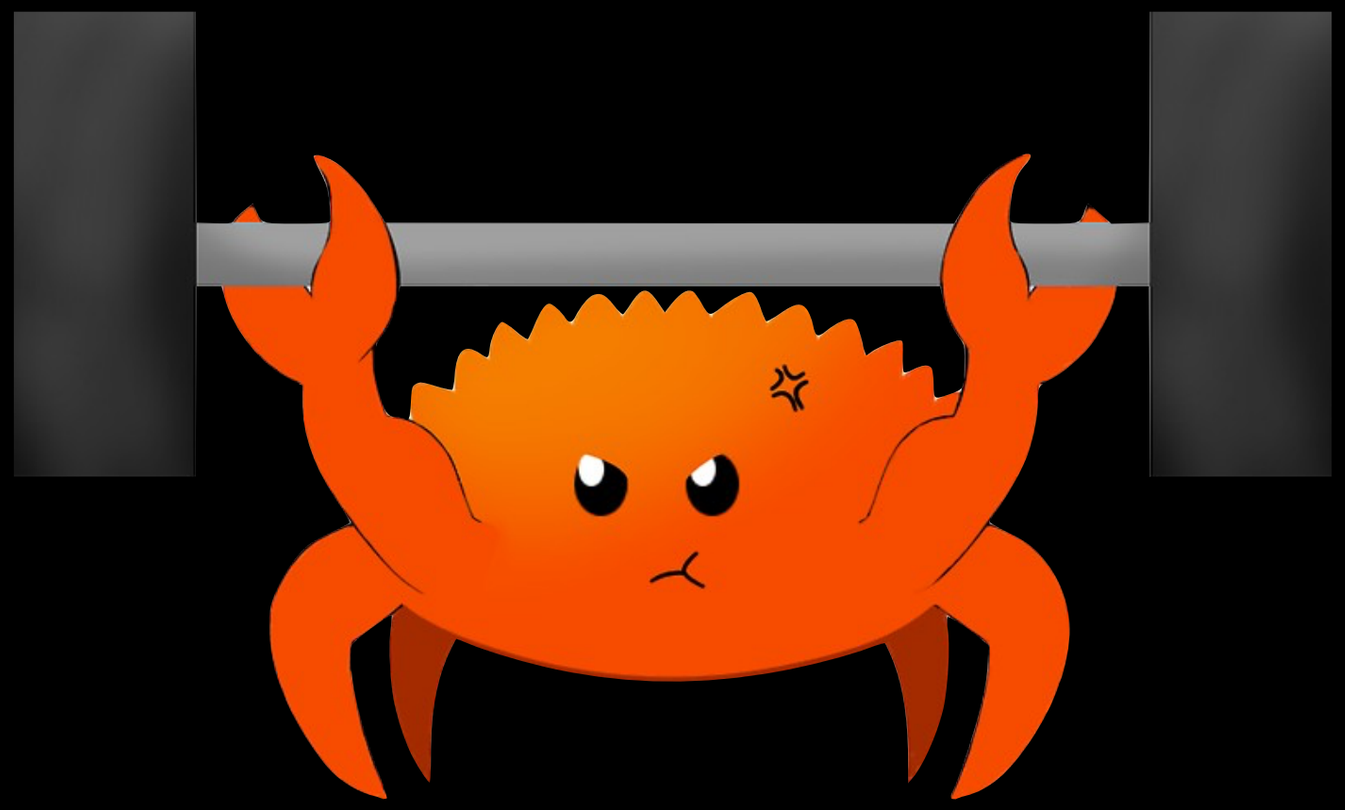
# The Beauty of Rust

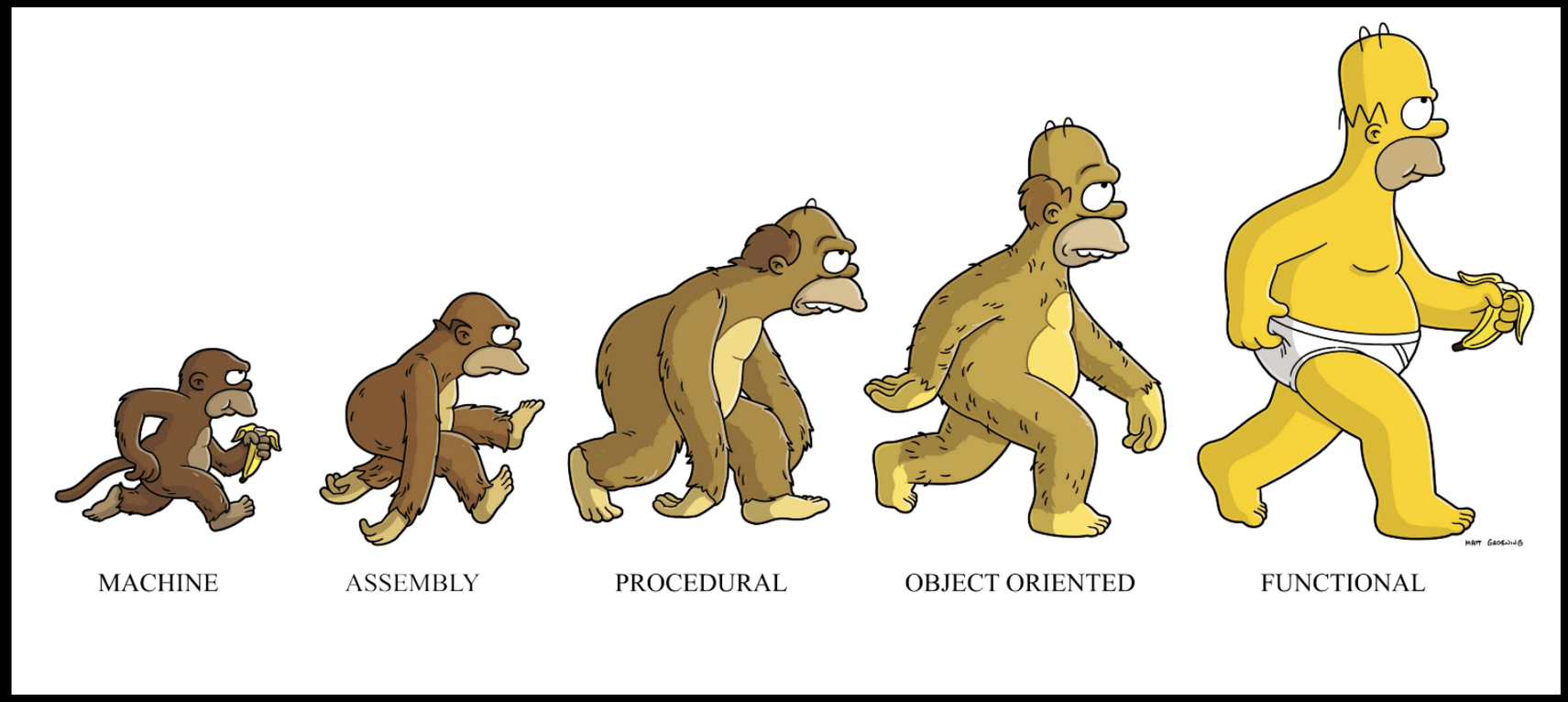
## Iterators and Closures



Peter Zdankin

# Programming Paradigms

- Programming Languages have gone a long way
- Assembly
- C
- Java
- Haskell




- Why did I pick these languages as examples?

# “GO TO statement considered harmful”

**Edgar Dijkstra**

# `--deep` Considered Harmful

This thread has been locked by a moderator.

