# 301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

***AP-23***: *RUST*

# The RUST programming language

- Brief history
- Overview of main concepts
- Avoiding Aliases + Mutable
- Ownership and borrowing
- Traits, generics and inheritance
- (Slides by Haozhong Zhang)

# Brief History

- Development started in 2006 by Graydon Hoare at Mozilla.
- Mozilla sponsored RUST since 2009, and announced it in 2010.
- In 2010 shift from the initial compiler in **OCaml** to a self-hosting compiler written in **Rust**, **`rustc`**: it successfully compiled itself in 2011.
- **`rustc`** uses **LLVM** as its back end.
- Most loved programming language in the Stack Overflow annual survey of 2016, 2017, 2018, 2019, 2020 and 2021.
- February 8, 2021: The development of the language passes to the Rust Foundation (non-profit independent) funded by da Mozilla, Microsoft, Google, AWS e Huawei.

# On RUST syntax

- **Rust** is a system programming language with a focus on safety, especially safe concurrency, supporting both functional and imperative paradigms.
- Concrete syntax similar to C and C++ (blocks, `if-else`, `while`, `for`), `match` for pattern matching
- *Despite the superficial resemblance to C and C++, the syntax of Rust in a deeper sense is closer to that of the ML family of languages as well as the Haskell language.*
- Nearly every part of a function body is an expression (including `if-else`).

# Memory safety

- Designed to be memory safe:
  - No null pointers
  - No dangling pointers
  - No data races
- Data values can only be initialized through a fixed set of forms, requiring their inputs to be already initialized. Compile time error if any branch of code fails to assign a value to the variable.
- To avoid the use on "null", Rust core library provides an **option type**, which can be used to test if a pointer has Some value or None.
- Rust also introduces syntax to manage **lifetimes**, and the compiler reasons about these through its ***borrow checker***.

# Memory management

- No garbage collection. Deterministic management of resources, with very low overhead.
- Memory and other resources managed through Resource Acquisition Is Initialization (**RAII**), with optional reference counting. [Resource allocation is done during object initialization, by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor.]
- Rust favors stack allocation (default). No implicit boxing.
- Safety in the use of pointers/references/aliases is guaranteed by the **Ownership System** and by the compilation phase of **borrowing checking**.

# Ownership System

- Rust has an ownership system, based on concepts of ownership, borrowing and lifetimes
- Data are immutable by default, and declared mutable using **`mut`**.
- All values have a unique owner where the scope of the value is the same as the scope of the owner.
- A resource can be borrowed from its owner (via assignment or parameter passing) according to some rules.
- Values can be passed by immutable reference using **`&T`**, by mutable reference using **`&mut T`** or by value using **`T`**.
- At all times, there can either be multiple immutable references or one mutable reference to a resource. This is checked statically.

# Types and polymorphism

- **Type inference**, for variables declared with the **let** keyword.
- Classes are defined using **structs** for fields and implementations (**impl**) for methods.
- **No inheritance** in RUST! ➔ Pushing **composition over inheritance**
- The type system supports **traits**, corresponding to Haskell type classes, for ad hoc polymorphism.
- Traits can contain abstract methods or also concrete (default) methods. They cannot declare fields.
- Support for **bounded universal explicit polymorphism** with **generics**, as in Java, where bounds are one or more traits.

# Generic functions

- Generic functions may have the generic type of parameter bound by one or more traits. Within such a function, the generic value can only be used through those traits.
- Therefore a generic function can be type-checked when defined (as in Java, unlike C++ templates).
- However, *implementation* of Rust generics similar to typical implementation of C++ templates: a separate copy of the code is generated for each instantiation.
- This is called **monomorphization** and contrasts with the type erasure scheme of Java.
  - Pros: optimized code for each specific use case
  - Cons: increased compile time and size of the resulting binaries.

