalar Types - Integer in General

- What values exist (the **domain**)?

Type	Minimum	Maximum
`u8`	0	2^8-1
`u16`	0	2^{16}-1
`u32`	0	2^{32}-1
`u64`	0	2^{64}-1
`u128`	0	2^{128}-1

Type	Minimum	Maximum
`i8`	$-(2^7)$	2^7-1
`i16`	$-(2^{15})$	2^{15}-1
`i32`	$-(2^{31})$	2^{31}-1
`i64`	$-(2^{63})$	2^{63}-1
`i128`	$-(2^{127})$	2^{127}-1

Scalar Types - Integer in General

- What values exist (the **domain**)?

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

Scalar Types - Integer in General

- What values exist (the **domain**)?
- How are they represented in **memory**?

Scalar Types - Integer in General

- What values exist (the **domain**)?
- How are they represented in **memory**?

Rust

```rust
pub static a: u32  = 1;
pub static b: u64  = 0x01;
pub static c: u128 = 0b00_01;
```

https://godbolt.org/z/EPnoz5cre

Scalar Types - Integer in General

- What values exist (the **domain**)?
- How are they represented in **memory**?
- What **operations** are allowed on those values?
- How the values **behave** when operations are applied?

Scalar Types - Integer in General

- What values exist (the **domain**)?
- How are they represented in **memory**?
- What **operations** are allowed on those values?
- How the values **behave** when operations are applied?

Rust

```rust
fn main() {
    let age:u8 = u8::MAX;   //255
    let x:u8 = u8::MAX + 1;
    let y:u8 = u8::MAX + 2;
    println!("age is {} ",age);
    println!("x is {}",x);
    println!("y is {}",y);  }
```

https://godbolt.org/z/EPnoz5cre

Scalar Types - Integer in General

- What values exist (the **domain**)?
- How are they represented in **memory**?
- What **operations** are allowed on those values?
- How the values **behave** when operations are applied?

Scalar Types - Integer in General

- What values exist (the **domain**)?
- How are they represented in **memory**?
- What **operations** are allowed on those values?
- How the values **behave** when operations are applied?

Rust:

Force users to make conscious choices about potentially unsafe operations.

Compound Data Types - Arrays

```rust
let a: [u8; 8] = [1, 2, 4, 8, 16, 32, 64, 128];
let b: [u8; 8] = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80];
let c: u64     = 0x01_02_04_08_10_20_40_80;
```

Compound Data Types - Arrays

```rust
let a: [u8; 8] = [1, 2, 4, 8, 16, 32, 64, 128];
let b: [u8; 8] = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80];
let c: u64     = 0x01_02_04_08_10_20_40_80;
```

Question:

are a, b and c the same?

https://godbolt.org/z/G3doh6e4v

Compound Data Types

1. Arrays [T; N]
   - Fixed-size, allocated on the stack (unless part of a heap-allocated structure).
   - Contiguous elements, so arr[0], arr[1], … are stored back-to-back in memory.
   - Access is fast (constant time) because the compiler can compute offsets

```rust
let arr: [i32; 3] = [1, 2, 3];

let first = arr[0];
let second = arr[1];

println!("Array: {:?}", arr);
```

Compound Data Types

1. Arrays ([T; N]): fixed-size, stack-allocated, cannot change length.

Compound Data Types

1. Arrays ([T; N]): fixed-size, stack-allocated, cannot change length.
2. Vectors (Vec):

- Dynamic-size, heap-allocated, can grow or shrink.
- The buffer inside the vector is contiguous on the heap.

```rust
let mut v = vec![1,2,3]; // The numbers 1, 2, 3 are stored contiguously in memory.
```

- `v` is a stack-allocated variable representing the pointer to the buffer, plus length and capacity metadata.

Compound Data Types: grouped values of different types

Different types bundled together, called a “tuple”:

```rust
let today: (u8, u8, u32) = (4, 9, 2025);
let tomorrow: (u8, u8, u32) = (5, 9, 2025);
```

Compound Data Types: grouped values of different types

Different types bundled together, called a “tuple”:

```rust
let today: (u8, u8, u32) = (4, 9, 2025);
let tomorrow: (u8, u8, u32) = (5, 9, 2025);
```

Which we can name:

```rust
type Date = (u8, u8, u32);
// ...
let today: Date = (4, 9, 2025);
let tomorrow: Date = (5, 9, 2025);
```

Compound Data Types: grouped values of different types

Or a more common way to write that:

```rust
struct Date {
  day: u8,
  month: u8,
  year: u32,
}
// ...
let today: Date = Date {
  day: 4,
  year: 2025,
  month: 9,
};

let year = today.year;
```

Struct layout

- Rust has lots of freedom with struct layouts
- https://doc.rust-lang.org/reference/type-layout.html
- Optimized code can take advantage of this

Compound Data Types: Struct

2. Vectors (Vec): dynamic-size, heap-allocated, can grow or shrink.

```rust
let mut v:Vec<i32> = vec![1,2,3];
```

Rust

- `v` is a stack-allocated variable representing the pointer to the buffer, plus length and capacity metadata.