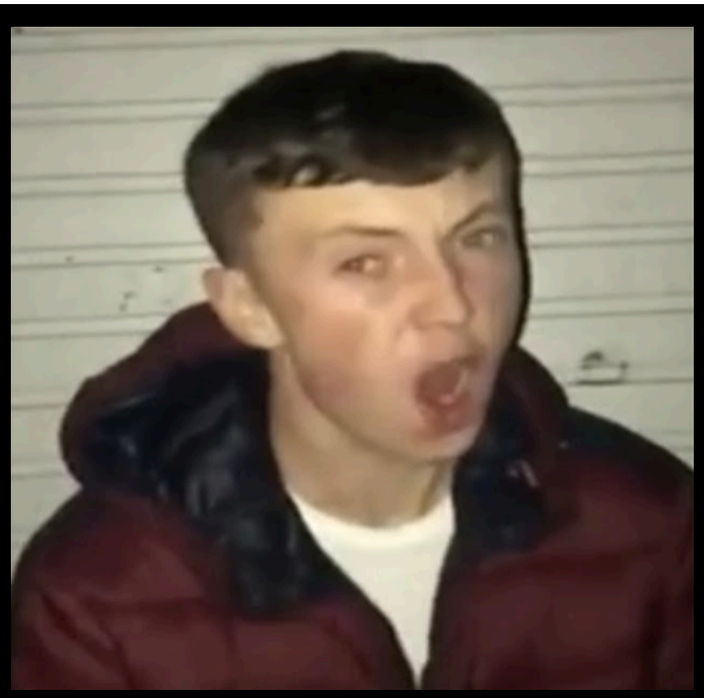
2.4k

Many of the notarisation and Gatekeeper problems I see are caused by folks signing their product using the `--deep` option. While that can work in some circumstances, I generally recommend against it. There are two issues with `--deep`:

- It applies the same code signing options to every code item that it signs, something that's not appropriate in general. For example, you might have an app containing a nested command-line tool, where the app and the tool need different entitlements. The `--deep` option will apply the same entitlements to both, which is a serious mistake.
- It only signs code that it can find, and it only finds code in nested code sites. If you put code in a place where the system is expecting to find data, `--deep` won't sign it.

The first issue is fundamental to how `--deep` works, and is the main reason you should not use it. The second issue is only a problem if you don't follow the rules for nesting code and data within a bundle, as documented in Placing Content in a Bundle.

**Some weirdo on the apple forum**



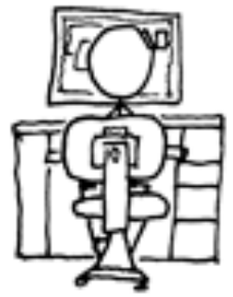
# Spaghetti Code Era
## (Non-structured imperative)

- Do you know goto?
- Jump (un)conditionally to a label
- Control flow unpredictable
- Debugging terribly complicated
- (C still allows goto)
- (Using goto in your C code gets you in trouble with most people here)

I COULD RESTRUCTURE THE PROGRAM'S FLOW

OR USE ONE LITTLE 'GOTO' INSTEAD.

EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?

# Imperative/Procedural Era

- Goto was eliminated
- Functions must be used
- Jump in a function, do stuff, jump back
- If, while, do, for, switch

