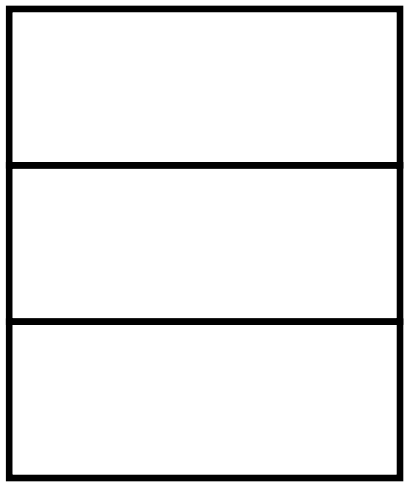
Pinning in Rust

Benno Lossin (y86-dev@proton.me)

September 7, 2022

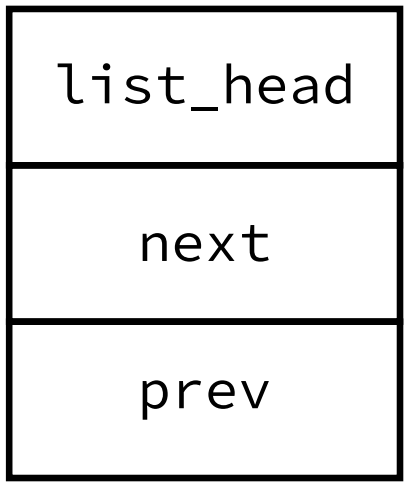
What is Pinning?

- "pinned" = "stable address"



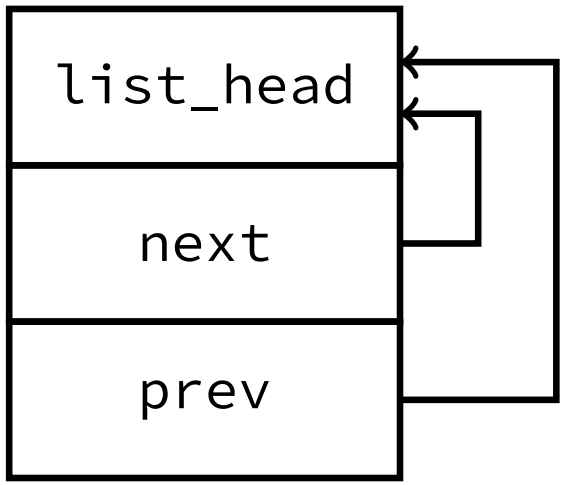
What is Pinning?

- "pinned" = "stable address"



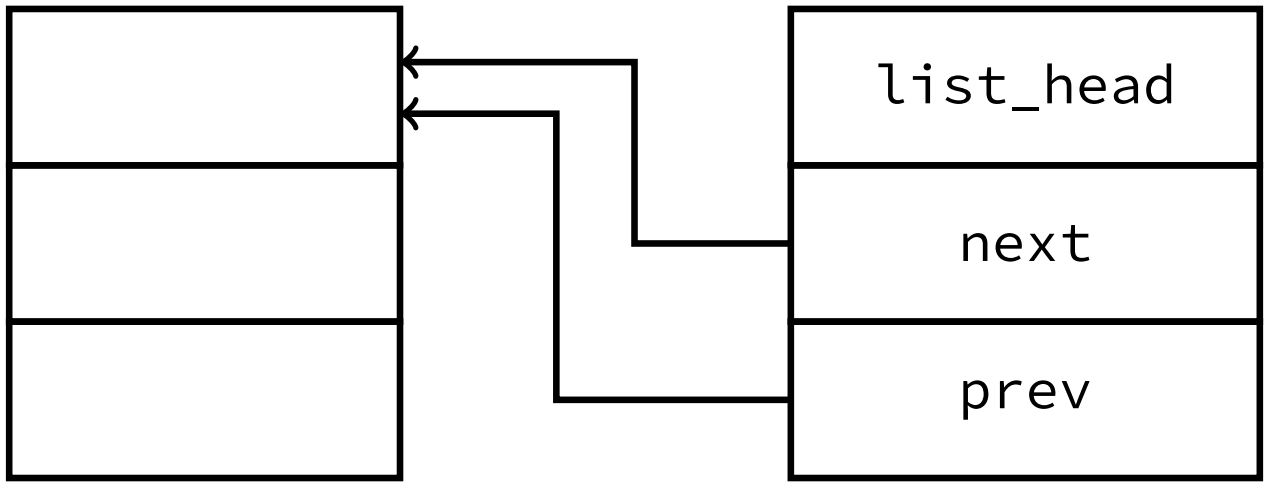
What is Pinning?

