RUST GRAZ – 02 DATA TYPES AND STD::ITER::ITERATOR

Lukas Prokop

July 25, 2019

DATA TYPES

PRIMITIVE DATA TYPES

INTEGERS

```
let a = 42;         // 32-bit signed int
let b = 0xDEAD;     // hexadecimal
let c = 0o4242;     // octal
let d = 0b101010; // binary
let e = 2_147_483_647; // _ for readability
```

per default: 32-bit signed integer (via RFC 212)

INTEGER TYPE SUFFIXES

```
let f = 0i64;   // 0 as signed 64-bit int
let g = 0u8;    // 0 as unsigned 8-bit int
let h = 0u16;   // 0 as unsigned 16-bit int
let i = 0u32;   // 0 as unsigned 32-bit int
let j = 0u64;   // 0 as unsigned 64-bit int
let k = 0u128;  // 0 as unsigned 128-bit int
```

Question: Give me usecases for 128-bit integers.

Explicit type declarations:

```
let l: u128 = 0;
```

OVERFLOW

```
fn main() {
    let f = 255u8;
    println!("{}", f + 1);
}
```

OVERFLOW

```
user@sys ~ % ./test
thread 'main' panicked at 'attempt to add with overflow', test2.rs:
note: Run with `RUST_BACKTRACE=1` environment variable to display a
101 user@sys ~ % RUST_BACKTRACE=1 ./test
thread 'main' panicked at 'attempt to add with overflow', test2.rs:
stack backtrace:
   0: std::sys::unix::backtrace::tracing::imp::unwind_backtrace
             at src/libstd/sys/unix/backtrace/tracing/gcc_s.rs:39
   1: std::sys_common::backtrace::_print
             at src/libstd/sys_common/backtrace.rs:71
…  9: test2::main
  10: std::rt::lang_start::{{closure}}
  11: std::panicking::try::do_call
… 15: main
  16: __libc_start_main
  17: _start
101 user@sys ~ %
```

WHAT HAPPENS IF AN INTEGER OVERFLOWS IN C?

WHAT HAPPENS IF AN INTEGER OVERFLOWS IN C?

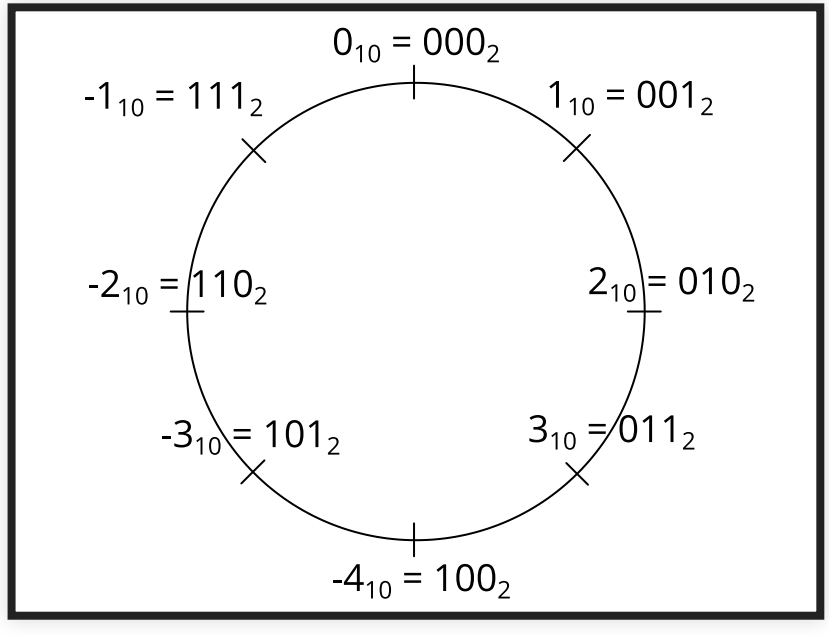
Undefined behavior for signed integers.
Modulo power of two for unsigned integers in C11.

Undefined behavior (in the sense of C) in rust should be restricted to `unsafe` (see the Rust book and reddit). But there is no formal model.

