C vs. Rust

C (the good parts)

Efficient code especially in resource-constrained environments

Direct control over hardware

Performance over safety

- Memory managed manually
- No periodic garbage collection
- Desirable for advanced programmers

But…

Type errors easy to make

- Integer promotion/coercion errors
- Unsigned vs. signed errors
- Integer casting errors

and...

Memory errors easy to make

- Null pointer dereferences
- Buffer overflows, out-of-bound access (no array-bounds chec king)
- Format string errors
- Dynamic memory errors
  - Memory leaks
  - Use-after-free (dangling pointer)
  - Double free

Cause software crashes and security vulnerabilities.

Example: C is good

Lightweight, low-level control of memory

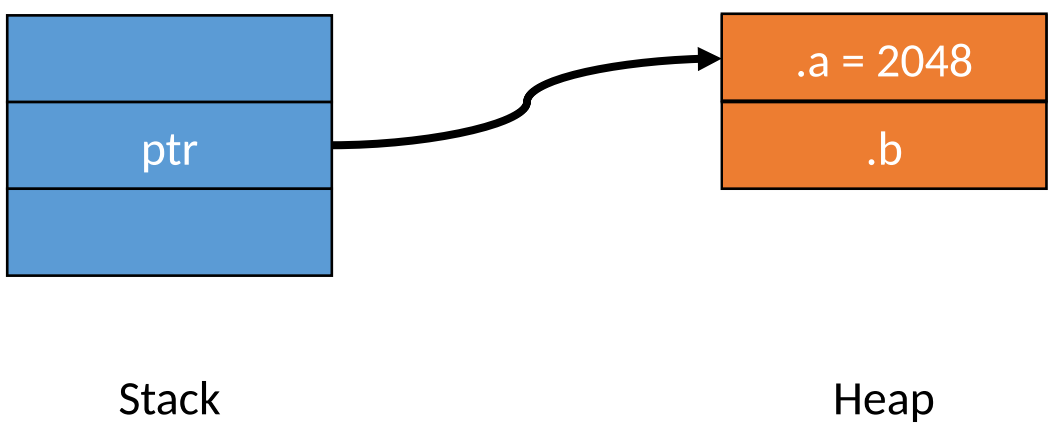
```c
typedef struct Dummy { int a; int b; } Dummy;

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

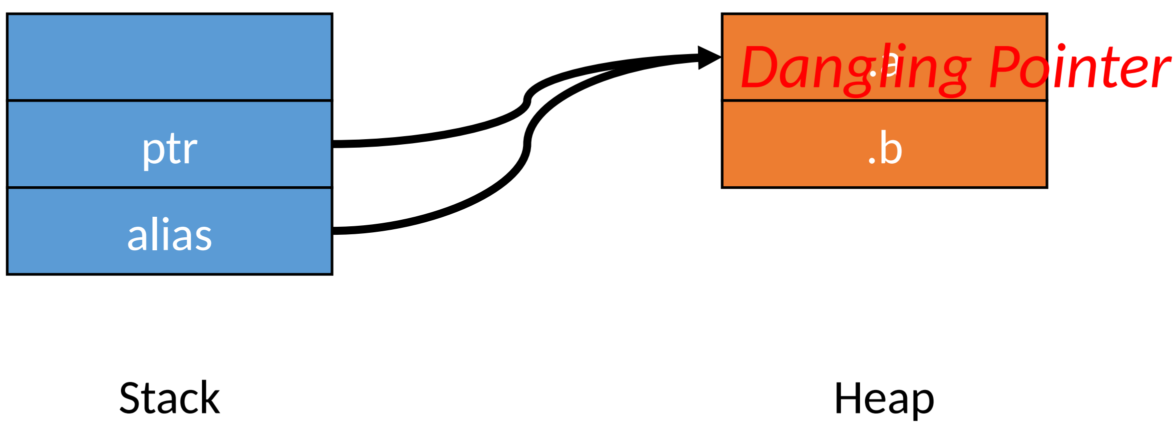
```c
typedef struct Dummy { int a; int b; } Dummy;

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*



Solved by managed languages

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

- Restrict direct access to memory
- Run-time management of memory via periodic garbage c ollection
- 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

Rust

From the official website (http://rust-lang.org):