# An Introduction to Rust Programming Language

Haozhong Zhang

Jun 1, 2015

***Slides freely adapted by the lecturer***

# As a programming language …

```
fn main() {
    println!(“Hello, world!”);
}
```

- **Rust** is a *system programming language* barely on the *hardware*.
    - No *runtime* requirement (*eg*. GC/Dynamic Type/…)
    - More *control* (*over* memory allocation/destruction/…)
    - …

# More than that …

C/C++

Haskell/Python

more control,
less safety

less control,
more safety

**Rust**

*more control,*
*more safety*

# Rust overview

Performance, as with C

- Rust compilation to object code for bare-metal performance

But, supports memory safety

- Programs dereference only previously allocated pointers that have not been freed
- Out-of-bound array accesses not allowed

With low overhead

- Compiler checks to make sure rules for memory safety are followed
- Zero-cost abstraction in managing memory (i.e. no garbage collection)

Via

- Advanced type system
- **Ownership**, **borrowing**, and **lifetime** concepts to prevent memory corruption issues

But at a cost

- Cognitive cost to programmers who must think more about rules for using memory and references as they program

# Rust and typing

## Primitive types

- `bool`
- `char` (4-byte unicode)
- `i8/i16/i32/i64/isize`
- `u8/u16/u32/u64/usize`
- `f32/f64`

## Separate bool type

- C overloads an integer to get booleans
- Leads to varying interpretations in API calls
  - True, False, or Fail? 1, 0, -1?
  - Misinterpretations lead to security issues
  - Example: PHP **`strcmp`** returns 0 for both equality *and* failure!

## Numeric types specified with width

- Prevents bugs due to unexpected promotion/coercion/rounding

# Immutability by default

By default, Rust variables are immutable

- Usage checked by the compiler

mut is used to declare a resource as mutable.

```rust
1 fn main() {
2     let a: i32 = 0;
3     a = a + 1;
4     println!("{}" , a);
5 }
```

```
rustc 1.14.0 (e8a012324 2016-12-16)
error[E0384]: re-assignment of immutable variable `a`
 --> <anon>:3:5
  |
2 |     let a: i32 = 0;
  |         - first assignment to `a`
3 |     a = a + 1;
  |     ^^^^^^^^^ re-assignment of immutable variable

error: aborting due to previous error
```

```rust
fn main() {
    let mut a: i32 = 0;
    a = a + 1;
    println!("{}" , a);
}
```

```
rustc 1.14.0 (e8a012324 2016-12-16)
1
Program ended.
```

# Example: C is good

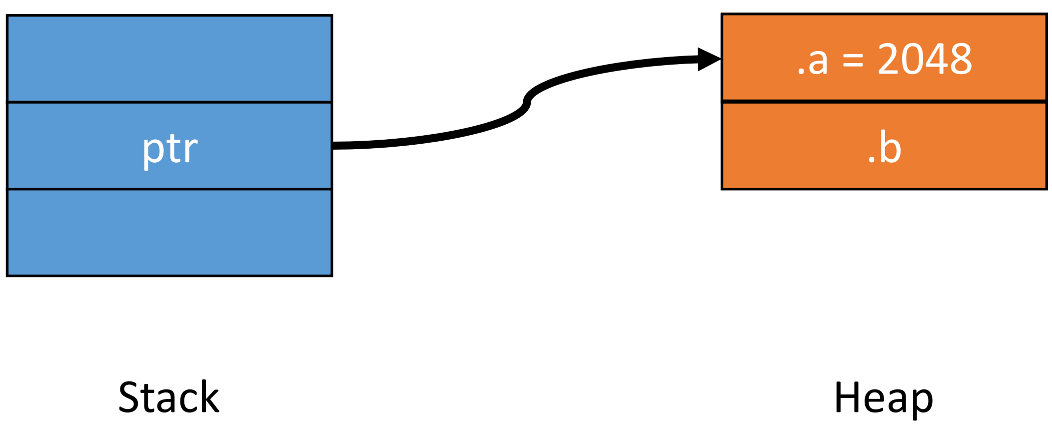
## Lightweight, low-level control of memory

```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    ptr->a = 2048;
    free(ptr);
}
```

