bitvec
little bits of memory

A magical library by myrrlyn

@ myrrlyn

Who Am I

- Rust Enthusiast

@ myrrlyn

Who Am I

- Rust Enthusiast
- Satellite Software Engineer

@ myrrlyn

But enough about me

What even *is* `bitvec`

`@ myrrlyn`

Apples vs Oranges

C++

- has bit-field syntax
- has `<bitset>`
- packs `std::vector<bool>`

Rust

- does not have it
- does not have this either
- also does not do *that*

Why Did I Make `bitvec`

Jealousy

- Anything C++ can do, Rust should do also

Spite

- Anything C++ can do, Rust should do *better*

But What *Is* `bitvec`

Pointer to [u1]

- Describes any region of memory with bit precision
- ANY region: can start and end at any bit in a byte

…with more power!

- Users can specify element size (u8, u16, u32, u64) and bit ordering
- BYO Bit Ordering

(How) Did I Do That?

Hint: Scarily

@ myrrlyn

How It Doesn't Work

- Pointer, With Extras
- This is too big
- Can't become a reference
- Can't be used in any traits

```
struct BadPointer<T> {
  ptr: *const T,
  bit: u8,
}
```

Does not become

```
&T
```

How It Does Work

- Slice pointer
- First-class language item
- Points to the start of memory, and counts how many things are there
- **Can** become a reference
- **Can** be used in traits

```
struct SlicePtr<T> {
  ptr: *const T,
  len: usize,
}
```

does become

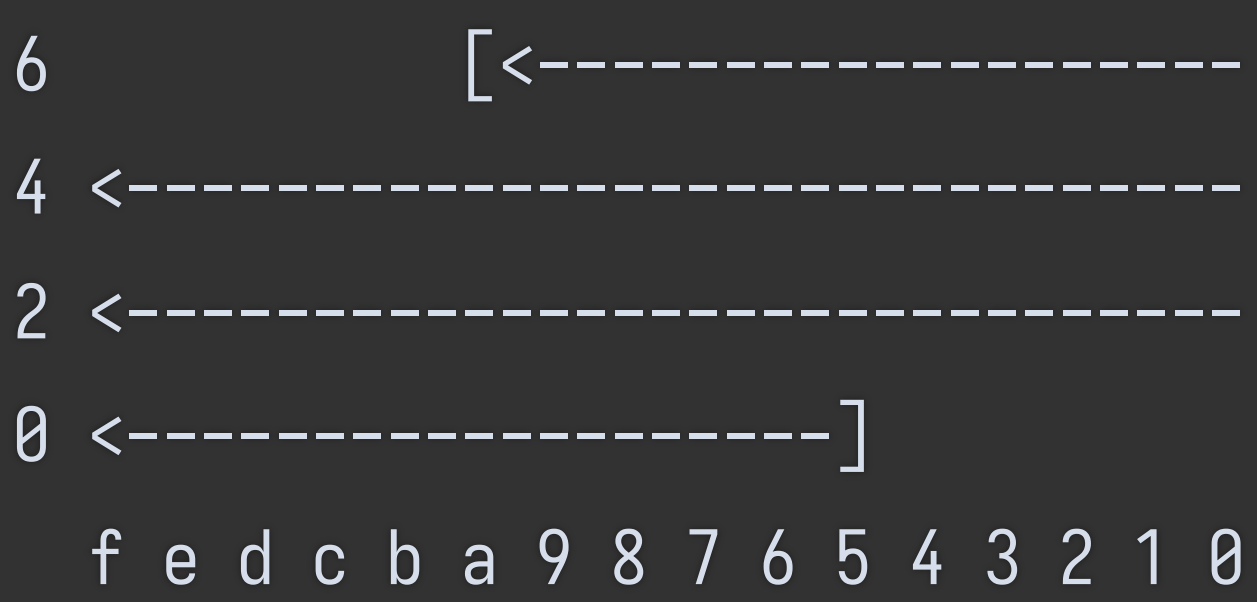
```
&[T]
```

Nifty Details

- Still a slice pointer
- Can go anywhere a slice pointer can
- **Can** become a reference
- **Can** be used in traits
- Cannot be used as a slice pointer!
- Can only index 12% of `usize`

```
template <typename T>
struct BitPtr {

  // byte select
  size_t ptr_head
  : ctzll(alignof(T));

  // data address
  size_t ptr_data
  : sizeof(uintptr_t) * 8
  - ctzll(alignof(T));

  // bit count
  size_t len_head : 3;

  // bit select
  size_t len_bits
  : sizeof(size_t) * 8 - 3;

};
```