Rust is a *system programming language* barely on *hardware*.
- No *runtime* requirement (runs fast)
- *Control over* memory allocation/destruction.
- Guarantees memory safety

Developed to address severe memory leakage and corruption bugs in Firefox
- First stable release in 5/2015

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’s type system

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

Rust and typing

Arrays stored with their length `[T; N]`

- Allows for both compile-time and run-time checks on arra y access via `[]`

C

```
void main(void) {
    int nums[8] =
{1,2,3,4,5,6,7,8};
    for ( x = 0; x < 10; i++ )
    printf(“%d\n”,nums[i]);
}
```

Rust

```
fn main() {
    let nums = vec![1,2,3,4,5,6,7,8];
    for x in 0..10 {
        println!("{}",nums[x]);
    }
}
```

```
7
8
thread 'main' panicked at 'index out of bounds: the len is 8 but the index is 8',
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Program ended.

Rust and bounds checking

But…

- Checking bounds on every access adds overhead

```rust
fn main() {
     let nums = vec![1,2,3,4,5,6,7,8];
     for x in 0..10 {
         println!("{}",nums[x]);
     }
}
```

- Arrays t … erators
  - Can us

```rust
fn main() {
     let nums = vec![1,2,3,4,5,6,7,8];
     for num in &nums {
         println!("{}",num);
     }
}
```

Rust vs C typing errors

Recall issues with implicit integer casts and promotion in C

`-1 > 0U`

`2147483647U  < -2147483648`

Rust's type system prevents such comparisons

```c
int main() {
    unsigned int a = 4294967295;
    int b = -1;
    if (a == b)
        printf("%u == %d\n",a,b);
}
```

```
mashimaro <~>  9:44AM % ./a.out
4294967295 == -1
```

```rust
fn main() {
    let a:u32 = 4294967295;
    let b:i32 = -1;
    if a == b {
        println!("{} == {}", a, b);
    }
}
```

```
rustc 1.15.1 (021bd294c 2017-02-08)
error[E0308]: mismatched types
 --> <anon>:4:13
  |
4 |     if a == b {
error: aborting due to previous error
```

Rust vs C typing errors

Same or different?

```c
int main() {
    char a=251;
    unsigned char b = 251;
    printf("a = %x\n", a);
    printf("b = %x\n", b);

    if (a == b)
        printf("Same\n");
    else
        printf("Not Same\n");
}
```

```
mashimaro<> % ./a.out
a = fffffffb
b = fb
Not Same
```

```rust
fn main() {
    let a:i8 = 251;
    let b:u8 = 251;

    if a == b {
        println!("Same");
    } else {
        println!("Not Same");
    }
}
```

```
rustc 1.15.1 (021bd294c 2017-02-08)
error[E0308]: mismatched types
 --> <anon>:5:13
  |
5 |     if a == b {
error: aborting due to previous error
```

Rust vs C typing errors

201 > 200?

```c
#include <stdio.h>
int main() {
    unsigned int ui = 201;
    char c=200;
    if (ui > c)
        printf("ui(%d) > c(%d)\n",ui,c);
    else
        printf("ui(%d) < c(%d)\n",ui,c);
}
```

```
mashimaro <~> 12:50PM % ./a.out
ui(201) < c(-56)
```

```rust
fn main() {
    let ui:u32 = 201;
    let c:i8 = 200;
    if ui > c {
        println!("ui({}) > c({})",ui,c);
    } else {
        println!("ui({}) < c({})",ui,c)
    }
}
```

```
rustc 1.15.1 (021bd294c 2017-02-08)
error[E0308]: mismatched types
 --> <anon>:4:13
  |
4 |     if ui > c {
```

Rust vs C typing errors

In Rust, casting allowed via the “as” keyword

- Follows similar rules as C
- But, warns of literal problem before performing the prom otion with sign extension

```c
#include <stdio.h>
int main() {
  char c=128;
  unsigned int uc;
  uc = (unsigned int) c;
  printf("%x %u\n",uc, uc);
}
```

```
mashimaro <~>  1:24PM % ./a.out
ffffff80 4294967168
```

```rust
fn main() {
    let c:i8 = 128;
    let uc:u32 = c as u32;
    println!("uc = {}", uc);
}
```