- "pinned" = "stable address"



What is Pinning?

- "pinned" = "stable address"
- moving data invalidates pointers



Why is this a Problem in Rust?

Why is this a Problem in Rust?

- all types in Rust are moveable

Why is this a Problem in Rust?

- all types in Rust are moveable
- but some types *need* to have a stable address

Why is this a Problem in Rust?

- all types in Rust are moveable
- but some types *need* to have a stable address
- Rust's solution: consider pointers pinned via `Pin<P>`, where `P: Deref<Target = T>`

Why is this a Problem in Rust?

- all types in Rust are moveable
- but some types *need* to have a stable address
- Rust's solution: consider pointers pinned via `Pin<P>`, where `P: Deref<Target = T>`
- `Pin<P>` prevents access to `&mut T`

Why is this a Problem in Rust?

- all types in Rust are moveable
- but some types *need* to have a stable address
- Rust's solution: consider pointers pinned via `Pin<P>`, where `P: Deref<Target = T>`
- `Pin<P>` prevents access to `&mut T`
  $\Longrightarrow$ e.g. no calls to `mem::swap`

Why is this a Problem in Rust?

- all types in Rust are moveable
- but some types *need* to have a stable address
- Rust's solution: consider pointers pinned via `Pin<P>`, where `P: Deref<Target = T>`
- `Pin<P>` prevents access to `&mut T`
  $\Longrightarrow$ e.g. no calls to `mem::swap`
- however certain types (e.g. usize/u8...u64 etc.) do not care about being pinned, since they implement `Unpin`

Why is this a Problem in Rust?

- all types in Rust are moveable
- but some types *need* to have a stable address
- Rust's solution: consider pointers pinned via `Pin<P>`, where `P: Deref<Target = T>`
- `Pin<P>` prevents access to `&mut T`
  $\Longrightarrow$ e.g. no calls to `mem::swap`
- however certain types (e.g. usize/u8...u64 etc.) do not care about being pinned, since they implement `Unpin`
- When `T: Unpin` then `Pin<P>` behaves like `P`
  e.g. `Pin<Box<u64>>` behaves like `Box<u64>`

An Example

```
1  pub struct SelfReferential {
2      value: u32,
```

An Example

```
1  pub struct SelfReferential {
2      value: u32,
3      ptr: *const u32,
```

An Example

```
1  pub struct SelfReferential {
2      value: u32,
3      ptr: *const u32,
4      _pin: PhantomPinned,
5  }
```

An Example

```rust
pub struct SelfReferential {
    value: u32,
    ptr: *const u32,
    _pin: PhantomPinned,
}

impl SelfReferential {
    /// SAFETY: Callers need to pin the returned value
    /// and then initialize ptr.
    pub unsafe fn new(value: u32) -> Self {
        Self {
            value,
            ptr: core::ptr::null(),
            _pin: PhantomPinned,
        }
    }
}
```

APIs with `Pin<P>`

- explicit API support required (as `&mut` cannot be obtained)

APIs with `Pin<P>`

- explicit API support required (as `&mut` cannot be obtained)
- `fn foo(self: Pin<&mut Self>)` instead of
  `fn foo(&mut self)`

APIs with `Pin<P>`

- explicit API support required (as `&mut` cannot be obtained)
- `fn foo(self: Pin<&mut Self>)` instead of `fn foo(&mut self)`
- a pinned value can *never* be unpinned

APIs with `Pin<P>`

- explicit API support required (as `&mut` cannot be obtained)
- `fn foo(self: Pin<&mut Self>)` instead of `fn foo(&mut self)`
- a pinned value can *never* be unpinned
- `Pin<&mut Self>` creates a new problem: how do we access the fields?