Compound Data Types: Struct

2. Vectors (Vec): dynamic-size, heap-allocated, can grow or shrink.

```rust
let mut v:Vec<i32> = vec![1,2,3];
```

- `v` is a stack-allocated variable representing the pointer to the buffer, plus length and capacity metadata.

```rust
struct Vec<T> {
  ptr: *mut T,     // pointer to heap-allocated buffer
  len: usize,      // number of elements
  capacity: usize // allocated space in buffer
}
```

Exercise: 5 minutes

Go to play.rust-lang.org or open your fav editor and define a `struct UdpHeader` with these fields:

<table>
<tr><th colspan="34">IPv6 pseudo header format</th></tr>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><th>0</th><th>0</th><td colspan="32" rowspan="4">Source IPv6 Address</td></tr>
<tr><th>4</th><th>32</th></tr>
<tr><th>8</th><th>64</th></tr>
<tr><th>12</th><th>96</th></tr>
<tr><th>16</th><th>128</th><td colspan="32" rowspan="4">Destination IPv6 Address</td></tr>
<tr><th>20</th><th>160</th></tr>
<tr><th>24</th><th>192</th></tr>
<tr><th>28</th><th>224</th></tr>
<tr><th>32</th><th>256</th><td colspan="32">UDP Length</td></tr>
<tr><th>36</th><th>288</th><td colspan="24">Zeroes</td><td colspan="8">Next Header = Protocol[14]</td></tr>
<tr><th>40</th><th>320</th><td colspan="16">Source Port</td><td colspan="16">Destination Port</td></tr>
<tr><th>44</th><th>352</th><td colspan="16">Length</td><td colspan="16">Checksum</td></tr>
</table>

Adding a behavior to a type

- The `impl` keyword

```rust
struct SomeType {...};

impl SomeType {
  fn do_something1(...) {}          // associated functions
  fn do_something2(&self) {}        // associated methods
  pub const SOME_CONSTANT: u8 = 42; // associated constants
  ...
}
```

Adding a behavior to a type

- The `impl` keyword
- For example:

Rust

```rust
struct UdpHeader{
  ... // fields
};

impl UdpHeader {
  pub const ZEROS: u8 = 0; //no memory overhead per instance.
}
```

Adding a behavior to a type

- The `impl` keyword
- For example:

```rust
// somewehere in the standard library
struct u64;

impl u64 {
  fn add(&self, other: Self) -> u64 {}
}
```

Adding a behavior to a type

- The `impl` keyword
- For example:

```rust
// somewehere in the standard library
struct u64;

impl u64 {
  fn add(&self, other: Self) -> u64 {}
  // actualy roughly means:
  fn add(&self: u64, other: u64) -> u64 {}
}
```

Adding a behavior to a type

- The `impl` keyword
- Now we can:

```rust
// somewhere in the standard library
impl u64 {
  fn add(&self, other: Self) -> u64 {}
}

let x: u64 = 3;

x.add(5)
```

Adding a behavior to a type

- The `impl` keyword
- Now we can:

Rust

```rust
// somewhere in the standard library
impl u64 {
  fn add(&self, other: Self) -> u64 {}
}

let x: u64 = 3;

x.add(5)
// but you might be more used to:
x + 5
```

Exercise: 5 minutes

Go to play.rust-lang.org or open your fav editor and:

- Create a struct called `Range` with two integer fields, `start` and `end`
- Add these functions to the `Range` type
  - `len` which says how far `start` is from `end`
  - `middle` which gives the middle of the range
  - `new` which creates a new range and checks whether `end` > `start`

Hint: use `assert!(a > b)` to make sure conditions hold (and panic otherwise)

Types as Proofs

- Types don't have to be only related to memory
- They can also communicate that you checked something.

Types as Proofs

- Types don’t have to be only related to memory
- They can also communicate that you checked something.

```rust
struct Range { start: usize, end: usize,}

impl Range {
    fn new(start: usize, end: usize) -> Self {
        assert!(end > start, "end must be greater than start");
        Range { start, end }
    }
    pub fn len(&self) -> usize {
      self.end - self.start // cannot fail! (can't be negative)
    }
}
```

Types as Proofs

- Types don't have to be only related to memory
- They can also communicate that you checked something.

- Examples:
  - `NonZero<T>` proves that the integer `T` is not zero
  - `&str` is like bytes (`&[u8]`), but proves UTF-8
  - `String` is like a `Vec<u8>` but proves UTF-8

Types as Proofs

- Types don't have to be only related to memory
- They can also communicate that you checked something.

- Examples:
  - `NonZero<T>` proves that the integer `T` is not zero
  - `&str` is like bytes (`&[u8]`), but proves UTF-8
  - `String` is like a `Vec<u8>` but proves UTF-8

```rust
let bytes = vec![0xff, 0x61]; // Vec<u8> ok, not valid UTF-8
// let s = String::from_utf8(bytes).unwrap(); // would panic, invalid UTF-8

let s = String::from("hello"); // guaranteed UTF-8
```

Types as Proofs

- Types don't have to be only related to memory
- They can also communicate that you checked something.

