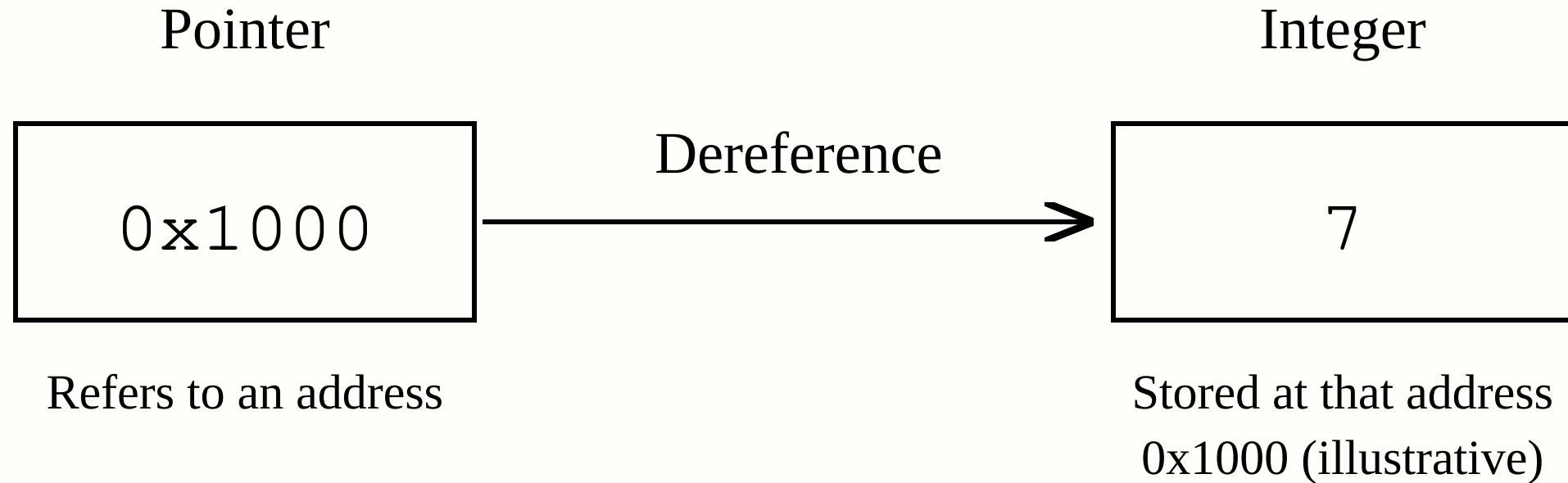


Ownership

Lukáš Hozda

NPRG082 · winter 2026/27

Pointers



- Sometimes we need to share a piece of memory
- Pointers let us observe remote state, and possibly mutate it (woo, dangerous!)

Stack and heap

- The **stack** holds temporary storage for active function calls
- Each call adds a **frame** for its local data
- Returning from a call releases its frame
- The **heap** holds separate blocks allocated while the program runs
- A heap block lasts until released, possibly after the allocating call returns

Values and bindings

```
fn main() {  
    let text = String::from("oak");  
    println!("{}", text.len());  
}
```

- A local text binding containing a String
- The String owns a buffer of UTF-8 bytes

Owned storage

text: String

pointer	
length	3
capacity	≥ 3

Owned heap buffer

o	a	k
---	---	---

UTF-8 bytes

Storage and ownership

- Local values usually occupy space in the stack frame
- `String` and `Vec<T>` own heap buffers
- Ownership rules apply to heap-free values too
- The compiler decides storage placement and machine copies

Structs

```
struct Point { x: i32, y: i32 }  
fn main() {  
    let mut point = Point { x: 3, y: 4 };  
    point.x += 1;  
    println!("{}", point.x, point.y);  
}
```

- Local struct values normally live on the stack
- Field order in memory is unspecified by default