Example Continued

```
1  pub struct SelfReferential {
2      value: u32,
3      ptr: *const u32,
4      _pin: PhantomPinned,
5  }
6
7  impl SelfReferential {
8      pub fn init(self: Pin<&mut Self>) {
9
10
11
12
13     }
14 }
```

Example Continued

```
1  pub struct SelfReferential {
2      value: u32,
3      ptr: *const u32,
4      _pin: PhantomPinned,
5  }
6
7  impl SelfReferential {
8      pub fn init(self: Pin<&mut Self>) {
9          self.ptr = &self.value;
10
11
12
13     }
14 }
```

Example Continued

```rust
pub struct SelfReferential {
    value: u32,
    ptr: *const u32,
    _pin: PhantomPinned,
}

impl SelfReferential {
    pub fn init(self: Pin<&mut Self>) {
        self.ptr = &self.value;
     // ^^^^^^^^^^^^^^^^^^^^^^^
     // cannot assign to data in dereference of
     // `Pin<&mut SelfReferential>`
    }
}
```

Example Continued

```rust
pub struct SelfReferential {
    value: u32,
    ptr: *const u32,
    _pin: PhantomPinned,
}

impl SelfReferential {
    pub fn init(self: Pin<&mut Self>) {
        // SAFETY: we do not move out of this
        let this: &mut Self = unsafe {
            Pin::get_unchecked_mut(self)
        };
        this.ptr = &this.value;
    }
}
```

Pin Projection

- accessing fields via projections

Pin Projection

- accessing fields via projections
- **either** `Pin<&mut Field>` (structural pinning) **or** `&mut Field` (not structural)

Pin Projection

- accessing fields via projections
- **either** `Pin<&mut Field>` (structural pinning) **or** `&mut Field` (not structural)
- **`unsafe`** necessary to ensure this invariant

Pin Projection

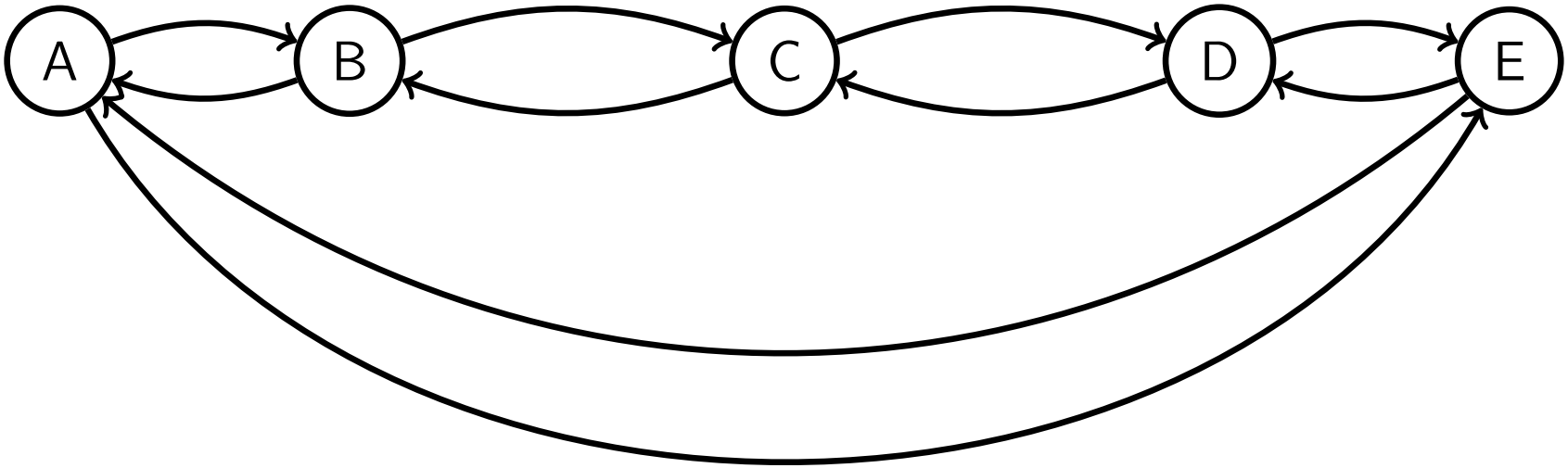
- accessing fields via projections
- **either** `Pin<&mut Field>` (structural pinning) **or** `&mut Field` (not structural)
- **`unsafe`** necessary to ensure this invariant
- `pin-project` is a macro crate creating these projections safely