- Examples:
  - `NonZero<T>` proves that the integer `T` is not zero
  - `&str` is like bytes (`&[u8]`), but proves UTF-8
  - `String` is like a `Vec<u8>` but proves UTF-8

- Zero-sized types are even possible:

```rust
struct ZeroSized {};
let x: ZeroSized = ZeroSized {};
```

https://www.hardmo.de/article/2021-03-14-zst-proof-types.md

Scalar Types - Integer in General

- What values exist (the **domain**)?
- How are they represented in **memory**?
- What **operations** are allowed on those values?
- How the values **behave** when operations are applied?

```rust
fn main() {
    let age:u8 = u8::MAX;   //255
    let x:u8 = u8::MAX + 1;
    let y:u8 = u8::MAX + 2;
    println!("age is {} ",age);
    println!("x is {}",x);
    println!("y is {}",y);  }
```

https://godbolt.org/z/EPnoz5cre

Compound Data Types: Struct

```rust
struct Date {
    day: u8,
    month: u8,
    year: u32,
}
```

Rust

Question:

How many bytes do we need for a `Date`?

https://godbolt.org/z/PYs6WTzvq

Compound Data Types: Struct

Rust:

Rust needs to know the size of every type at compile time because it stores variables directly on the stack.

Compound Data Types: Struct

Rust:

Rust needs to know the size of every type at compile time because it stores variables directly on the stack.

```rust
struct Node {
  value: i32,
  next: Node, // ERROR: recursive type
}
```

Compound Data Types: Struct

Rust:

Rust needs to know the size of every type at compile time because it stores variables directly on the stack.

```rust
struct Node {
  value: i32,
  next: Node, // ERROR: recursive type
}
```

- The compiler cannot know the size of Node: Node contains a Node,… infinitely.
- Problem: this is a type with **unknown size** — Rust refuses this.

Compound Data Types: Nested Structs

Solution: `Box<T>` - a pointer to a value on the heap.

- A pointer always has a known, fixed size.
- So if we wrap a recursive type in `Box`, Rust now sees:

```rust
struct Node {
    value: i32,
    next: Box<Node>, // fixed size pointer
}
```

Compound Data Types: Nested Structs

Solution: `Box<T>` - a pointer to a value on the heap.

- A pointer always has a known, fixed size.
- So if we wrap a recursive type in `Box`, Rust now sees:

```rust
struct Node {
    value: i32,
    next: Box<Node>, // fixed size pointer
}
```

- size of `Node` = size of `i32` + size of the pointer (`Box`)
- The actual `Node` that `Box` points to is on the heap, and the heap can grow arbitrarily — Rust doesn't need to know its full size at compile time.

Compound Data Types: Nested Structs

Solution: `Box<T>` - a pointer to a value on the heap.

- A pointer always has a known, fixed size.
- So if we wrap a recursive type in `Box`, Rust now sees:

```rust
struct Node {
    value: i32,
    next: Box<Node>, // fixed size pointer
}
```

where Box is defined as:

```rust
struct Box<T> {
  ptr: *mut T
 }
```

Compound Data Types: Nested Structs

Rust

```rust
// NOTE: pseudocode
impl Box<T> {
  pub fn new(value: T) -> Self {
    let pointer = malloc(size_of::<T>());
    // write the value there
    *pointer = value;
    // return a wrapped pointer
    return Box(pointer);
  }
}
```

Compound Data Types: Nested Structs

Rust

```rust
// NOTE: pseudocode
impl Box<T> {
  pub fn drop(&mut self) {
    // automatically frees
    free(self.pointer);
  }
}
```

Recap

- Types (mostly*) disappear at runtime
- But types do influence code generation: https://godbolt.org/z/d9ofb1YvK

Rust:

Force users to make conscious choices about potentially unsafe operations.

Rust:

Rust needs to know the size of every type at compile time because it stores variables directly on the stack.

Function Signatures

- Communicates behavior of code through types

```rust
fn is_even(value: i64) -> bool {...}
fn contains(haystack: &[i64], needle: i64) -> bool {...}

// can you guess the name?
fn ______(haystack: &[i64], needle: i64) -> usize {...}
```

Function Signatures

- Communicates behavior of code through types

Rust

```
fn is_even(value: i64) -> bool {...}
fn contains(haystack: &[i64], needle: i64) -> bool {...}

// can you guess the name?
type Index = usize;
fn ______(haystack: &[i64], needle: i64) -> Index {...}
```

Individual Assignment

- Graded for 50% of your grade
- DSMR Telegram parser
  - See https://cese.ewi.tudelft.nl for all info
- An assignment to get you familiar with the basics of Rust
- Don’t be scared about the sheer amount of documentation online, take it step by step
- Ask questions in the labs