FUN FACT ABOUT INTEGER OVERFLOWS

In 2014, the music video for PSY's Gangnam Style received so many views on YouTube that it breached the maximum possible viewership number within a signed 32-bit integer. YouTube subsequently amended the maximum count to a 64-bit integer.

TWO'S COMPLEMENT



DIVISION BY ZERO

```
fn main() {
    println!("{}", 1/0);
}
```

```
error: attempt to divide by zero
 --> test2.rs:2:20
  |
2 |     println!("{}", 1/0);
  |                    ^^^
  |
  = note: #[deny(const_err)] on by default

error: reaching this expression at runtime will panic or abort
 --> test2.rs:2:20
  |
2 |     println!("{}", 1/0);
  |                    ^^^ attempt to divide by zero

error: aborting due to 2 previous errors
```

CASTING / PRIMITIVE TYPE COERSION

```
fn main() {
    let n = 255u8;
    println!("{}", n as u16 + 1);
}
```

- `as` keyword for casting
- necessary to prevent overflows and underflows

FLOATING POINT NUMBERS

The IEEE 754 “IEEE Standard for Floating-Point Arithmetic” standard defines:

- **arithmetic formats:** sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
- **interchange formats:** encodings that may be used to exchange floating-point data in an efficient and compact form

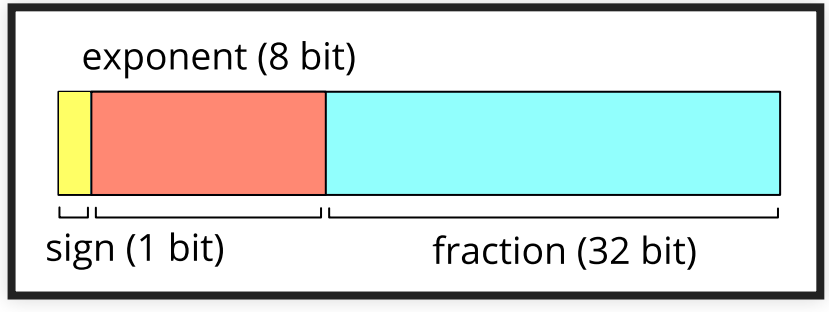
FLOATING POINT NUMBERS

- **rounding rules:** properties to be satisfied when rounding numbers during arithmetic and conversions
- **operations:** arithmetic and other operations (such as trigonometric functions) on arithmetic formats
- **exception handling:** indications of exceptional conditions (such as division by zero, overflow, etc.)

FLOATING POINT NUMBERS

```
fn main() {
    let f = 3141.5962f32;
    println!("{}", f);          // 3141.5962
    println!("{:.2}", f);       // 3141.60
    println!("{1:.0$}", 2, f); // 3141.60
    println!("{:e}", f);        // 3.1415962e3
    println!("{}", f as u32);  // 3141
}
```

FLOATING POINT NUMBERS



(1, 8, 23) for 32-bit floats.

(1, 11, 52) for 64-bit floats (“double”).

FLOATING POINT NUMBERS

sign	=	0	non-negative
exponent	=	-3	3 dec. positions dot offset RTL
fraction	=	31415962	

$$1 \cdot 3.1415962 \cdot 10^{-3}$$

But in **binary**!

`3141.5962 = 0 10001010 10001000101100111`

in binary.

FLOATING POINT NUMBER: INACCURACY

1./5 cannot be represented in binary accurately.
Approximations are used.

```
fn main() {
    let base = 1. / 5f32;
    for i in 1..=14u32 {
        let current_float = base * 5f32.powi(i as i32);
        let current_int = 5u32.pow(i - 1);
        println!("{}", current_float as i32 - current_int as i
    }
}
```

0 (11 times), then -1, then -1, then -21

FLOATING POINT NUMBER: INACCURACY

```rust
use std::f32::consts::PI;

fn main() {
    println!("{}", (PI / 2.).tan());
    // -22877334
}
```

FLOATING POINT NUMBER: SPECIAL VALUES

Two different zeros are specified.

```rust
fn main() {
    println!("{:032b}", 0f32.to_bits());
    // 00000000000000000000000000000000
    println!("{:032b}", (-0f32).to_bits());
    // 10000000000000000000000000000000
    println!("{}", -0f32 == 0f32);
    // true
}
```