Drop Guarantee

- we still need one more guarantee

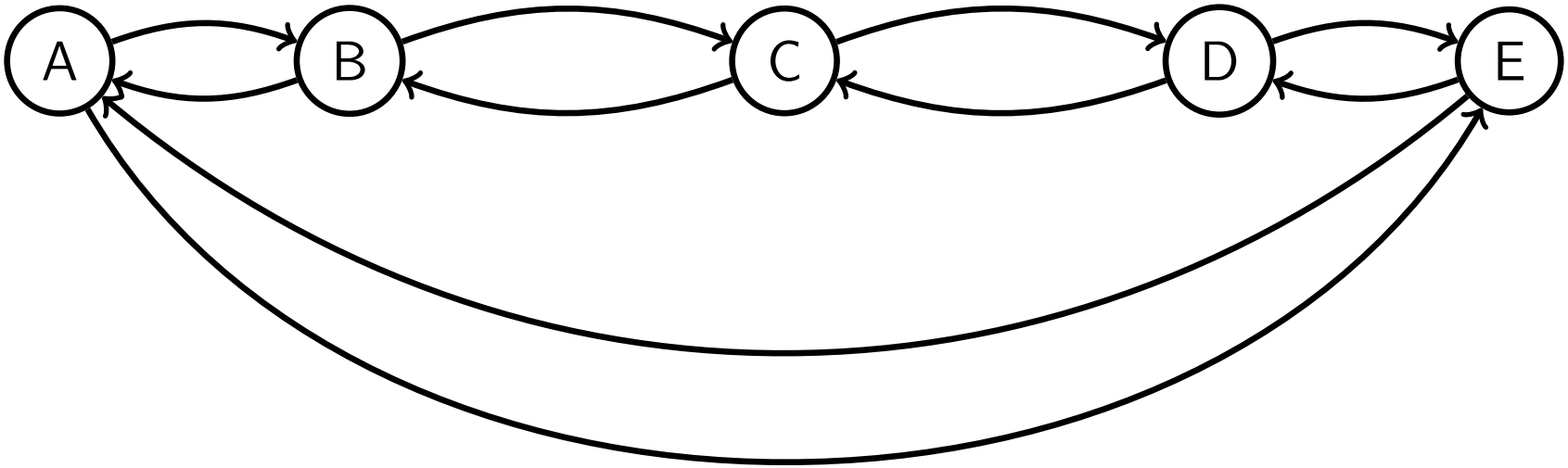
Drop Guarantee