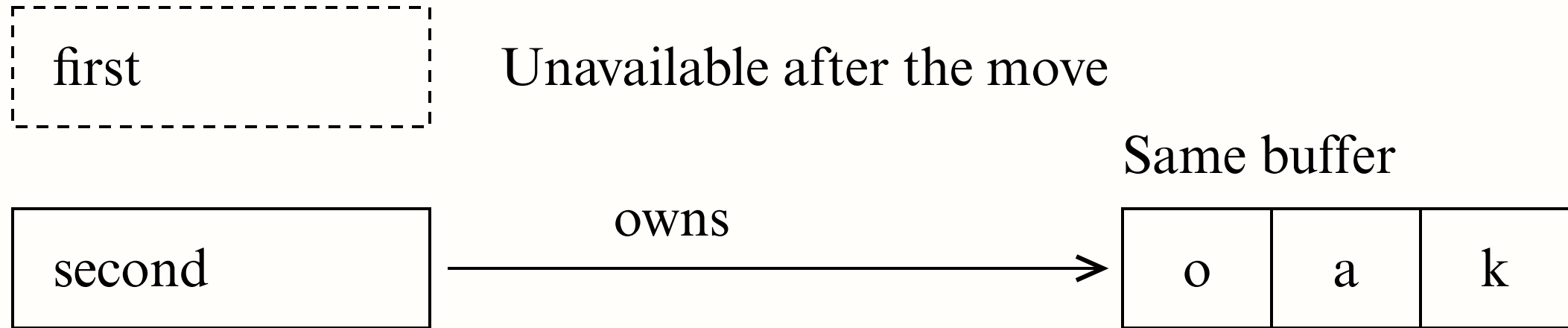
Ownership transfer

```
fn main() {  
    let first = String::from("oak");  
    let second = first;  
    println!("{second}");  
}
```

- Assignment moves a non-Copy value
- second now owns the string

String move

After `let second = first;`



- Ownership transfer without duplicating the buffer
- The old binding has no value available for use or drop

Boxed values

```
fn main() {  
    let first = Box::new(7);  
    let mut second = first;  
    *second += 1;  
    println!("{second}");  
}
```

- `Box<T>` owns a heap-allocated `T`
- Moving the box transfers ownership of the allocation
- Dropping the box drops `T` and releases the allocation

Moves without allocation

```
struct Token;  
fn main() {  
    let first = Token;  
    let _second = first;  
    let _third = first;  
}
```

- Token is a unit struct with no fields
- Even a zero-sized type is non-Copy by default

Reinitialization

```
fn main() {  
    let mut first = String::from("oak");  
    let second = first;  
    first = String::from("elm");  
    println!("{first} {second}");  
}
```

- A moved binding can receive a new value

Owned parameters

```
fn byte_len(text: String) -> usize {  
    text.len()  
}  
  
fn main() {  
    let name = String::from("oak");  
    println!("{}", byte_len(name));  
}
```

- The argument moves into the parameter, which is dropped on return

Return values

```
fn name() -> String {  
    let text = String::from("oak");  
    text  
}  
fn main() {  
    let result = name();  
    println!("{result}");  
}
```

- Returning the value transfers ownership to the caller

Implicit copies

```
fn main() {  
    let first = 7;  
    let mut second = first;  
    second += 1;  
    println!("{first} {second}");  
}
```

- i32 implements Copy, so both bindings have usable values

Copy as a type property

- Implicit, bitwise duplication in value contexts
- Integers, floats, bool, char, tuples and arrays of Copy elements
- Explicit implementation or derivation for user-defined types
- Type size and stack placement do not decide whether a type is Copy

Derived Copy

```
#[derive(Clone, Copy)]
struct Pair(i32, i32);
fn main() {
    let first = Pair(3, 4);
    let second = first;
    println!("{}", first.0, second.1);
}
```

- Pair is a tuple struct with fields .0 and .1
- Every field must be Copy, and the type cannot implement Drop

Explicit duplication

```
fn main() {  
    let first = String::from("oak");  
    let mut second = first.clone();  
    second.push('s');  
    println!("{first} {second}");  
}
```