```
rustc 1.15.1 (021bd294c 2017-02-08)
warning: literal out of range for i8, #
default
 --> <anon>:2:16
  |
2 |     let c:i8 = 128;
uc = 4294967168
```

Rust vs C typing errors

Recall issues with unchecked underflow and overflow

- Silent wraparound in C

```
int main() {
    unsigned int a = 4;
    a = a - 3;
    printf("%u\n",a-2);
}
mashimaro <~>  9:35AM % ./a.out
4294967295
```

-

```
fn main() {
    let mut a:u32 = 4;
    a = a - 3;
    println!("{}", a - 2);
}
```

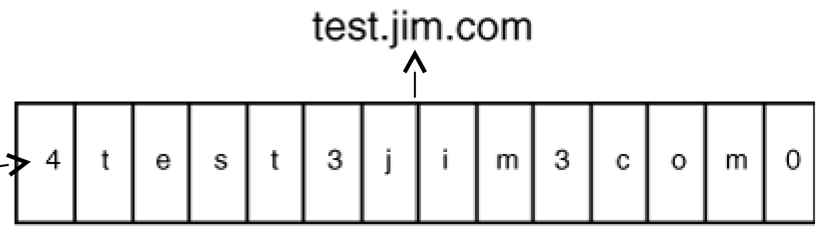
```
rustc 1.15.1 (021bd294c 2017-02-08)
thread 'main' panicked at 'attempt to subtract with overflow',
stack backtrace:
```

Recall previous C vulnerability

DNS parser vulnerability

- `count` read as byte, then `count` bytes concatenated to `nameStr`

```
char *indx;
int count;
char nameStr[MAX_LEN]; //256
...
memset(nameStr, '\0', sizeof(nameStr));
...
indx = (char *)(pkt + rr_offset);
count = (char)*indx;
while (count){
   (char *)indx++;
   strncat(nameStr, (char *)indx, count);
   indx += count;
   count = (char)*indx;
   strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
}
nameStr[strlen(nameStr)-1] = '\0';
```



What if count = 128?

Sign extended then used in strncat

Type mismatch in Rust

```
char *strncat(char *dest, const char *src, size_t n);
```

http://www.informit.com/articles/article.aspx?p=686170&seqNum=6

Another C vulnerability

2002 FreeBSD `getpeername()` bug (B&O Ch. 2)

- Kernel code to copy hostname into user buffer
  - `copy_from_kernel()` call takes signed `int` for size from user
  - `memcpy` call uses unsigned `size_t`
- What if adversary gives a length of "-1" for his buffer size?

```
#define KSIZE 1024
char kbuf[KSIZE]
void *memcpy(void *dest, void *src, size_t n);

int copy_from_kernel(void *user_dest, int maxlen){
        /* Attempt to set len=min(KSIZE, maxlen) */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
        memcpy(user_dest, kbuf, len);
        return len;
}
```

Type mismatch in Rust

`(KSIZE < -1)` is false, so `len = -1`

`memcpy` casts `-1` to 2^{32}`-1`

Unauthorized kernel memory copied out

Rust's Ownership & Borrowing

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

Compiler enforced:

- Every resource has a unique *owner*.
- Others can *borrow* the resource from its owner (e.g. create an *alias*) with restrictions
- Owner *cannot* free or mutate its resource while it is borrowed.

No need for runtime | Memory safety | Data-race freedom

But first...mutability

By default, Rust variables are immutable

- Usage checked by the compiler

mut is used to declare a resource as mutable.

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

http://is.gd/OQDszP