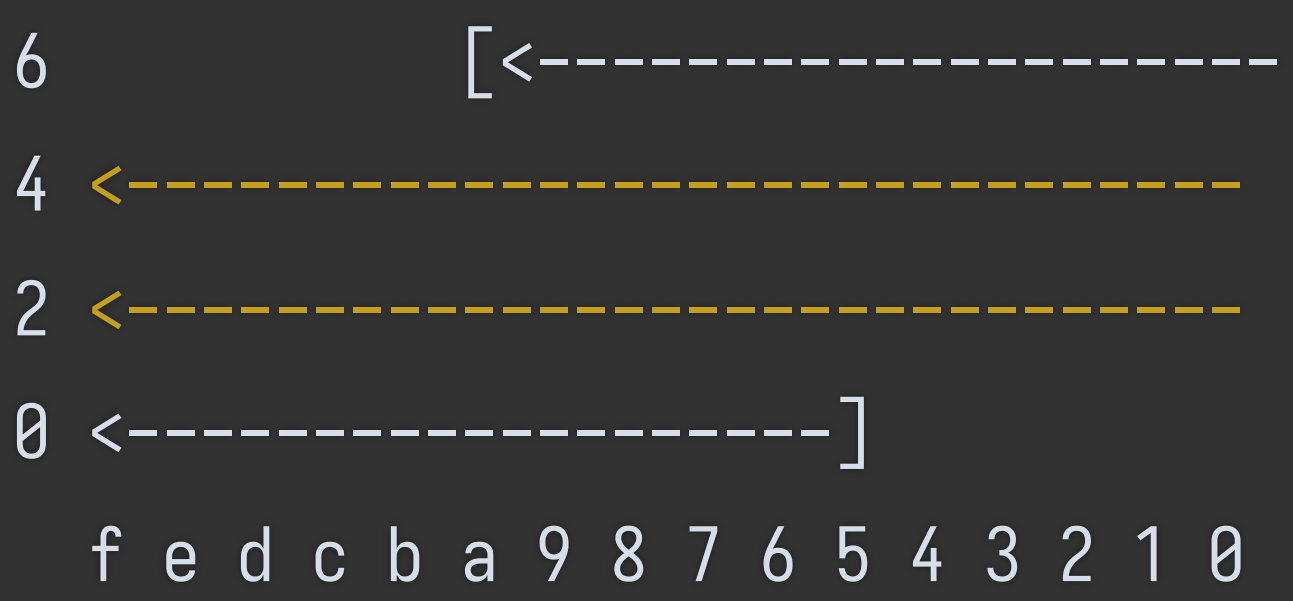
Show Me The Memory



Suppose we have this span of
`<LittleEndian, u16>`

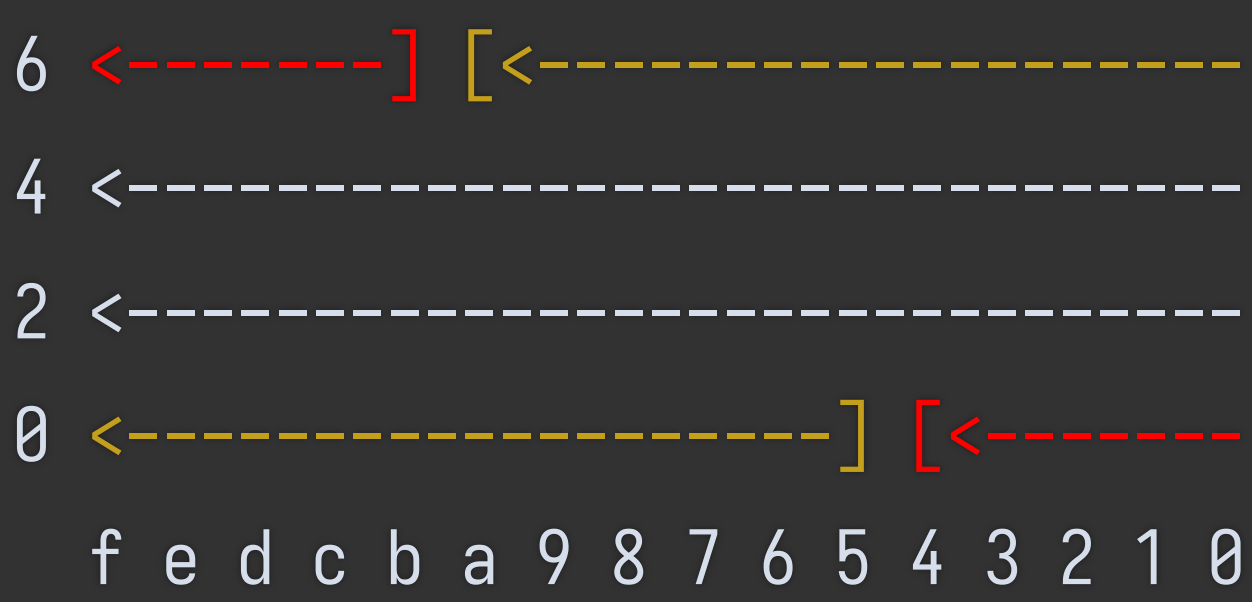
How do we work with it?

