UA Rust Conference 2024

July 27

online & offline

Learn. Develop. Discover.

All proceeds from the tickets will be donated to support **Ukraine**

near

Campus community

kumeka team

BOHEMIA AUTOMATION

Out of the Box Systems


ITERATE!
FASTER!

NICHOLAS CAMERON

ncameron.org

ITERATE! FASTER!

What are iterators?

ITERATE! FASTER!

What are iterators?

Are iterators fast?

ITERATE! FASTER!

What are iterators?

Are iterators fast?

How to use iterators efficiently?

ncameron.org

ITERATE! FASTER!

Understand performance characteristics of iterators

WHAT ARE ITERATORS?

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
```

[+] Show 75 methods

```
}
```

ncameron.org

```
fn foo(v: Vec<i32>) {
    v.into_iter()
        .map(|x| x + 10)
        .for_each(|x| println!("{x}"));
}
```

ncameron.org

```
fn foo(v: Vec<i32>) {
    v.into_iter()
        .map(|x| x + 10)
        .for_each(|x| println!("{x}"));
}
```

ncameron.org

```
fn foo(v: Vec<i32>) {
    v.into_iter()
        .map(|x| x + 10)
        .for_each(|x| println!("{x}"));
}
```

```rust
fn foo(v: Vec<i32>) {
    for x in v {
        let y = x + 10;
        println!("{y}");
    }
}
```

ncameron.org

```
fn foo(v: Vec<i32>) {
    let mut iter = v.into_iter();
    while let Some(x) = iter.next() {
        let y = x + 10;
        println!("{y}");
    }
}
```

ncameron.org

ARE ITERATORS FAST?

PERFORMANCE IS COMPLEX

PERFORMANCE IS COMPLEX

Context-dependent,

non-linear,

unpredictable

PERFORMANCE IS COMPLEX

Context-dependent, non-linear, unpredictable

Compiler optimisations

PERFORMANCE IS COMPLEX

Context-dependent, non-linear, unpredictable

Compiler optimisations

Modern CPUs

ncameron.org

WE DON’T COUNT OPCODES ANY MORE

ncameron.org

WE DON'T COUNT OPCODES ANY MORE

(Random)
memory
access

WE DON'T COUNT OPCODES ANY MORE

(Random) memory access

Allocation

ncameron.org

WE DON'T COUNT OPCODES ANY MORE

(Random) memory access

Allocation

Branching

ncameron.org

ARE ITERATORS FAST?

YES

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
```

[+] Show 75 methods

```
}
```

ncameron.org

CONTRACT

Creation is trivial

Iteration is trivial

CREATION IS TRIVIAL

- Does not allocate
- Zero-copy

CREATION IS TRIVIAL

- Does not allocate
- Zero-copy
- Does not iterate
- Does no pre-computation

CREATION IS TRIVIAL

- Does not allocate
- Zero-copy
- Does not iterate
- Does no pre-computation

- Corollary: iterator adapters must be lazy

```
fn foo(v: Vec<i32>) {
    v.into_iter()
        .map(|x| x + 10)
        .for_each(|x| println!("{x}"));
}
```

ncameron.org

```
fn foo(v: Vec<i32>) {
    v.into_iter()
        .map(|x| x + 10)
        .for_each(|x| println!("{x}"));
}
```

ncameron.org

```
fn foo(v: Vec<i32>) {
    v.into_iter()
        .map(|x| x + 10);
        // .for_each(|x| println!("{x}"));
}
```

ncameron.org

TRIGGERING EVALUATION

```
collect
for_each
fold
reduce
sum
count
…
```

CONTRACT

Creation
is trivial

Iteration
is trivial

ITERATION IS TRIVIAL

- Does not allocate
- No bounds checks

ITERATION IS TRIVIAL

- Does not allocate
- No bounds checks

- **BUT can still be significant**

PERFORMANCE CHARACTERISTICS

PERFORMANCE CHARACTERISTICS

Highly optimisable

```
fn foo() -> u64 {
    (0..u64::MAX).take(10).sum()
}
```

ncameron.org

```
fn foo() -> u64 {
    (0..u64::MAX).take(10).sum()
}
```

ncameron.org

```
fn foo() -> u64 {
    (0..u64::MAX).take(10).sum()
}
```

ncameron.org

```
foo:
    mov     eax, 45
    ret
```

ncameron.org

PERFORMANCE CHARACTERISTICS

Highly optimisable

No dynamic dispatch

PERFORMANCE CHARACTERISTICS

Highly optimisable

No dynamic dispatch

Low overhead

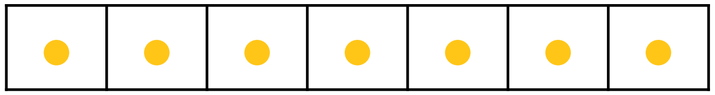
ARE ITERATORS FAST?

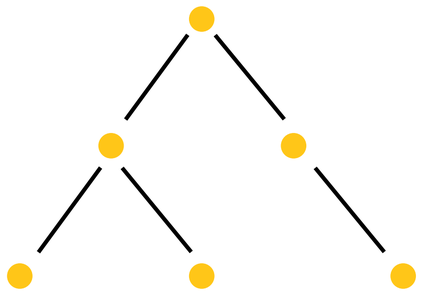
Similar performance to hand-written loops

ncameron.org

PERFORMANCE CHARACTERISTICS

Performance may depend on underlying collection





PERFORMANCE CHARACTERISTICS

Iteration by reference vs by value

`iter()`

`into_iter()`

HOW TO USE ITERATORS EFFICIENTLY?

USING ITERATORS

USING ITERATORS

- Avoid unnecessary evaluation
  - Especially `collect`ing

USING ITERATORS

- Avoid unnecessary evaluation
  - Especially `collecting`
- Prefer to pass iterators rather than collections
  - Use `impl Iterator`

USING ITERATORS

- Prefer `extend` to `append`
  - `<Vec as Extend>::extend(impl IntoIterator<Item=T>)`
  - `Vec::append(&mut Vec<T>)`
- Prefer `drain` to `split_off`
  - `drain(...) -> impl Iterator<T>`
  - `split_off(...) -> Vec<T>`

USING ITERATORS

- Prefer `len` to `count`

USING ITERATORS

- Beware of large accumulators in `fold`

WRITING ITERATORS

WRITING ITERATORS

- Follow the contract
  - Creation must be cheap
  - Iteration must be cheap

WRITING ITERATORS

- Override provided methods
  - Especially `fold`, `try_fold`, `advance_by`

WRITING ITERATORS

- Implement `size_hint`

SUMMARY

SUMMARY

Iterators are fast

Lazy semantics

By reference
By value

SUMMARY

Be aware of eagerly evaluating

Be aware of underlying collections

Be aware of passing large values

ncameron.org

SUMMARY

Creation and iteration must be cheap

Override provided methods

`size_hint`

ncameron.org

THANK YOU!

ncameron.org

ncameron.org/perf-course

UA Rust Conference 2024

July 27

online & offline

Learn. Develop. Discover.

All proceeds from the tickets will be donated to support **Ukraine**

near

Campus community

kumeka team

BOHEMIA AUTOMATION

Out of the Box Systems