- we still need one more guarantee



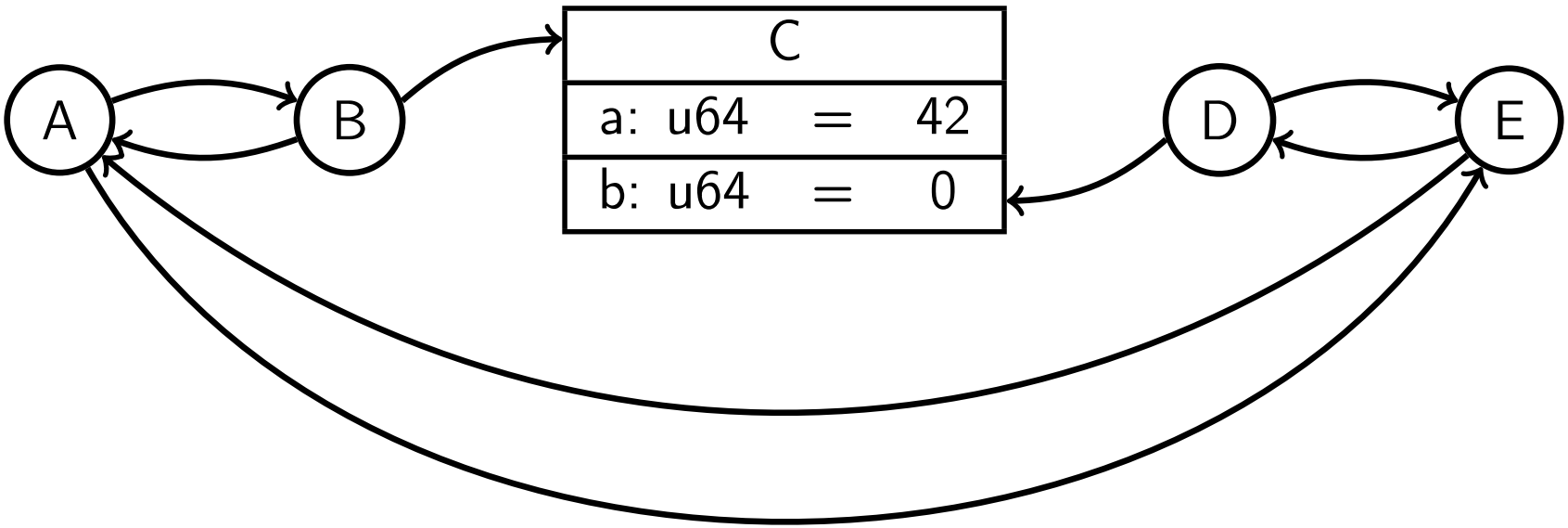
Drop Guarantee

- what if we just repurpose `C`?