```
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.
```

Ownership and lifetimes

There can be only one “owner” of an object

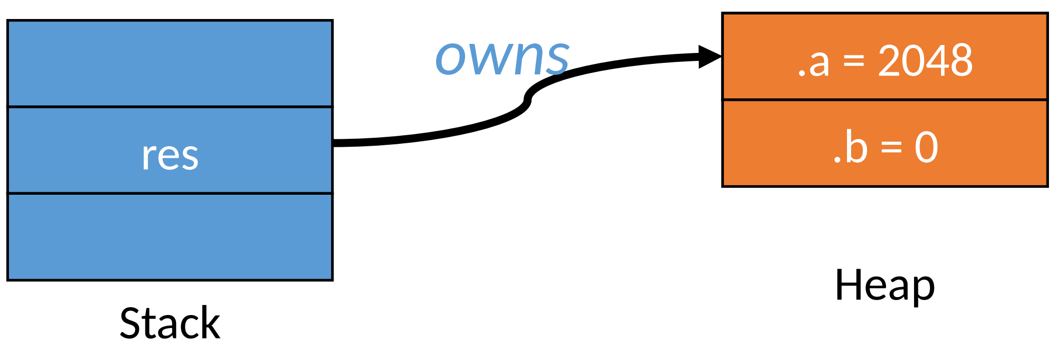
- When the “owner” of the object goes out of scope, its data is automa tically freed
- Can not access object beyond its lifetime (checked at compile-time)

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

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

Memory allocation

Resource owned by res *is freed automatically*



Assignment changes ownership

```rust
//#[derive(Clone)]
struct Point { x: i32, y: i32 }

fn main() {
    let a = Point { x: 1, y: 2};
    let b = a;

    println!("{}, {}", a.x, a.y);
}
```

http://is.gd/pZKiBw

```
rustc 1.15.1 (021bd294c 2017-02-08)
error[E0382]: use of moved value: `a.x`
 --> <anon>:8:24
  |
6 |     let b = a;
  |         - value moved here
7 |
8 |     println!("{}, {}", a.x, a.y);
  |                        ^^^ value used here after move
  |
```

Ownership transfer in function calls

```
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);
}
```

Compiler Error!

Ownership is moved from res to arg

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

arg is out of scope and the resource is freed automatically

Borrowing

You can borrow ownership of an object in order to modif y it with some restrictions

- You cannot borrow mutable reference from immutable object
  - Or mutate an object immutably borrowed
- You cannot borrow more than one mutable reference (atomic ity)
  - You can borrow an immutable reference many times
- There cannot exist a mutable reference and an immutable on e simultaneously (removes race conditions)
- The lifetime of a borrowed reference should end before the lif etime of the owner object does (removes use after free)

Borrowing example

You cannot borrow mutable reference from immutab le object

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

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

    res.a = 2048;

    let borrower = &mut res;
}
```

Error: Resource is immutable

Error: Cannot get a mutable borrowing of an immutable resource

Borrowing example (&)

You cannot mutate an object immutably borrowed

```
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;
}
```

Compiler Error: Cannot mutate via an immutable reference

Resource is still owned by res. No free here.

Borrowing example (&mut)

You cannot borrow more than one mutable reference

~~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

Immutable, shared borrowing (&)

You can borrow more than one immutable reference

- But, there cannot exist a mutable reference and an immutable one s imultaneously

Aliasing ✚ ~~Mutation~~

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

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

Finally,

The lifetime of a borrowed reference should end befo
re the lifetime of the owner object does

Use-after free in C

```c
void some_dumb_function(){
    int *used_after_free = malloc(sizeof(int));

    /* ... after use */
    free(used_after_free);


    /* what the... */
    printf("%d", *used_after_free);
}
```

Memory allocated to int

Then freed

Then used after free

If these calls are far away from each other,
this bug can be very hard to find.

Caught by Rust at compile-time

Unique ownership, borrowing, and lifetime rules easil y enforced

```
fn main() {
    // This binding lives in the main function
    let name = String::from("Hello world!");
    let mut name_ref = &name;
    {
        let newname = String::from("Goodbye!");
        name_ref = &newname;
    }
    println!("name is {}", &name_ref);
}
```

```
rustc 1.15.1 (021bd294c 2017-02-08)
error: `newname` does not live long enough
  --> <anon>:8:5
   |
7  |         name_ref = &newname;
   |                     ------- borrow occurs here
8  |     }
9  |     println!("name is {}", &name_ref);
10 | }
   | - borrowed value needs to live until here
error: aborting due to previous error
```

Dangling pointer in C

Recall scoping issues example (B&O Ch 3, Procedures)

```
int* func(int x) {
     int n;
     int *np;
     n = x;
     np = &n;
     return np;
}
```

Local variable is allocated in stack, a temporal storage of function.

Reference returned, but variable now out of scope (dangling pointer)

What does `np` point to after function returns?