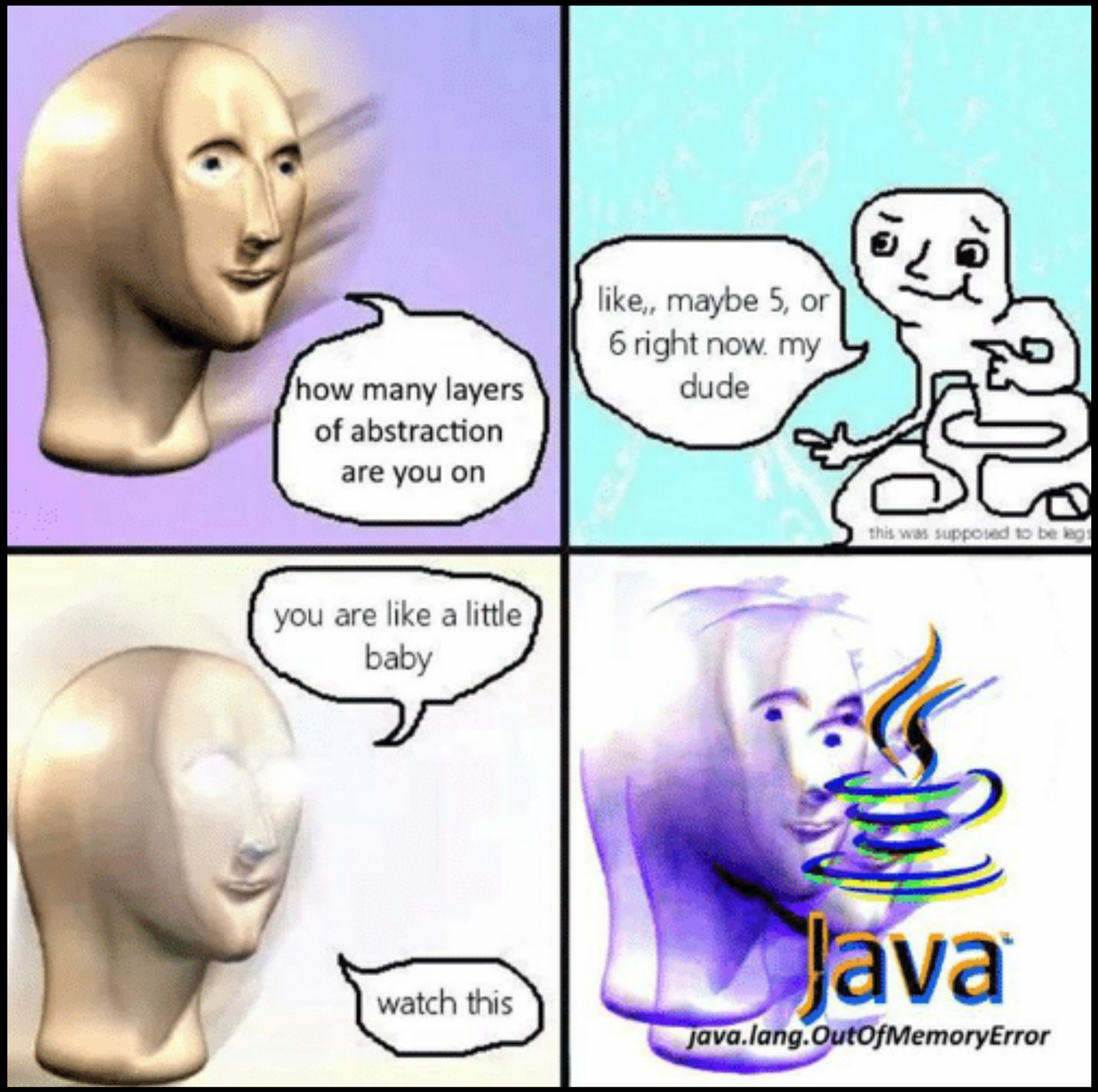
- No real object orientation, encapsulation and modularization (yet)

“Yeah, sure, you can do OOP in C.. I guess? But there are just like 5 people that do so. 2 of them are Lukas”

**Me**

“The object-oriented version of spaghetti code is, of course, ‘lasagna code’. Too many layers

**Roberto Waltman**


how many layers of abstraction are you on
like,, maybe 5, or 6 right now. my dude
this was supposed to be leg
you are like a little baby
watch this
Java
java.lang.OutOfMemoryError

# Object Orientation

- Golden Hammer of CS Lectures
- They teach it, regardless if it fits the purpose
- Can be useful if you program a zoo, UI Frameworks or games
- You can easily end up with ridiculously complex state machines

# Functional Programming

- Global State is a mess
- By having pure functions, you have side effect free code
- Pure functions can be tested easily
- Add functions as parameters
- Very high level

- How does this lead to Rust?