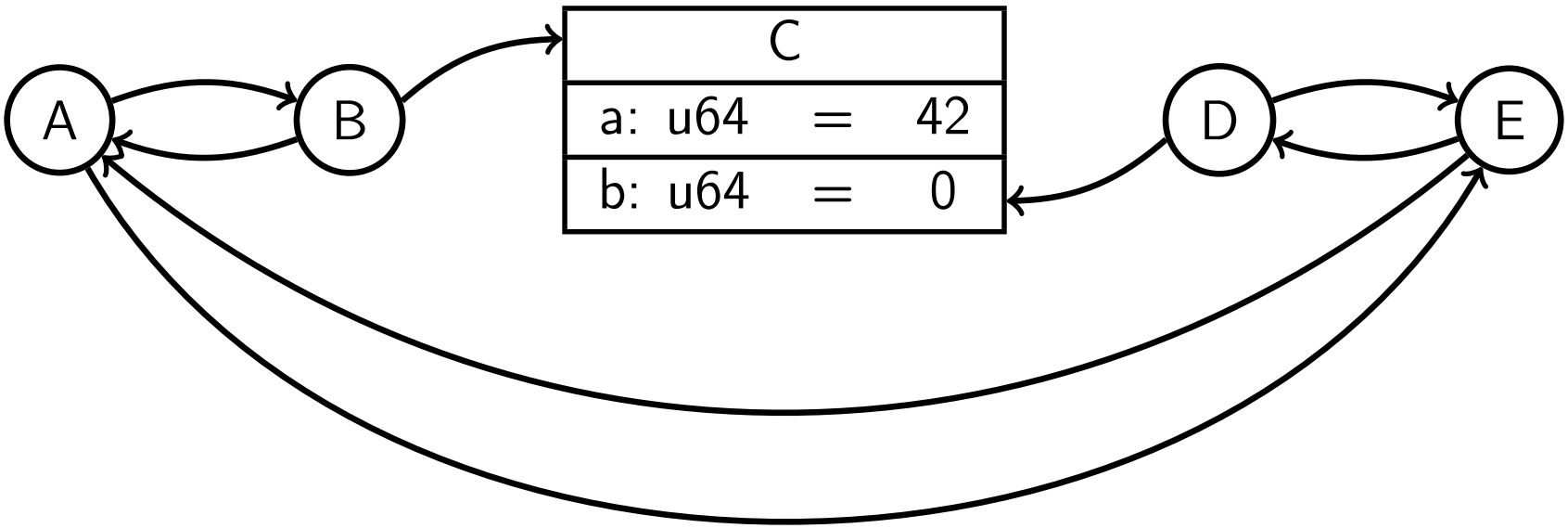
Drop Guarantee

- what if we just repurpose `C`?



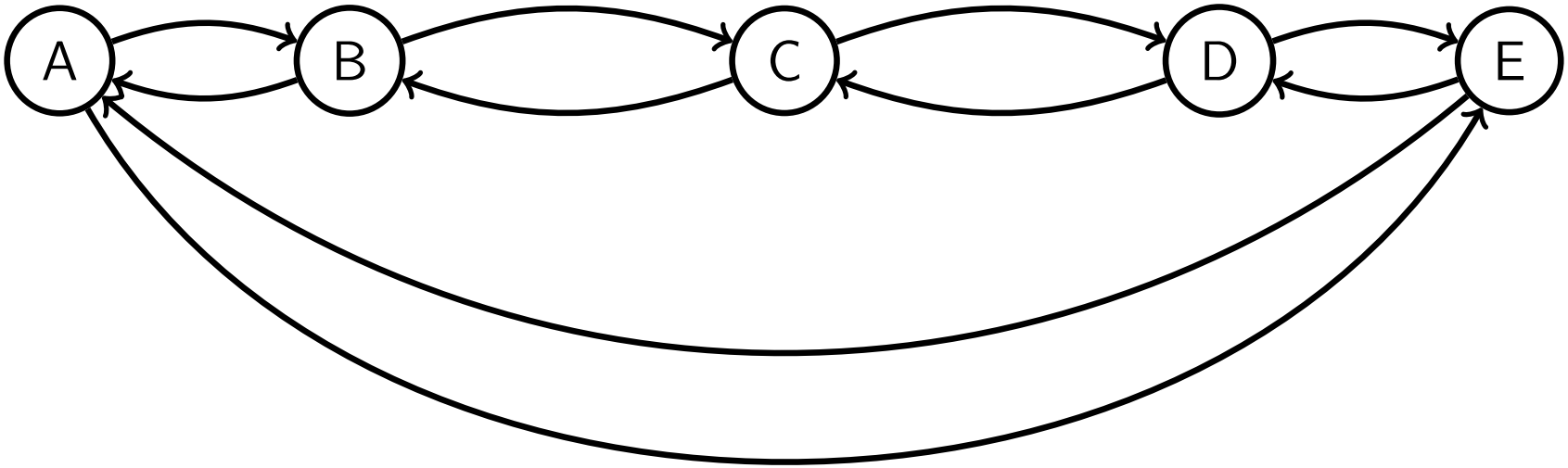
Drop Guarantee

- oh no, our linked list contains garbage!