Show Me The Memory



The middle two elements are completely owned, no contest

Show Me The Memory



The outer elements are aliased!

Does Aliasing Actually Matter? (no.)

Multithreaded

- Uses atomics by default.
- Free on x86
- I don't test on ARM, but it's probably fine
- `Ordering::Relaxed` is good enough™

Single-Threaded

- `[dependencies.bitvec]`
  `default-features = false`
- uses `Cell<T>`
- Safe aliasing, no concurrency

Types Are Just Lies We Agree To Believe

- You don't need to make your region atomic before making it a `BitSlice`
- It's actually only atomic *while* it's a `BitSlice`
- You and I only care about machine instructions
- the compiler only cares about reference correctness

```
#[cfg(feature = "atomic")]
type Access = AtomicU8;

#[cfg(not(feature = "atomic"))]
type Access = Cell<u8>;

0x0000_7fff_1063_ab3e
as *const u8
as *const u8::Access
```

What Does The API Look Like

Rust Standard Library

- Raw Pointer `*const [bool]`
- Slice Reference `&[bool]`
  - with mutability! `&mut [bool]`
- Vector `Vec<bool>`
  - macros! `vec![false, true, ...]`
- Boxed Slice `Box<[bool]>`
  - refcounts! `Arc<[bool]>`

`bitvec` crate

- `BitPtr<T>` (ITAR Restricted)
- `&BitSlice<C, T>`
  - `&mut BitSlice<C, T>`
- `BitVec<C, T>`
  - `bitvec![C, T; 0, 1, ...]`
- `BitBox<C, T>`
  - `unimplemented!("PRs welcome")`

@ myrrlyn

How Do I Make A BitSlice

- Import the prelude
- Make some data
- Reinterpret the memory region

```
use bitvec::prelude::*;
let mut data = [0u8; 16];
let bits = data.bits_mut::<BigEndian>();
```

Wait, What's That C Type Parameter

- `bitvec` exports a `Cursor` trait
- It maps from abstract counting to concrete bit positions
- `bitvec` provides two implementors
  - `BigEndian`: start at high bit, work downwards
  - `LittleEndian`: start at low bit, work upwards
- You can provide your own
  - Follow the rules listed in the docs
  - Do not lie to me, because I trust you

Wait, *Why*'s That `C` type parameter?

- IP packets use little-endian bit ordering
- TCP packets use big-endian bit ordering
- Do you trust yourself to remember that?
-
```
type IpPkt = BitSlice<LittleEndian, u32>;
type TcpPkt = BitSlice<BigEndian, u32>;
```

Why Would I, The Audience, Use This

- Memory compaction:
  - `&[bool]` and `Vec<bool>` are now 12% of their original size.
  - Their handles did not become larger
- Roll your own `[Option<T>]`:
  - `BitSlice + [MaybeUninit<T>]`: it's smaller!
- I/O protocol buffers:
  - TCP specifies fields less than one byte wide.
  - You could shift and mask yourself
  - Or you could … not. `bitvec` 0.16 has bitfields built in.

How Do I, The Audience, Use This

- Depend on it
  - ```
    # Cargo.toml
    [dependencies]
    bitvec = "*"
    ```
- Do some text substitution
  - ```
    use bitvec::prelude::*;

    &[bool] → &BitSlice
    Vec<bool> → BitVec
    ```
- Fix any errors that arise (no `IndexMut` means no `[]=`)
- File an issue if you think they shouldn't happen

Thanks!

@ myrrlyn