# Functional Programing in Rust

- Rust promises zero-cost abstractions
- Make use of functional aspects, like higher order functions, first class functions and an immutable global state
- All this in a system programming language?
- Does this look as bad as in C++?

# Closures

- Anonymous functions
- Closures can “capture” variables from the outside

- “What should the thread do, if started?”
- “What should the Webserver serve to a client?”

```rust
fn  add_one_v1    (x: u32) -> u32 { x + 1 }

let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

# Function vs Closure

- Closures can capture values

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }

    let y = 4;

    assert!(equal_to_x(y));
}
```

Doesn’t compile

# Capturing Variables

- Closures can access variables from the scope they are defined in
- Capture everything in | |
- Interesting for multithreading
- Values are borrowed or moved
- move | x | or | move x | moves ownership of x into the closure

```rust
fn main() {
    let mut x = 4;
    let equal_to_x = |z| z == x;
    let y = 4;

    assert!(equal_to_x(y));
}
```

This works

```rust
fn main() {
    let mut x = 4;
    let equal_to_x = |z| z == x;
    x += 1;
    let y = 4;

    assert!(equal_to_x(y));
}
```

This doesn’t

# Move Ownership

```
fn main() {
    let x = vec![1, 2, 3];

    let equal_to_x = move |z| z == x;

    println!("can't use x here: {:?}", x);
    let y = vec![1, 2, 3];
    assert!(equal_to_x(y));
}
```

Doesn’t compile

# Iterators

```rust
pub trait Iterator {
    type Item;

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

    // methods with default implementations elided
}
```

# Perform something on a sequence of somethings

```
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {}", val);
    }
```

So high level!

# De-sugaring

/Users/peterzdankin/Projects/rust/joy

```
#[prelude_import]
use ::std::prelude::v1::*;
#[macro_use]
extern crate std;

fn main() {
            let v1 =

                <[_]>::into_vec(box [1, 2, 3]);
            let v1_iter = v1.iter();
            {
                let _result =
                    match ::std::iter::IntoIterator::into_iter(v1_iter) {
                        mut iter =>
                        loop  {
                            let mut __next;
                            match ::std::iter::Iterator::next(&mut iter) {
                                ::std::option::Option::Some(val) =>
                                __next = val,
                                ::std::option::Option::None => break ,
                            }
                            let val = __next;
                            {
                                {
                                    $crate::io::_print(<$crate::fmt::Arguments>::new_v1(&["Got: ",
                                                                                         "\n"],
                                                                                        &match (&val,)
                                                                                             {
                                                                                             (arg0,)
                                                                                             =>
                                                                                             [<$crate::fmt::ArgumentV1>::new(arg0,
                                                                                                                             $crate::fmt::Display::fm
t)],
                                                                                         }));
                                };
                            }
                        },
                    };
                _result
            }
:
```

```rust
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

# Functions on iterators consume the iterator

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

Can’t use v1_iter after this

# Iterators that produce other iterators

- Iterators don’t do anything, unless they are consumed

```rust
let v1: Vec<i32> = vec![1, 2, 3];
v1.iter().map(|x| x + 1);
```

Nothing is done here

```rust
    let v1: Vec<i32> = vec![1, 2, 3];

    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
```

Type required

Collect does magic things

```rust
    assert_eq!(v2, vec![2, 3, 4]);
```

# Filter

- Filter all elements that fulfill a property

