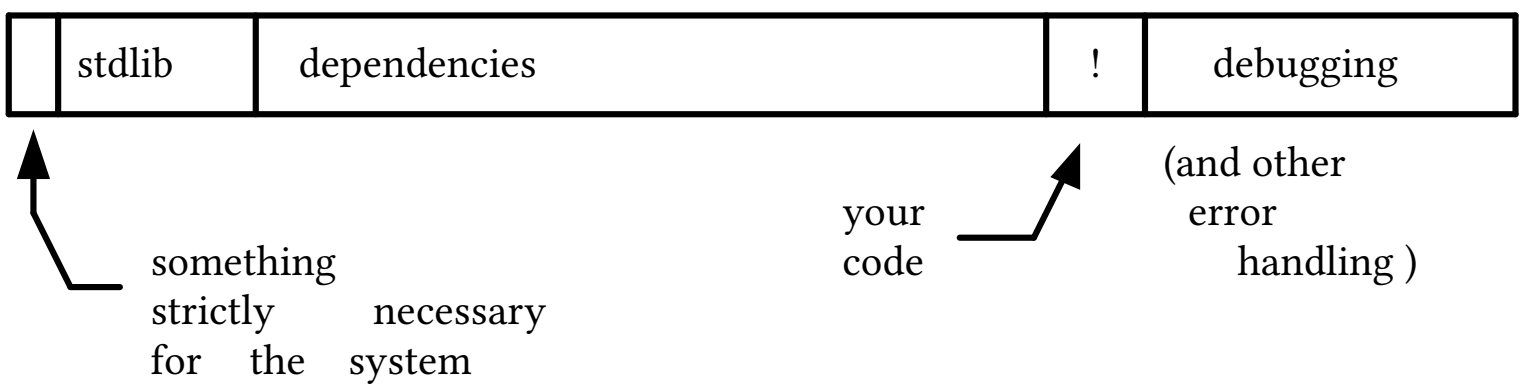
Size of Rust binaries

Link to this presentation:

Link to the code samples

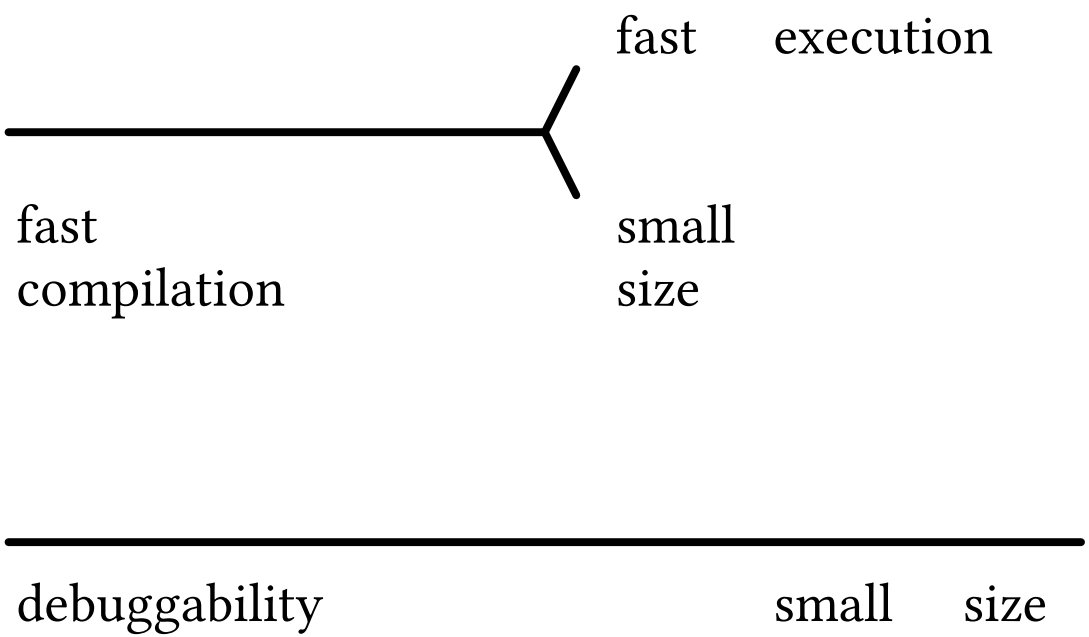
Part 1: bloated programs

Typical bloated program




```
size = C + k * project_size
```

Tradeoffs


fast execution
fast compilation
small size
debuggability
small size

Reasons why big

```
size = C + k * project_size
```

- No dynamic linking
- Monomorphisation
- Debugging features
- Stdlib algorithms

Reasons why big: No dynamic linking

Typical Rust-generated executable or cdylib is mostly **self-sufficient** and reaches for **OS-supplied libraries** only when really needed.