*Precise memory layout*

*Lightweight reference*

*Destruction*



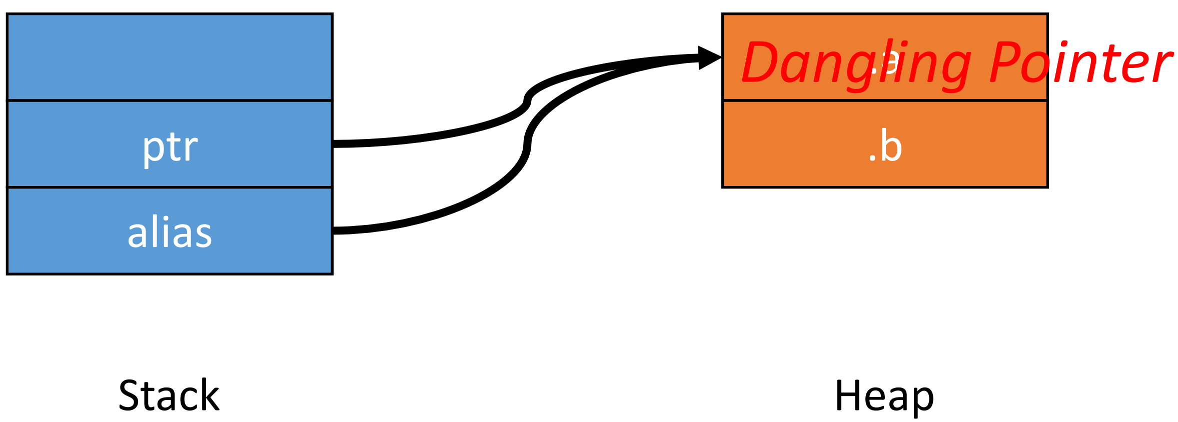
# Example: C is not so good

```c
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    Dummy *alias = ptr;
    free(ptr);
    int a = alias.a;
    free(alias);
}
```

*Use after free*

*Double free*

*Aliasing* ✚ *Mutation*



# Other problems with aliasing + mutation

- Make programs more confusing
- May disallow some compiler's optimizations

```
int a, b, *p, *q;
    ...
a = *p;  /* read from the variable referred to by p*/

*q = 3; /* assign to the variable referred to by q */

b = *p; /* read from the variable referred to by p */
```

- Cause for a long time of inefficiency of C versus FORTRAN compilers

# Solved by managed languages

Java, Python, Ruby, C#, Scala, Go...

- Restrict direct access to memory
- Run-time management of memory via periodic garbage collection
- No explicit malloc and free, no memory corruption issues
- But
  - Overhead of tracking object references
  - Program behavior unpredictable due to GC (bad for real-time systems)
  - Limited concurrency ("global interpreter lock" typical)
  - Larger code size
  - VM must often be included
  - Needs more memory and CPU power (i.e. not bare-metal)

# Requirements for system programs

- Must be fast and have minimal runtime overhead
- Should support direct memory access, but be memory –safe

Rust provides Box<T> to point to data on the Heap

- Boxes allow you to store data on the heap rather than the stack..
- Boxes don’t have performance overhead, other than storing their data on the heap instead of on the stack. But they don’t have many extra capabilities either.

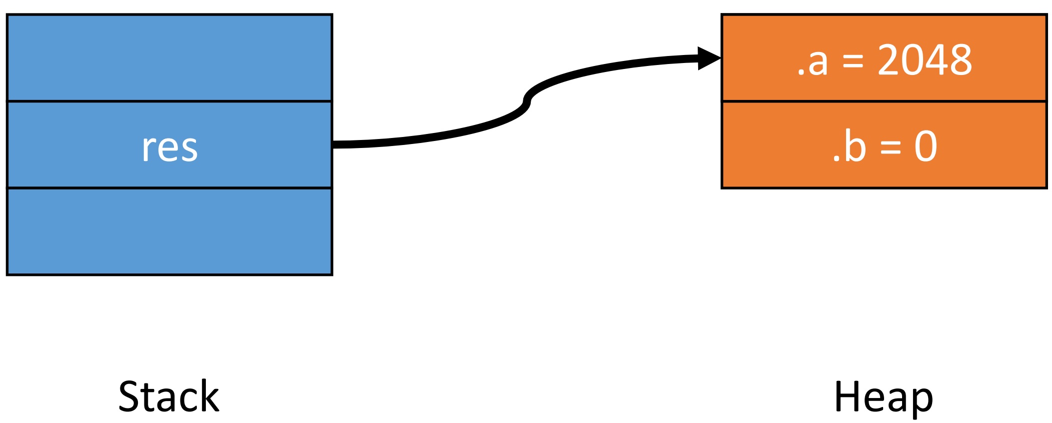
# Rust’s Solution: Zero-cost Abstraction

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res: Box<Dummy> = Box::new(Dummy {
                                  a: 0,
                                  b: 0
                              });
    res.a = 2048;
}
```

*Memory allocation*

*Variable binding*

*Resource owned by* res *is freed automatically*



# Side Slide: Type Inference

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res: Box<Dummy> = Box::new(Dummy {
                                 a: 0,
                                 b: 0
                             });
    res.a = 2048;
}
```

