```rust
// There will be code, here's how large the font will be.
fn main() {
    println!("Hello, world!");
}
```

Me

Hugo of all trades

- Originally from Sweden, called Edinburgh home for almost 10 years
- Rust since 2017, professionally at Lookback since 2021

Background - Coloured functions

"What color is your function?" by Bob Nystrom

- `function blue() {}`
- `async function red() {}`

```
async function red() {
    const blueResult = blue();
    return await otherRed(blueResult);
}
```

We can call blue functions from red functions.

```
function blue() {
    // Missing `await`
    const redResult = red();
    return otherBlue(redResult);
}
```

We cannot call red functions from blue functions.

```
async function noLongerBlue() {
    const redResult = await red();
    return otherBlue(redResult);
}
```

Must convert blue to a red function to call red. Red functions spread like wildfire.

Enough Javascript, let's talk Rust

Background - Coloured functions

"Let futures be futures" by withoutboats

- `fn blue() {}`
- `async fn red() {}`
- `fn green() {}`

Background - Coloured functions

Caller 饋 / Callee →	Blue	Red	Green
Green	No	No	Yes
Blue	Yes	No	Yes
Red	No	Yes	Yes

Background - Coloured functions

I lied

- async fn red() {}
- async fn teal() {}
- async fn orange() {}
- async fn purple() {}

Background - Coloured functions

- `async fn read(stream: &tokio::net::TcpStream) {}`
- `async fn read(stream: &glommio::net::TcpStream) {}`
- `async fn read(stream: &async_net::TcpStream) {}`
- `async fn read(stream: &tokio_uring::net::TcpStream) {}`

Sans-IO

Sans-IO - What is it?

- Originally from the Python world.
- Make all functions green.
- Leverage inversion of control.
- Not just for IO.
- ping(8)

```rust
struct Ping {
    // Attributes omitted
}

impl Ping {
    fn handle_input(&mut self, input: Input) -> Output {
        todo!()
    }
}
```

```rust
enum Input<'b> {
    /// Some input data and the current time.
    Data((&'b [u8], Instant)),
    /// Timeout reached, here's the current time.
    Time(Instant),
}

enum Output {
    /// Send this data to the network.
    Send(Vec<u8>),
    /// Ask me again at this time.
    Timeout(Instant),
    /// An event occurred.
    Event(Event),
}

enum Event {
    Result {
        seq_num: u16,
        rtt: Option<Duration>,
    }
}
```

```rust
impl Ping {
    fn handle_input(&mut self, input: Input) -> Result<Output, Error> {
        match input {
            Input::Datagram(buf, now) => self.hande_datagram(buf, now),
            Input::Time(now) => Ok(self.handle_timeout(now)),
        }
    }
}
```

```rust
fn main() -> Result<()> {
    let socket = create_socket()?;
    let addr: SocketAddrV4 = "8.8.8.8:0".parse()?;
    let sock_addr = SockAddr::from(addr);
    // Construct the sans-IO core struct, Ping.
    let mut ping = Ping::new(*addr.ip(), Duration::from_millis(1000));

    let mut buf: [MaybeUninit<u8>; 1500] = [MaybeUninit::uninit(); 1500];
    let mut timeout_until = Instant::now();
    let mut last_recv: Option<Input<'_>>;

    loop {
        // 1. Read from the socket or timeout
        let timeout = (timeout_until - Instant::now()).max(Duration::from_millis(1));
        socket.set_read_timeout(Some(timeout))?;
        last_recv = read_from_socket(&socket, &mut buf)?;

        // 2. Handle the input
        timeout_until = handle_input(&mut last_recv, &mut ping, &socket, addr, &sock_addr)?;
    }
}
```

```rust
fn read_from_socket<'b>(
    socket: &Socket,
    buf: &'b mut [MaybeUninit<u8>; 1500],
) -> Result<Option<Input<'b>>> {
    match socket.recv_from(buf) {
        Ok((len, _)) => {
            // SAFETY: We read at least `len` bytes into `buf`, so it's safe to read from it.
            let read = unsafe { slice_assume_init_ref(&buf[..len]) };
            Ok(Some(Input::Datagram(read, Instant::now())))
        }
        Err(e) => {
            if e.kind() == std::io::ErrorKind::WouldBlock {
                Ok(Some(Input::Time(Instant::now())))
            } else {
                return Err(e.into());
            }
        }
    }
}
```

```rust
fn handle_input(
    last_recv: &mut Option<Input<'_>>,
    ping: &mut Ping,
    socket: &Socket,
    addr: SocketAddrV4,
    sock_addr: &SockAddr,
) -> Result<Instant> {
    loop {
        let input = last_recv
            .take()
            .unwrap_or_else(|| Input::Time(Instant::now()));
        let output = ping.handle_input(input)?;

        match output {
            ping::Output::Event(event) => {
                handle_event(event, addr);
            }
            ping::Output::Send(vec) => {
                socket.send_to(&vec, sock_addr)?;
            }
            ping::Output::Timeout(instant) => {
                break Ok(instant);
            }
        }
```

```rust
fn main() -> Result<()> {
    let socket = create_socket()?;
    let addr: SocketAddrV4 = "8.8.8.8:0".parse()?;
    let sock_addr = SockAddr::from(addr);
    // Construct the sans-IO core struct, Ping.
    let mut ping = Ping::new(*addr.ip(), Duration::from_millis(1000));

    let mut buf: [MaybeUninit<u8>; 1500] = [MaybeUninit::uninit(); 1500];
    let mut timeout_until = Instant::now();
    let mut last_recv: Option<Input<'_>>;

    loop {
        // 1. Read from the socket or timeout
        let timeout = (timeout_until - Instant::now()).max(Duration::from_millis(1));
        socket.set_read_timeout(Some(timeout))?;
        last_recv = read_from_socket(&socket, &mut buf)?;

        // 2. Handle the input
        timeout_until = handle_input(&mut last_recv, &mut ping, &socket, addr, &sock_addr)?;
    }
}
```

```rust
fn main() -> Result<()> {
    let socket = create_socket()?;
    let addr: SocketAddrV4 = "8.8.8.8:0".parse()?;
    let sock_addr = SockAddr::from(addr);
    // Construct the sans-IO core struct, Ping.
    let mut ping = Ping::new(*addr.ip(),
Duration::from_millis(1000));

    let mut buf: [MaybeUninit<u8>; 1500] =
[MaybeUninit::uninit(); 1500];
    let mut timeout_until = Instant::now();
    let mut last_recv: Option<Input<'_>>;

    loop {
        // 1. Read from the socket or timeout
        let timeout = (timeout_until -
Instant::now()).max(Duration::from_millis(1));
        socket.set_read_timeout(Some(timeout))?;
        last_recv = read_from_socket(&socket, &mut
buf)?;

        // 2. Handle the input
        timeout_until = handle_input(&mut
last_recv, &mut ping, &socket, addr, &sock_addr)?;
```