FLOATING POINT NUMBER: SPECIAL VALUES

```rust
use std::f32;
fn main() {
    println!("{}", f32::NAN);           // NaN
    println!("{}", f32::INFINITY);      // inf
    println!("{}", f32::NEG_INFINITY);  // -inf
    let neg_inf = f32::NEG_INFINITY;
    println!("{}",
        f32::INFINITY == -neg_inf &&
        f32::INFINITY == neg_inf * neg_inf &&
        f32::INFINITY != f32::NAN &&
        !(f32::NEG_INFINITY < f32::NAN) &&
        f32::NAN != f32::NAN &&
        f32::NEG_INFINITY < 0f32
    );  // true
}
```

FLOATING POINT NUMBER: SPECIAL VALUES

```rust
use std::f32;

fn main() {
    println!("{}", 0. * f32::INFINITY); // NaN
    println!("{}", f32::INFINITY - 1.); // inf
    println!("{}", f32::INFINITY - f32::INFINITY); // NaN
    println!("{}", 2. % 0.); // NaN
}
```

Comparison with NaN is always false except for `!=` (always true). A total ordering is defined assuming `-0 == 0` are one value.

FLOATING POINT NUMBER: SPECIAL VALUES

IEEE 754 specifies two Not-A-Number (NAN) values. Signalling NaNs are inaccessible (like in most other languages).

```
fn main() {
    println!("{}", 0f32/0f32);
}
```

FLOATING POINT NUMBER: SPECIAL VALUES

- Signalling NaNs are used for debugging
- Quiet NaNs indicate arithmetic errors

```
use std::f32;

fn main() {
    let nan = f32::NAN;
    println!("{}", nan + nan * nan / 3.);
    // quiet until the return value is displayed
}
```

FLOATING POINT NUMBER: SOMETHING I CANNOT EXPLAIN

```rust
use std::f32;

fn main() {
    println!("{}", f32::NAN as u32);
}
```

FLOATING POINT NUMBER: SOMETHING I CANNOT EXPLAIN

```rust
use std::f32;

fn main() {
    println!("{}", f32::NAN as u32); // 0
}
```

DOCUMENTATION

For more, see the documentation:

- doc.rust-lang.org/std/primitive.f32
- doc.rust-lang.org/std/primitive.f64

OTHER PRIMITIVE TYPES

- char: `'x'`, `'λ'`
- str: `"Hello World"`, `r"raw string"`
- bool: `true`, `false`
- isize, usize: `1usize`, `42isize`

primitive collection types:

- arrays
- slices
- tuples

To be discussed in the following talks.

ITERATOR

`std::iter::Iterator trait` sounds super technical, but I want to talk about the opposite: the *concept* of `std::iter::Iterator trait`.

For now: traits are interfaces implemented by types/data structures. Details will follow soon.

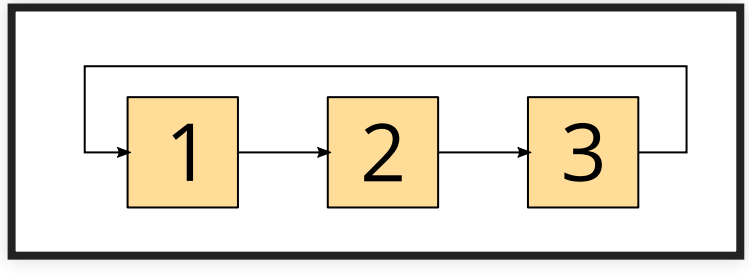
ITERATOR

Iterators are fundamental. Whenever we have a collection of values, we want to iterate them. And here we want to discuss *some* methods implemented by *every* iterator. Methods common for functional languages.