- `String::clone` creates a separate owned buffer with the same contents

Move, copy, clone

- A **move** transfers ownership and makes the source value unavailable
- A **copy** duplicates the value implicitly
- A **clone** duplicates it through a type-defined operation
- Cloning may allocate or share a resource through a shared pointer

Partial moves

```
fn main() {  
    let pair = (String::from("L"), String::from("R"));  
    let left = pair.0;  
    println!("{left} {}", pair.1);  
}
```

- Moving one field leaves the other field available

Whole value after a partial move

```
fn main() {  
    let pair = (String::from("L"), String::from("R"));  
    let _left = pair.0;  
    println!("{pair:?}");  
}
```

- E0382 reports that the whole tuple is no longer available

Conditional moves

```
fn main() {  
    let text = String::from("oak");  
    if text.len() > 2 {  
        let _taken = text;  
    }  
    println!("{text}");  
}
```

- A value must be available on every path reaching its use

Destruction

- Initialized local values are dropped at scope exit
- Local bindings are dropped in reverse declaration order
- Owned fields are dropped recursively
- Implement Drop for additional type-specific cleanup

Observable drop order

```
struct Trace(u8);  
impl Drop for Trace {  
    fn drop(&mut self) { print!("{}", self.0); }  
}  
fn main() {  
    let _outer = Trace(1);  
    { let _inner = Trace(2); }  
    print!("end ");  
}
```

- Inner scope first, then the remaining outer local

Early release

```
fn main() {  
    let text = String::from("oak");  
    drop(text);  
    println!("{text}");  
}
```

- `std::mem::drop` takes its argument by value
- E0382 follows from transferring ownership into `drop`

Replacement

```
struct Trace(u8);  
impl Drop for Trace {  
    fn drop(&mut self) { print!("{}", self.0); }  
}  
fn main() {  
    let mut value = Trace(1);  
    value = Trace(2);  
    print!("body ");  
    drop(value);  
}
```

- Assignment drops the old value before storing the new one

Borrowed access

```
fn main() {  
    let value = 7;  
    let view = &value;  
    println!("{}", *view);  
}
```

Shared references

```
fn main() {  
    let text = String::from("oak");  
    let first = &text;  
    let second = first;  
    println!("{first} {second}");  
}
```

- `&T` is `Copy`, even when `T` is not
- Copying a reference does not duplicate the owned value

Mutable references

```
fn main() {  
    let mut value = 7;  
    let view = &mut value;  
    *view += 1;  
    println!("{value}");  
}
```

- `&mut T` provides exclusive access and is not Copy
- The reference binding itself need not be `mut`

Methods

```
struct Counter { value: u32 }  
impl Counter {  
    fn value(&self) -> u32 { self.value }  
}  
fn main() {  
    let counter = Counter { value: 7 };  
    println!("{}", counter.value());  
}
```

- Define methods in an `impl` block
- `&self` borrows the receiver, automatically in `counter.value()`

Mutating methods

```
struct Counter { value: u32 }  
impl Counter {  
    fn increment(&mut self) { self.value += 1; }  
}  
fn main() {  
    let mut counter = Counter { value: 7 };  
    counter.increment();  
    println!("{}", counter.value);  
}
```

- `&mut self` requires exclusive access to the receiver

Consuming methods

```
struct Label { text: String }  
impl Label {  
    fn into_text(self) -> String { self.text }  
}  
fn main() {  
    let label = Label { text: String::from("oak") };  
    let text = label.into_text();  
    println!("{text}");  
}
```

- `self` takes the receiver by value, moving this non-Copy label

Conflicting access

```
fn main() {  
    let mut text = String::from("oak");  
    let view = &text;  
    text.push('s');  
    println!("{view}");  
}
```

- E0502 reports shared access overlapping the mutable access in push

Last use of a borrow

```
fn main() {  
    let mut text = String::from("oak");  
    let view = &text;  
    print!("{view} ");  
    text.push('s');  
    println!("{text}");  
}
```

- The borrow can end after its last use, before the binding's scope ends

Exclusive access

```
fn main() {  
    let mut value = 7;  
    let first = &mut value;  
    let second = &mut value;  
    println!("{first} {second}");  
}
```