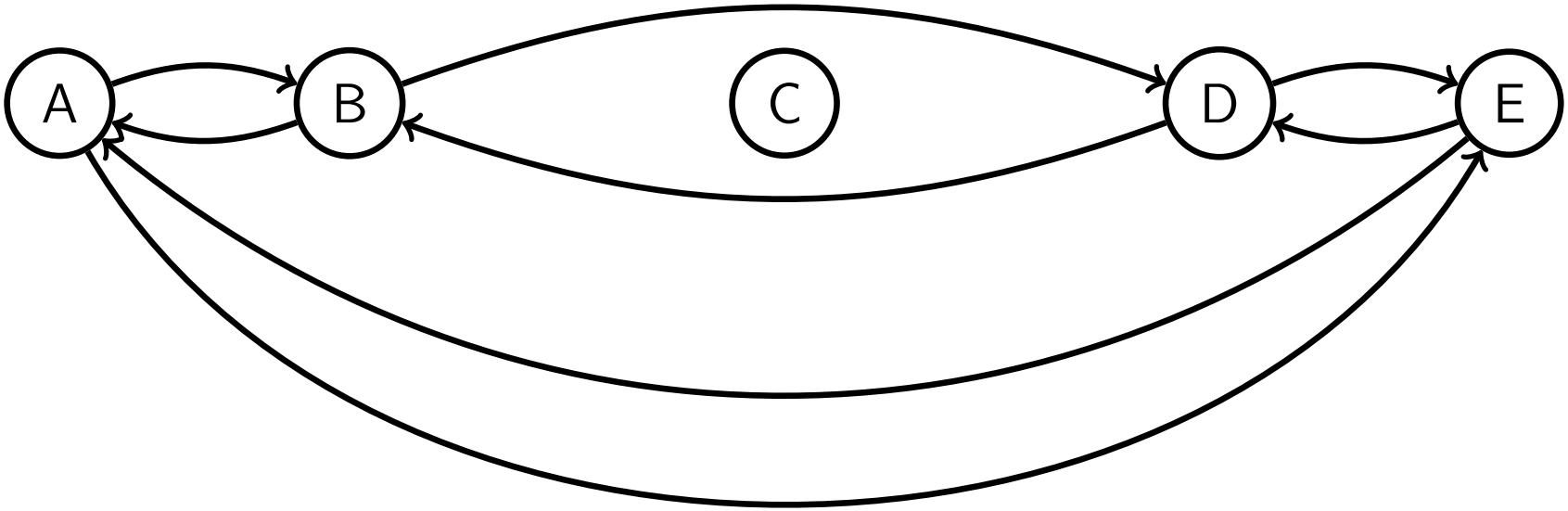
Drop Guarantee

- ensures that no memory is repurposed before `drop()` is called



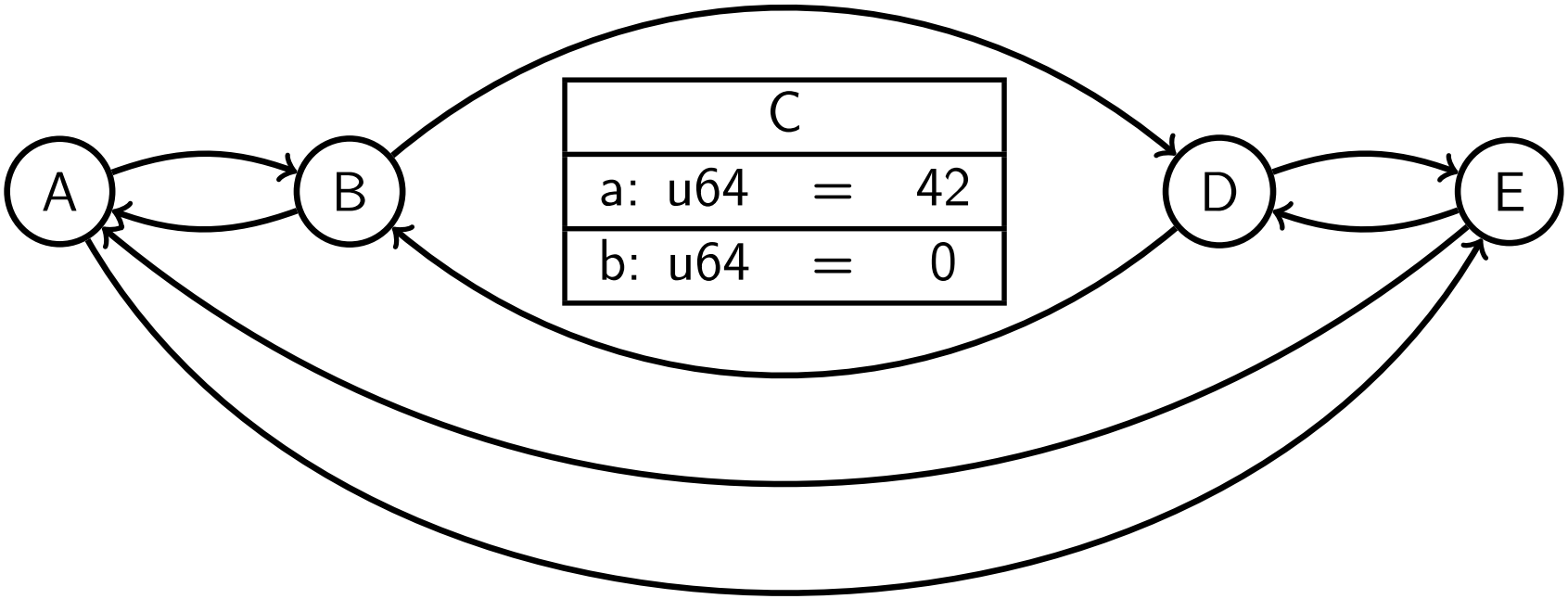
Drop Guarantee

- calling `drop()` on C unlinks it from the list



Drop Guarantee

- now we can repurpose C



Drop Guarantee

- ensure that no memory is repurposed before `drop()` is called
- repurposing is overwriting using `ptr::write` without `drop`ping before

Drop Guarantee

- ensure that no memory is repurposed before `drop()` is called
- repurposing is overwriting using `ptr::write` without `drop`ping before
- repurposing is deallocation

Additional Pitfalls

- `Drop` implementation: implicitly takes `Pin<&mut Self>` instead of `&mut self`

Additional Pitfalls

- `Drop` implementation: implicitly takes `Pin<&mut Self>` instead of `&mut self`
- even `drop(&mut self)` is not allowed to move out of structurally pinned fields

Additional Pitfalls

- `Drop` implementation: implicitly takes `Pin<&mut Self>` instead of `&mut self`
- even `drop(&mut self)` is not allowed to move out of structurally pinned fields
- this is why packed structs cannot be pinned!

Additional Pitfalls

- `Drop` implementation: implicitly takes `Pin<&mut Self>` instead of `&mut self`
- even `drop(&mut self)` is not allowed to move out of structurally pinned fields
- this is why packed structs cannot be pinned!
- only implement `Unpin` if all structurally pinned fields are `Unpin`

Conclusion

- `Pin<P>` pins the pointee indefinitely if `!Unpin`

Conclusion

- `Pin<P>` pins the pointee indefinitely if `!Unpin`
- `Pin<P>` requires API support

Conclusion

- `Pin<P>` pins the pointee indefinitely if `!Unpin`
- `Pin<P>` requires API support
- **`unsafe`** needed for pin-projections

Conclusion

- `Pin<P>` pins the pointee indefinitely if `!Unpin`
- `Pin<P>` requires API support
- **`unsafe`** needed for pin-projections
- `Drop` and `Pin<P>` interactions: `Drop` guarantee, pinned even in `drop`