See doc.rust-lang.org/std/iter/trait.Iterator.html

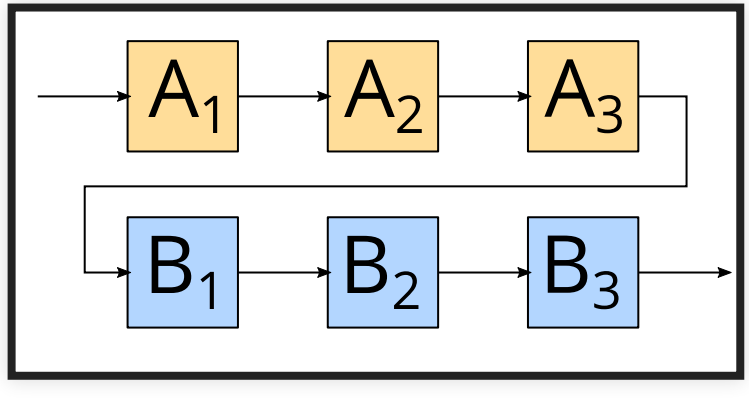
`cycle`

Given one iterator, we cycle through the elements.



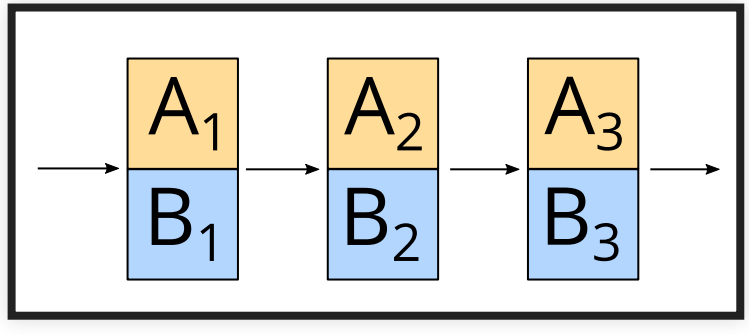
chain

Given two iterators, we iterate through the elements of the first iterator. Then we iterate through the elements of the second iterator.



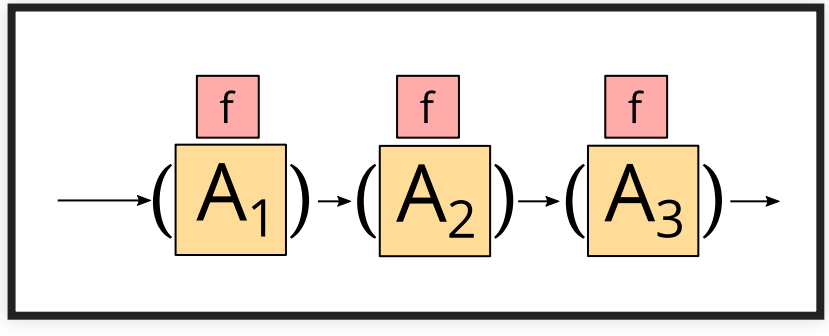
`zip`

Given n iterators, we iterate over the smallest number of elements. The iterated elements are n-tuples.



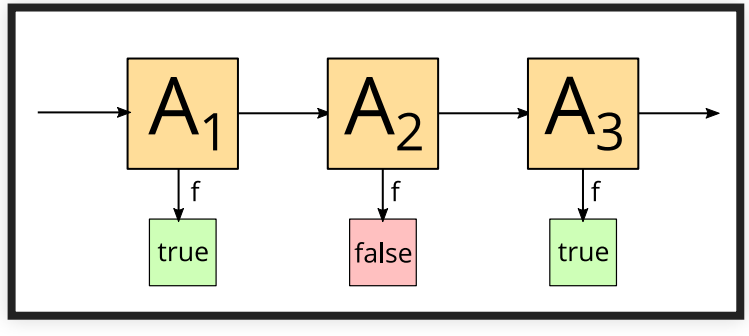
`map`

Given one iterator and a function, we apply a function to each element.



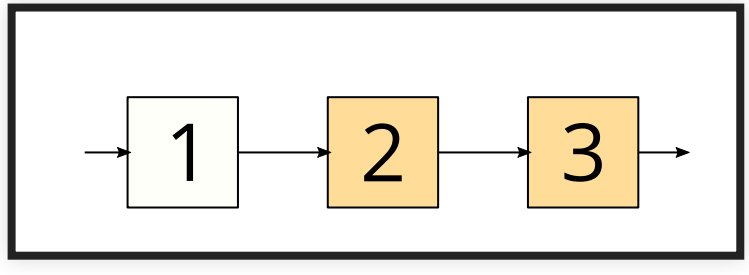
`filter`

Given an iterator and a function, we only return elements for which the function returns true.