```
fn main() {
    let x: Vec<_> = (1..20).into_iter()
            .filter(|x| x % 2 == 0)
            .collect();
    println!("{:?}",x);
}
```

# Iterator DIY

# Our Struct

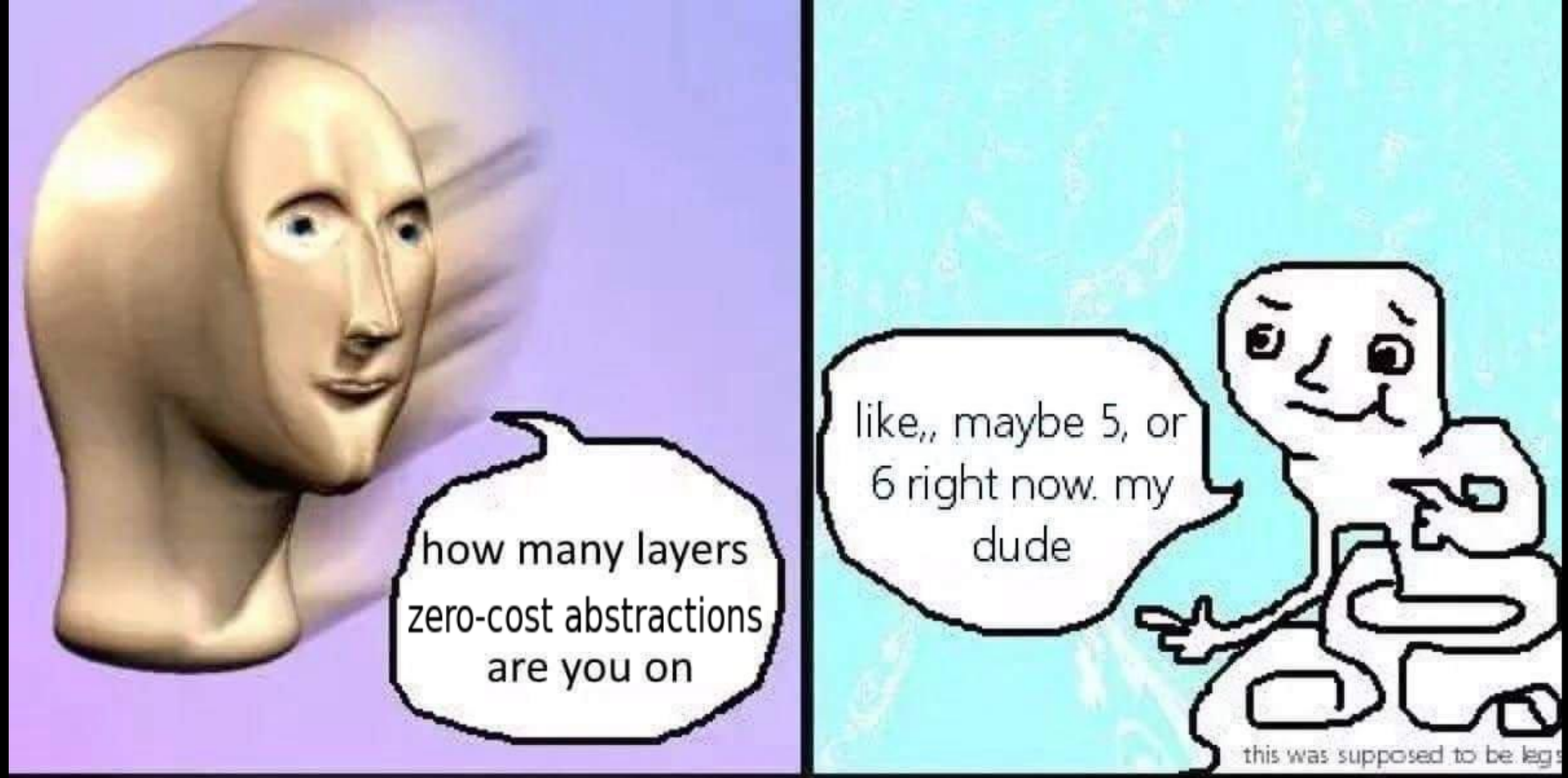
```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