# Rust’s Solution: Ownership & Borrowing

*~~Aliasing~~* + *~~Mutation~~*

Compiler enforces:

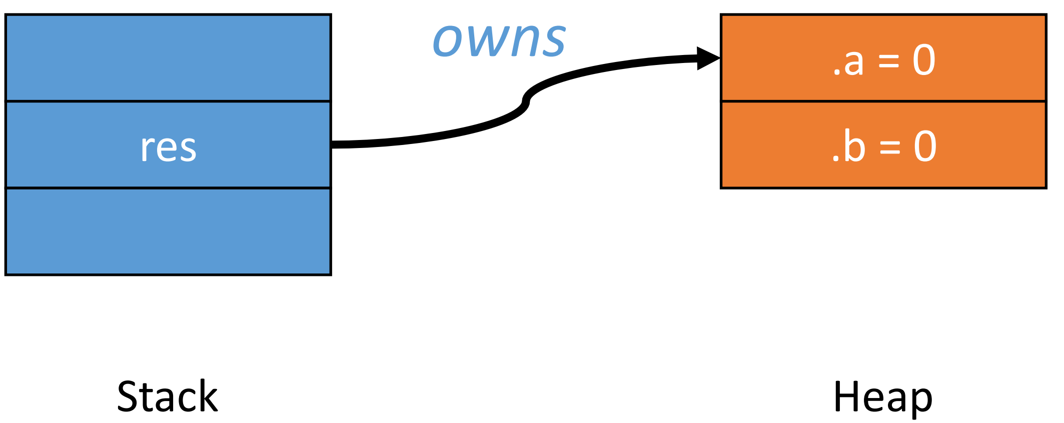
- Every resource has a unique *owner*.
- Others can *borrow* the resource from its owner.
- Owner *cannot* free or mutate its resource while it is borrowed.

No need for runtime Memory safety Data-race freedom

# Ownership

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                          a: 0,
                          b: 0
                      });
}
```

*res is out of scope and its resource is freed automatically*



# Ownership: Unique Owner

~~Aliasing~~ ✚ *Mutation*

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                        a: 0,
                        b: 0
                    });
    take(res);
    println!("res.a = {}", res.a);
}
```

*Compiling Error!*

*Ownership is moved from res to arg*

```
fn take(arg: Box<Dummy>) {
}
```

*arg is out of scope and the resource is freed automatically*

# Immutable/Shared Borrowing (&)

*Aliasing* ✚ ~~*Mutation*~~

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{
                            a: 0,
                            b: 0
                        });
    take(&res);
    res.a = 2048;
}
```

*Resource is returned from arg to res*

*Resource is immutably borrowed by arg from res*

```
fn take(arg: &Box<Dummy>) {
    arg.a = 2048;
}
```

*Compiling Error: Cannot mutate via an immutable reference*

*Resource is still owned by res. No free here.*

# Immutable/Shared Borrowing (&)

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{a: 0, b: 0});
    {
        let alias1 = &res;
        let alias2 = &res;
        res.a = 2048;
        let alias3 = alias2;
    }
    res.a = 2048;
}
```

- Read-only sharing

# Mutable Borrowing (&mut)

~~Aliasing~~ ✚ *Mutation*

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{a: 0, b: 0});

    take(&mut res);
    res.a = 4096;

    let borrower = &mut res;
    let alias    = &mut res;
}

fn take(arg: &mut Box<Dummy>) {
    arg.a = 2048;
}
```

*Mutably borrowed by arg from res*

*Multiple mutable borrowings are disallowed*

*Returned from arg to res*

# Concurrency & Data-race Freedom

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {a: 0, b: 0});

    std::thread::spawn(move || {
        let borrower = &mut res;
        borrower.a += 1;
    });

    res.a += 1;
}
```

*Spawn a new thread*

*res is mutably borrowed*

*Error: res is being mutably borrowed*

# Other smart pointers in Rust

| Type | Sharable? | Mutable? | Thread Safe? |
|---|---|---|---|
| & | yes * | no | no |
| &mut | no * | yes | no |
| Box | no | yes | no |
| Rc | yes | no | no |
| Arc | yes | no | yes |
| RefCell | yes ** | yes | no |
| Mutex | yes, in Arc | yes | yes |

* but doesn't own contents, so lifetime restrictions.