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let addr: SocketAddrV4 = "8.8.8.8:0".parse()?;
    let mut socket = Socket::new_icmp_v4(addr)?;
    // Construct the sans-IO core struct, Ping.
    let mut ping = Ping::new(*addr.ip(),
Duration::from_millis(1000));

    let mut buf = [0u8; 1500];
    let mut timeout_until = Instant::now();
    let mut last_recv: Option<Input<'_>>;

    loop {
        // 1. Read from the socket or timeout
        let timeout = (timeout_until -
Instant::now()).max(Duration::from_millis(1));
        last_recv = Some(read_from_socket(&mut
socket, &mut buf, timeout).await?);

        // 2. Handle the input
        timeout_until = handle_input(&mut
last_recv, &mut ping, &mut socket, addr).await?;
    }
}
```

```rust
#[test]
fn test_starts_by_returning_echo() {
    let (mut ping, now) = setup();
    let input = Input::Time(now);
    let output = ping.handle_input(input).unwrap();
    assert!(matches!(output, Output::Send(_)));
    let data = output.unwrap_send();
    validate_echo_request(&data);
}
```

```rust
#[test]
fn test_handles_response() {
    let (mut ping, now) = setup();
    let input = Input::Time(now);
    let output = ping.handle_input(input).unwrap();
    assert!(matches!(output, Output::Send(_)));
    let reply = make_echo_reply(0x1337, 0);

    let input = Input::Datagram(&reply, now + ms(23));
    let output = ping
        .handle_input(input)
        .expect("should handle the response");
    let event = output.unwrap_event();
    let (seq_num, rtt) = event.unwrap_result();
    assert_eq!(seq_num, 0);
    assert_eq!(rtt, Some(ms(23)));
}
```

```rust
#[test]
fn test_handle_response_timeout() {
    let (mut ping, now) = setup();
    let input = Input::Time(now);
    let output = ping.handle_input(input).unwrap();
    assert!(matches!(output, Output::Send(_)));

    let input = Input::Time(now + ms(999));
    let output = ping.handle_input(input).unwrap();
    assert!(
        matches!(output, Output::Timeout(_)),
        "No response or timeout after 999ms"
    );

    let input = Input::Time(now + ms(1000));
    let output = ping.handle_input(input).unwrap();
    assert!(
        matches!(output, Output::Send(_)),
        "Should send another ping first"
    );

    let input = Input::Time(now + ms(1000));
    let output = ping.handle_input(input).unwrap();
    let event = output.unwrap_event();
```

```rust
#[derive(Debug)]
pub enum Output<'b> {
    /// An event that happened
    Event(Event),
    /// Some data to send
    Send(&'b [u8]),
    /// We don't need to do anything until more data is received or the timeout is reached
    Timeout(Instant),
}

pub trait Context {
    fn buffer(&mut self, size: usize) -> &mut [u8];
}

pub fn handle_input<'s: 'b, 'b>(
    &'s mut self,
    input: Input,
    context: &'b mut impl Context,
) -> Result<Output<'b>, Error> {
    // omitted
}

```

I need you to make sans-IO crates

Cool stuff

sans-io crates

- quinn- QUIC/HTTP3 implementation https://github.com/quinn-rs/quinn
- str0m - WebRTC implementation https://github.com/algesten/str0m/
- librice - ICE implementation https://github.com/ystreet/librice
- rc-zip - ZIP file handling https://github.com/bearcove/rc-zip

Related interesting stuff

- Coroutines - To build sans-IO state machines.
- Abusing Futures - To build sans-IO state machines with async/await.
- Effects - Powerful abstractions for being generic over sync/async among other things.
- Keyword generics initiative Upcoming proposal to allow being generic of sync/async in Rust.