Reasons why big: Monomorphisation

Static dispatch is popular and leads to better performance, but may bloat up the code.

Intruducting otherwise unneeded **`dyn`** can help with battling the bloat. There are crates to automate that in some cases.

Reasons why big: Debugging features

Rust executables typically contain **helpful messages** to be displayed in case of panic.

Reasons why big: stdlib algorithms

Typically algorithms in stdlib (e.g. sorting) are **aimed at performance**, not compactness.

Reasons for pursuing minsized

- Getting ready for the **embedded** with access to hold tools
- **Security** review of dependencies
  - Less bytes - **less code**
  - **Gamified** - nice to see the number shrink as you go though the deps.
- **Aesthetical** resons: bloat=bad.
- **Educational** exercise.

Part 2A: Shrinking challenge (normal measures)

Sample project

Let's make a **TCP forwarder** as a sample project.

- userspace
- async
- single threaded
- CLI arguments with a help message

Note: used `rustc 1.84.0-nightly (bc5cf994d 2024-11-05)`

Basic version source code (part 1)

```
use clap::Parser;    use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
#[derive(Parser)]    struct Args {
    listen_addr: SocketAddr,
    connect_addr: SocketAddr,
}
#[tokio::main(flavor="current_thread")]
async fn main() -> anyhow::Result<()> {
    ...
    Ok(())
}
```

Basic version source code (part 2)

```
let Args { listen_addr, connect_addr } = Args::parse();
let l = TcpListener::bind(listen_addr).await?;
while let Ok((mut s, ..)) = l.accept().await {
    tokio::spawn(async move {
        let mut c = TcpStream::connect(
                                connect_addr).await?;
        tokio::io::copy_bidirectional(
                               &mut s, &mut c).await?;
        Ok::<(),anyhow::Error>(())
    });
}
```

Basic version's metrics

`cargo build --release --target=x86_64-unknown-linux-gnu` + `strip`

target	size
amd64 linux	1308k
i686 musl	1304k
arm64 android	1180k
windows64	2212k
arm64 mac	1172k

Other languages (for a comparison)

target	Go size	Haskell size
amd64 linux	2136k	2976k
i686 linux	1968k	
arm64 android	2372k	
windows64	2216k	
arm64 mac	2168k	

With **tinygo** and appropriate `Netdev` I expect 100-200k to be reachable.

“Free beer” optimisations

- LTO
- codegen units = 1
- opt-level = “s”

“Free beer” reduction

target	earlier size	new base size	reduction
amd64 linux	1308k	848k	35.2%
i686 musl	1304k	908k	30.4%
arm64 android	1180k	776k	34.2%
windows64	2212k	772k	65.1%
arm64 mac	1172k	764k	34.8%

Deeper compilation mode changes

Lossy.

```
cargo build --profile=myrelease \
    -Z build-std=panic_abort,std \
    -Z build-std-features=panic_immediate_abort \
    --target=x86_64-unknown-linux-gnu
```

- no unwinding
- build_std
- panic_immediate_abort

Other notable unstable settings

- `-Z build-std-features=optimize_for_size`
- `RUSTFLAGS="-Zlocation-detail=none"`
- `RUSTFLAGS="-Zfmt-debug=none"`
- `-Zvirtual-function-elimination`
- `-Zno-jump-tables`

Lean build results

target	base size	lean size	reduction
amd64 linux	848k	400k	52.8%
i686 musl	908k	484k	46.7%
arm64 android	776k	416k	46.4%
windows64	772k	448k	42.0%
arm64 mac	764k	360k	52.9%

Offending crates

```
cargo bloat --crates

File  .text     Size Crate
4.7%  45.6% 285.5KiB std
3.4%  33.5% 209.5KiB clap_builder
1.2%  11.8%  73.7KiB tokio
0.4%   3.8%  24.1KiB base (our crate)
```