** while there is no mutable borrow

# Lifetimes

- A *lifetime* is a construct the compiler (or more specifically, its *borrow checker*) uses to ensure all borrows are valid.
- A variable's lifetime begins when it is created and ends when it is destroyed.
- Lifetimes are mostly inferred, but can be made explicit using generics
- The compiler checks that every borrowed variable/reference has a lifetime that is longer than the borrower

# Lifetime inference

```
// Lifetimes are annotated below with lines denoting the creation
// and destruction of each variable.
// `i` has the longest lifetime because its scope entirely encloses
// both `borrow1` and `borrow2`. The duration of `borrow1` compared
// to `borrow2` is irrelevant since they are disjoint.
fn main() {
    let i = 3; // Lifetime for `i` starts. ────────────────┐
    { //                                                   │
        let borrow1 = &i; // `borrow1` lifetime starts. ──┐│
        //                                                ││
        println!("borrow1: {}", borrow1); //              ││
    } // `borrow1 ends. ──────────────────────────────────┘│
    { //                                                   │
        let borrow2 = &i; // `borrow2` lifetime starts. ──┐│
        //                                                ││
        println!("borrow2: {}", borrow2); //              ││
    } // `borrow2` ends. ─────────────────────────────────┘│
}   // Lifetime ends. ─────────────────────────────────────┘
```

# Explicit Lifetimes

```
// `print_refs` takes two references to `i32` which have different
// lifetimes `'a` and `'b` (passed as generic parameters).
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}
// A function which takes no arguments, but has a lifetime parameter `'a`.
fn failed_borrow<'a>() {
    let _x = 12;
    // ERROR: `_x` does not live long enough
    // let y: &'a i32 = &_x;
    // The lifetime of `&_x` is shorter than that of `y`.
    // A short lifetime cannot be coerced into a longer one.
}
fn main() {
    let (four, nine) = (4, 9);  // Create variables to be borrowed below.
    print_refs(&four, &nine); // Borrows of both variables are passed
    // In other words, the lifetime of `four` and `nine` must
    // be longer than that of `print_refs`.
    failed_borrow();
}
```

# Other aspects of Rust: Enums

- Algebraic data types
- First-class
  - Instead of integers (C/C++)
- Structural
  - Parameters
  - Replacement of **union** in C/C++

# Enums

```rust
enum RetInt {
    Fail(u32),
    Succ(u32)
}

fn foo_may_fail(arg: u32) -> RetInt {
    let fail = false;
    let errno: u32;
    let result: u32;
    ...
    if fail {
        RetInt::Fail(errno)
    } else {
        RetInt::Succ(result)
    }
}
```

# Enums: No Null Pointers

```
enum std::option::Option<T> {
    None,
    Some(T)
}

struct SLStack {
    top: Option<Box<Slot>>
}

struct Slot {
    data: Box<u32>,
    prev: Option<Box<Slot>>
}
```

# Enums: Trees a ADT

```
#[derive(Debug)]
enum Tree<T> {
    Empty,
    Node(T, Box<Tree<T>>, Box<Tree<T>>)
}

fn main() {
    let tree = Tree::Node(
        42,
        Box::new(Tree::Node(
            0,
            Box::new(Tree::Empty),
            Box::new(Tree::Empty)
        )),
        Box::new(Tree::Empty));

    println!("{:?}", tree);
        // prints Node(42, Node(0, Empty, Empty), Empty)
}
```

# Pattern Match

```
let x = 5;

match x {
    1             => println!("one"),
    2             => println!("two"),
    3|4           => println!("three or four"),
    5 ... 10      => println!("five to ten"),
    e @ 11 ... 20 => println!("{}", e);
    _             => println!("others"),
}
```

*Compiler enforces the matching is complete*

# Pattern Match

```
enum std::option::Option<T> {
    None,
    Some(T)
}

struct SLStack {
    top: Option<Box<Slot>>
}

fn is_empty(stk: &SLStack) -> bool {
    match stk.top {
        None     => true,
        Some(..) => false,
    }
}
```

# Generic

```
struct SLStack {
    top: Option<Box<Slot>>
}

struct Slot {
    data: Box<u32>,
    prev: Option<Box<Slot>>
}

fn is_empty(stk: &SLStack) -> bool {
    match stk.top {
        None      => true,
        Some(..) => false,
    }
}
```