# Our Impl Iterator

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

```rust
#[test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

Lets rock


how many layers
zero-cost abstractions
are you on
like,, maybe 5, or
6 right now. my
dude
this was supposed to be leg


you are like a little
baby
Downloading hyper v0.11.2
Downloading mime v0.3.4
Downloading unicase v2.0.0
Downloading tokio-service v0.1.0
Downloading time v0.1.38
Downloading bytes v0.4.5
Downloading tokio-core v0.1.9
Downloading tokio-proto v0.1.1
Downloading tokio-io v0.1.3
Downloading base64 v0.6.0
Downloading language-tags v0.2.2
Downloading percent-encoding v1.0.0
Downloading log v0.3.8
Downloading httparse v1.2.3
Downloading futures v0.1.16
Downloading futures-cpupool v0.1.6
Downloading rustc_version v0.1.7
watch this
BoxFuture<T, E> = Box<Future<Item = T, Error = E> + Send>
Trait futures::sink::Sink
type Future: Future<Item = Self::Response, Error = Self::Error>;
fn call(&self, req: Self::Request) -> Self::Future;
fn start_send(
&mut self,
item: Self::SinkItem
Panics
Stream<Error = Error>
Trait tokio_proto::streaming::pipeline::ServerP
fn serve<S>(&self, new_service: S)
where
S: NewService + Send + Sync + 'static,
S::Instance: 'static,
P::ServiceError: 'static,
P::ServiceResponse: 'static,
P::ServiceRequest: 'static,
S::Request: From<P::ServiceRequest>,
S::Response: Into<P::ServiceResponse>,
S::Error: Into<P::ServiceError>,
Start up the server, providing the given service on it.
pub trait ServerProto<T: 'static>: 'static {
type Request: 'static;
type RequestBody: 'static;
type Response: 'static;
type ResponseBody: 'static;
type Error: From<Error> + 'static;
type Transport: Transport<Item = Frame<Self::Request, Self::RequestBody, Self::Error>,
SinkItem = Frame<Self::Response, Self::ResponseBody, Self::Error>>;
type BindTransport: IntoFuture<Item = Self::Transport, Error = Error>;
fn bind_transport(&self, io: T) -> Self::BindTransport;
}
+ 'static
-0.1.9
In addition to those basics, the builder provides some additional configuration, which is expected to grow over time.
The Service trait is a simplified interface making it easy to write network applications in a modular and reusable way, decoupled from the underlying protocol. It is one of
UnboundedReceiver<Result<(R, Sender<Result<S, E>>)>>;
Versions
0.1.9
0.1.8
0.1.7
0.1.6
where
I: IntoIterator,
Self::Item: PartialOrd<<I as IntoIterator>::Item>,
application framework for rapid development and highly scalable production deployments of clients and servers.
a PollEvented gets a little interesting when working with an arbitrary instance of mio::Ready

# Exercise *

- You have the numbers 0 to 5 and 1 to 5
- Match the numbers pairwise (0,1) (1,2) (2,3) (3,4) (4,5) (5, )
- Multiply each pair
- Find all products that are divisible by 3
- Sum these ones up

```rust
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
                                 .map(|(a, b)| a * b)
                                 .filter(|x| x % 3 == 0)
                                 .sum();
    assert_eq!(????, sum);
}
```

```rust
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
                                 .map(|(a, b)| a * b)
                                 .filter(|x| x % 3 == 0)
                                 .sum();
    assert_eq!(18, sum);
}
```

```
0  *  1     0
1  *  2     2
2  *  3     6      6
                    +
3  *  4     12     12
4  *  5     20
5
```

# What is the price of iterators?

- Iterators are slightly faster in a benchmark
- This doesn’t mean they are always faster
- They are compiled to roughly the same code
- High level abstraction -> efficient low level code

```
test bench_search_for  ... bench:  19,620,300 ns/iter (+/- 915,700)

test bench_search_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

# Exercises

- Print an iterator that only prints odd numbers
- Make an iterator that iterates over multiples of 7 and print each all values that are divisible by 5
- Make an iterator that iterates over multiples of 2, zip it with an iterator that iterates over multiples of 7, print the sum of the pair
- Create a closure that multiplies a number by 2, let an iterator map each number on that closure