Easy measure: limit `clap`'s features

```
cargo add clap --no-default-features -F help,derive,std
```

target	base size	base reduction	lean size	lean reduction
amd64 linux	768k	9.4%	336k	16.0%
i686 musl	840k	7.5%	428k	11.6%
arm64 android	708k	8.8%	360k	13.5%
windows64	680k	11.9%	368k	17.9%
arm64 mac	700k	8.4%	312k	13.3%

Replacing “clap” with “argh”

```
/// Port forwarder
#[derive(argh::FromArgs)]
struct Args {
    #[argh(positional)]
    listen_addr: SocketAddr,
    #[argh(positional)]
    connect_addr: SocketAddr,
}

let Args { listen_addr, connect_addr }
            = argh::from_env();
```

argh results

target	base size	base reduction	lean size	lean reduction
amd64 linux	556k	27.6%	192k	42.9%
i686 musl	648k	22.9%	296k	30.8%
arm64 android	528k	25.4%	228k	36.7%
windows64	480k	29.4%	216k	41.3%
arm64 mac	524k	25.1%	184k	41.0%

Limiting Tokio's features

```
cargo add tokio -F net,rt,macros,io-util
```

target	base size	base reduction	lean size	lean reduction
amd64 linux	428k	23.0%	124k	35.4%
i686 musl	520k	19.8%	220k	25.7%
arm64 android	404k	23.5%	156k	31.6%
windows64	400k	16.7%	192k	11.1%
arm64 mac	392k	25.2%	152k	17.4%

Removing `anyhow`

```
type Error = Box<dyn std::error::Error + Send + Sync>;
```

Errors still visible for CLI user, though not as nice.

target	base size	base reduction	lean size	lean reduction
amd64 linux	416k	2.8%	88k	29.0%
i686 musl	508k	2.3%	100k	54.5%
arm64 android	392k	3.0%	96k	38.5%
windows64	388k	3.0%	152k	20.8%
arm64 mac	392k	0.0%	120k	21.1%

What remains?

```
2.0%  31.1% 18.6KiB base (our crate)
1.7%  26.1% 15.6KiB tokio
1.3%  19.5% 11.7KiB core
1.0%  15.6%  9.3KiB std
0.2%   2.7%  1.6KiB alloc
0.2%   2.6%  1.6KiB [Unknown]
0.1%   1.2%    714B argh
```

What remains?

```
1.1%  16.1%  9.6KiB      base base::main::{{closure}}
1.0%  15.1%  9.0KiB      base base::main
0.3%   4.2%  2.5KiB     tokio tokio::...::poll
0.2%   2.6%  1.6KiB      core <Debug>::fmt
0.1%   2.2%  1.3KiB [Unknown] main
0.1%   1.8%  1.1KiB     tokio t.::r.::sch.::cu._th.::sh.2
0.1%   1.6%    983B     tokio copy_bidirectional::tr...on
0.1%   1.6%    978B      core core::fmt::Formatter::pad
```

What remains?

- Actual implementation
- Formatting

Apart from **UPX** (that would get us to 48k), there seems to be no easy ways to shrink this further.

Note that we are **93% down** from the original size.

Part 2B: Shrinking challenge (hacks)

Unportable trick: no_main

We have **lost Windows** portability.

There are still some error reports and CLI `--help` message.

There is ugly **startup function**, but CLI args and the algorithms niceties are **intact**. If stable is needed, the app can be turned into a staticlib and be used from a C wrapper.

```
#![no_main]
#[no_mangle]
fn main(mut argc: c_int, argv: *mut*mut c_char,
        _envp: *mut*mut c_char) -> c_int { ...
```

no_main results

target	base size	base reduction	lean size	lean reduction
amd64 linux	396k	4.8%	76k	13.6%
i686 musl	484k	4.7%	88k	12.0%
arm64 android	376k	4.1%	84k	12.5%
arm64 mac	376k	4.1%	100k	16.7%

Honourable mention: eyra

Fully Rust C library, including startup object files.

Worse UX: only basic output

Though we got rid of the formatted output, formatting code is **still present** in the executable.

target	base size	base reduction	lean size	lean reduction
amd64 linux	388k	2.0%	72k	5.3%
i686 musl	480k	0.8%	84k	4.5%
arm64 android	368k	2.1%	80k	4.8%
arm64 mac	360k	4.3%	100k	0.0%

Investigating why fmt

Start by **ensuring** empty program is really **empty**.

```
fn main(...) -> c_int {
    return 0;
    ...
}
```

```
File  .text Size     Crate Name
0.2%  14.3%  34B [Unknown] _start
0.0%   1.3%   3B [Unknown] main
```

Seems to be **close enough** to be considered **empty**.