# Generic

```
struct SLStack<T> {
    top: Option<Box<Slot<T>>>
}

struct Slot<T> {
    data: Box<T>,
    prev: Option<Box<Slot<T>>>
}

fn is_empty<T>(stk: &SLStack<T>) -> bool {
    match stk.top {
        None      => true,
        Some(..) => false,
    }
}
```

*Type implementing this trait*

# Traits (Typeclass in Haskell)

*Object of the type implementing this trait*

```
trait Stack<T> {
    fn new() -> Self;
    fn is_empty(&self) -> bool;
    fn push(&mut self, data: Box<T>);
    fn pop(&mut self) -> Option<Box<T>>;
}

impl<T> Stack<T> for SLStack<T> {
    fn new() -> SLStack<T> {
        SLStack{ top: None }
    }

    fn is_empty(&self) -> bool {
        match self.top {
            None     => true,
            Some(..) => false,
        }
    }
}
```

```
struct SLStack<T> {
    top: Option<Box<Slot<T>>>
}

struct Slot<T> {
    data: Box<T>,
    prev: Option<Box<Slot<T>>>
}
```

# Using Traits for Bounded Polymorphism

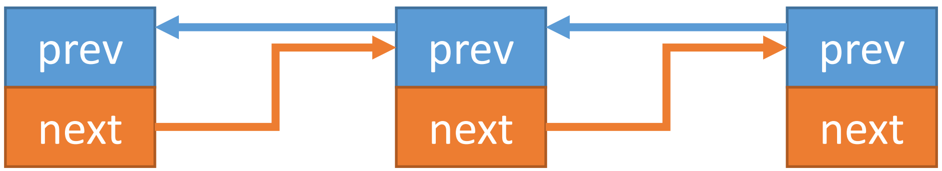
```
trait Stack<T> {
    fn new() -> Self;
    fn is_empty(&self) -> bool;
    fn push(&mut self, data: Box<T>);
   fn pop(&mut self) -> Option<Box<T>>;
}

fn generic_push<T, S: Stack<T>>(stk: &mut S,
                                data: Box<T>) {
    stk.push(data);
}

fn main() {
    let mut stk = SLStack::<u32>::new();
    let data = Box::new(2048);
    generic_push(&mut stk, data);
}
```

# Multiple traits as bounds

```
trait Clone {
    fn clone(&self) -> Self;
}

impl<T> Clone for SLStack<T> {
    ...
}

fn immut_push<T, S: Stack<T>+Clone>(stk: &S, data: Box<T>) -> S
{
    let mut dup = stk.clone();
    dup.push(data);
    dup
}

fn main() {
    let stk = SLStack::<u32>::new();
    let data = Box::new(2048);
    let stk = immut_push(&stk, data);
}
```

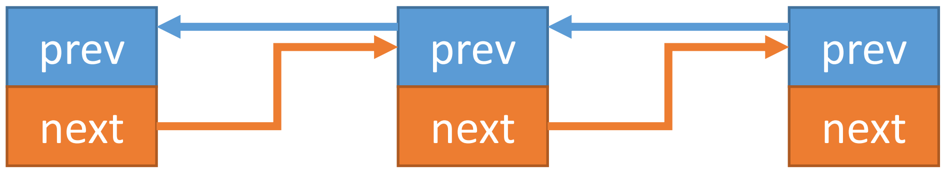
# And if Mutably Sharing is necessary?

- Mutably sharing is *inevitable* in the real world.
- Example: mutable doubly linked list




```
struct Node {
    prev: option<Box<Node>>,
    next: option<Box<Node>>
}
```

# Rust’s Solution: Raw Pointers




```
struct Node {
    prev: option<Box<Node>>,
    next: *mut Node
}
```

*Raw pointer*

- Compiler does *NOT* check the memory safety of most operations *wrt.* raw pointers.
- Most operations *wrt.* raw pointers should be encapsulated in a *unsafe* {} syntactic structure.

# Unsafe superpowers

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of unions

Note: *unsafe{}* does not switch off the borrow checker

# Rust’s Solution: Raw Pointers

```
let a = 3;

unsafe {
    let b = &a as *const i32 as *mut i32;
    *b = 4;
}

println!(“a = {}”, a);
```

*I know what I’m doing*

*Print “a = 4”*

# Foreign Function Interface (FFI)

- All foreign functions are unsafe.

```
extern {
    fn write(fd: i32, data: *const u8, len: u32) -> i32;
}

fn main() {
    let msg = "Hello, world!\n";
    unsafe {
        write(1, &msg[0], msg.len());
    }
}
```