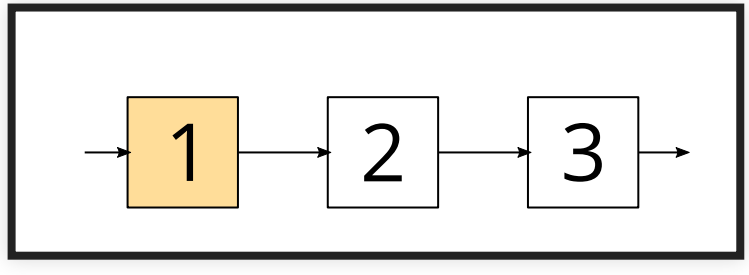
`skip`

Given an iterator and a number n, we skip the first n elements.



`take`

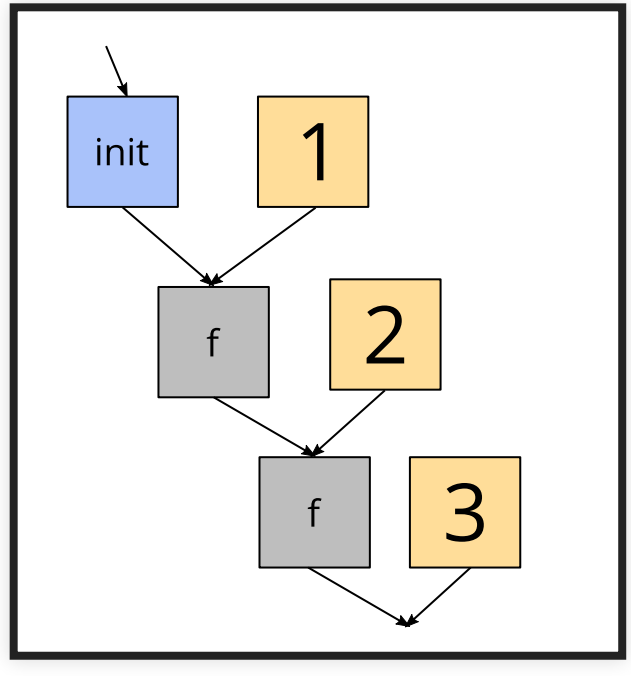
Given an iterator and a number `n`, we only return the first `n` elements.



`fold`

Given an iterator and an initial value, we take the first element and the initial value and supply it to the function. This intermediate value with the next value is supplied to the function. This continues. The last intermediate value becomes the final value.

Also called `reduce` or `inject`.



TOTAL ORDERING

`eq`, `ne`, `lt`, `le`, `gt`, `ge`

- The operations ≤ and ≥ define a partial ordering (reflexivity, antisymmetry, transitivity).
- Here we even have a total ordering (additionally connexity).

STD::ITER::ITERATOR

```
let a = [1, 2, 3];
let mut iter = a.iter();

assert_eq!(Some(&1), iter.next());
assert_eq!(Some(&2), iter.next());
assert_eq!(Some(&3), iter.next());
```

FINISH LINE

QUIZ

When shouldn't you use floating point numbers?

What is the default integer type? `let a = 42;`

Give an example of a rational number that cannot be represented accurately in IEEE 754

What does the `take` function do?

QUIZ

When shouldn't you use floating point numbers?

For monetary data.

What is the default integer type? `let a = 42;`

i32

Give an example of a rational number that cannot be represented accurately in IEEE 754

1 / 5

What does the `take` function do?

Iterator that returns the first n elements

NEXT TIME

Suggested topic

Control structures and functions

Topics to be done

traits, crates, borrowing, memory management, lifetimes, collections, strings, Unicode, regex, CSV/JSON/XML/YAML, i18n, logging, multithreading, data structures, documentation, cross compilation, concurrency, web assembly, …