- E0499 reports overlapping mutable borrows of the same value

Reborrowing

```
fn main() {  
    let mut value = 7;  
    let outer = &mut value;  
    let inner = &mut *outer;  
    *inner += 1;  
    *outer += 1;  
    println!("{value}");  
}
```

- Access through outer resumes after the last use of inner

Overlapping reborrows

```
fn main() {  
    let mut value = 7;  
    let outer = &mut value;  
    let inner = &mut *outer;  
    *outer += 1;  
    println!("{inner}");  
}
```

- The later use of `inner` keeps its exclusive borrow active

Reference lifetime

```
fn main() {  
    let view;  
    {  
        let text = String::from("oak");  
        view = &text;  
    }  
    println!("{view}");  
}
```

- E0597 requires the referent to exist for every use of the reference

Sequence slices

```
fn main() {  
    let values = vec![10, 20, 30];  
    let tail = &values[1..];  
    println!("{tail:?}");  
}
```

- `&[T]` borrows contiguous elements and records their length
- One owned buffer, accessed through a smaller view

Borrowed parameters

```
fn byte_len(text: &str) -> usize { text.len() }  
fn main() {  
    let owned = String::from("oak");  
    let length = byte_len(&owned);  
    println!("{length} {}", byte_len("elm"));  
}
```

- A `&str` parameter accepts a borrowed `String` or a string literal

Borrowed results

```
fn trimmed(text: &str) -> &str { text.trim() }  
fn main() {  
    let owned = String::from(" oak ");  
    let view = trimmed(&owned);  
    println!("{view}");  
}
```

- The result borrows the input without allocating a new string
- Lifetime elision relates the result to the single input borrow

Lifetime parameters

```
fn trimmed<'a>(text: &'a str) -> &'a str {  
    text.trim()  
}  
  
fn main() {  
    let owned = String::from(" oak ");  
    println!("{}", trimmed(&owned));  
}
```

- 'a names the borrow lifetime shared by input and result
- An annotation relates borrows without extending the owner's lifetime

Lifetime elision

- Omitted lifetime parameters follow rules in the signature
- Each elided input lifetime gets a distinct parameter
- With exactly one input lifetime, elided outputs use that lifetime
- The function body is checked against the resulting signature

Ambiguous output lifetime

```
fn choose(left: &str, right: &str) -> &str {  
    if left.len() >= right.len() { left } else { right }  
}  
fn main() {  
    println!("{}", choose("oak", "elm"));  
}
```

- E0106 requires a lifetime for the result
- Neither input lifetime is selected by elision

Explicit output lifetime

```
fn choose<'a>(left: &'a str, right: &'a str) -> &'a str {  
    if left.len() >= right.len() { left } else { right }  
}  
  
fn main() {  
    let left = String::from("oak");  
    println!("{}", choose(&left, "elm"));  
}
```

- Both inputs must be valid for the chosen 'a
- The owners can have different scopes

Borrowing methods

```
struct Text { value: String }  
impl Text {  
    fn before(&self, separator: &str) -> &str {  
        self.value.split(separator).next().unwrap()  
    }  
}  
  
fn main() {  
    let text = Text { value: String::from("oak,elm") };  
    println!("{}", text.before(","));  
}
```

- The returned slice borrows self, not separator

Receiver lifetime elision

```
fn before(&self, separator: &str) -> &str

fn before<'a, 'b>(
    &'a self, separator: &'b str,
) -> &'a str
```

- Equivalent method signatures inside `impl Text`
- With `&self` or `&mut self`, elided outputs use the receiver's lifetime
- Other borrowed arguments can have independent lifetimes

Ownership in signatures

```
fn inspect(text: &str) { /* shared access */ }  
fn change(text: &mut String) { /* exclusive access */ }  
fn consume(text: String) { /* owned input */ }  
fn create() -> String { String::new() }
```

Ownership choices

- Borrow for temporary access
- Transfer ownership when the recipient needs the value
- Clone when a second owned value is needed
- Return an owned value for newly created data