Investigating why fmt

- As `return 0` is **moved lower** in `fn main`, we see functions gradually **appear** in `cargo bloat`.
- **Formatting code appears** on the `rt.block_on(run(x))` line where we jump to our function.
- So we begin with `return Ok(())` in **`fn run`** and **move it down** the similar way.
- Formatting code **appears** when we get to the `TcpListener::bind(listen_addr).await?` line.
- Specifically it's the **question mark** that causes it to be included.

Victory over fmt?

```
struct DummyError;
impl<T:std::error::Error> From<T> for DummyError {
    fn from(_value: T) -> Self {
        DummyError
    }
}

async fn run(args: Args) -> Result<(), DummyError>;
```

Victory over fmt?

target	base size	base reduction	lean size	lean reduction
amd64 linux	384k	1.0%	56k	22.2%
i686 musl	480k	0.0%	76k	9.5%
arm64 android	364k	1.1%	60k	25.0%
arm64 mac	360k	0.0%	84k	16.0%

Victory over fmt?

But the **spectre of `fmt`** is still hauting the project.

```
9.3KiB              main
4.9KiB     tcpfwd tcpfwd::run::{{closure}}
2.4KiB     tokio tokio::runtime::task::raw::poll
1.6KiB     alloc <... core::fmt::Debug>::fmt
1.1KiB     tokio tokio::r...::shutdown2
1.0KiB     alloc <... core::fmt::Debug>::fmt
  983B     tokio copy_bidirectional::transfer_one_direction
  846B      core core::net::parser::<SocketAddr>::from_str
  665B     tokio tokio::...::scheduled_io::ScheduledIo::wake
  653B     tokio tokio::runtime::io::driver::Driver::turn
```

Extinguishing the last remnant of fmt

The culprit is **`io::Error::new`** calls in Tokio.

Let's **patch** it away.

```
cargo override --path ~/src/tokio/tokio
```

Starting with `panic!()` at the top of `TcpListener::bind` and going down (i.e. **reordering the lines** for the `panic` to be lower and lower). Also just searching for `Error::new`.

Extinguishing the last remnant of fmt

```
-            None => Err(io::Error::new(
-                io::ErrorKind::InvalidInput,
-                "invalid address family (not IPv4 or IPv6)",
-            )),                         # (and 6 similar chunks)
+            None => Err(io::ErrorKind::InvalidInput.into()),
```

target	base size	base reduction	lean size	lean reduction
amd64 linux	384k	0.0%	48k	14.3%
i686 musl	472k	1.7%	68k	10.5%
arm64 android	364k	0.0%	56k	6.7%
arm64 mac	360k	0.0%	84k	0.0%

Manual implementation: no tokio

```
cargo rm tokio
cargo add mio -F net,os-poll
```

Avoiding allocations, using my own `Option` to force `.bss`.

About 200 LOC.

target	base size	base reduction	lean size	lean reduction
amd64 linux	296k	22.9%	12k	75.0%
i686 musl	396k	16.1%	28k	58.8%
arm64 android	276k	24.2%	12k	78.6%
arm64 mac	280k	22.2%	52k	38.1%

Outro

Helpful link

https://github.com/johnthagen/min-sized-rust

Shrinking methods review (easy)

Re-stating what it takes to shrink an executable.

- Compilation **mode**
- **Dependency** trimming
  - features
  - crate replacements
- Shrinking the **requirements**

Shrinking methods review (hard)

Re-stating what it takes to shrink an executable.

- Nightly-only **hacks**
- `no_std` / `no_main` / `eyra` / `staticlib` / etc
- Dependency trimming
  - **patching**
    - libstd patching
- Manual adjustments to **inlining**, etc.

What typical Rust executable contains (1)

- startup code like `crt1.o`
- C standard library (or symbols for linking to it)
- Compiler functions library like `libgcc`
- Rust's main function wrapper

…

What typical Rust executable contains (2)

…

- Rust standard library
  - Formatting code
  - Unwinder
  - Threading / TLS initializer
- Your project's dependencies
- Your own code

The end