What happens if `np` is dereferenced after being returned?

http://thefengs.com/wuchang/courses/cs201/class/08/invalid_ref.c

Caught by Rust at compile-time

Ownership/Borrowing rules ensure objects are not accessed beyond lifetime

```rust
fn a_dumb_function() -> &i32 {
    let local_variable: i32;

    &local_variable
}

fn main() {
    let raw_pointer = a_dumb_function();

    *raw_pointer = 123;
}
```

borrowed pointer
cannot outlive
the owner!!

http://is.gd/3MTsSC

```
rustc 1.15.1 (021bd294c 2017-02-08)
error[E0106]: missing lifetime specifier
 --> <anon>:1:25
  |
1 | fn a_dumb_function() -> &i32 {
  |                         ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value,
  = help: consider giving it a 'static lifetime

error: aborting due to previous error
```

Summary

Languages offer trade-offs in terms of performance, ease of use, and safety

- Learn to be multi-lingual
- Learn how to choose wisely

Sources

- Haozhong Zhang "An Introduction to Rust Programming Language"
- Aaron Turon, *The Rust Programming Language*, Colloquium on Computer Systems Seminar Series (EE380) , Stanford University, 2015.
- Alex Crichton, *Intro to the Rust programming language*, http://people.mozilla.org/~acrichton/rust-talk-2014-12-10/
- *The Rust Programming Language*, https://doc.rust-lang.org/stable/book/
- Tim Chevalier, "Rust: A Friendly Introduction", 6/19/2013

Resources

- Rust website: http://rust-lang.org/
  - Playground: https://play.rust-lang.org/
  - Guide: https://doc.rust-lang.org/stable/book/
  - User forum: https://users.rust-lang.org/
  - Book: https://doc.rust-lang.org/stable/book/academic-research.html
- IRC: server: *irc.mozilla.org*, channel: *rust*
- Cargo: https://crates.io/
- Rust by example: http://rustbyexample.com/

Extra

Ownership and borrowing example

```rust
fn main() {
    let mut v = vec![];

    v.push("Hello");

    let x = &v[0];

    v.push("world");

    println!("{}", x);
}
```

v is an owner of the vector

x borrows the vector from v

now v cannot modify the vector
because it lent the ownership to x

http://is.gd/dEamuS

More than that …

C/C++ Haskell/Python

more control,
less safety

less control,
more safety

Rust

more control,
more safety

Concurrency & Data-race Freedom

```rust
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;
}
```

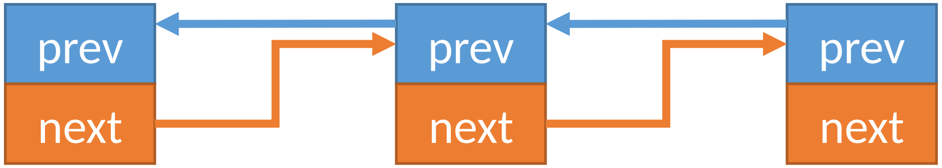
Spawn a new thread

res is mutably borrowed

Error: res is being mutably borrowed

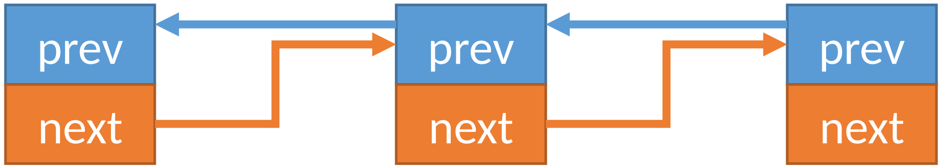
Mutably Sharing

- 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.

Rust's Solution: Raw Pointers

```
let a = 3;

unsafe {
    let b = &a as *const u32 as *mut u32;
    *b = 4;
}

println!("a = {}", a);
```

I know what I'm doing

Print "a = 4"

Unsafe

Life is hard.

Foreign Function Interface (FFI)

All foreign functions are unsafe (e.g. libc calls)

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

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

Inline Assembly is unsafe

```
#![feature(asm)]
fn outl(port: u16, data: u32) {
    unsafe {
        asm!("outl %0, %1"
             :
             : "a" (data), "d" (port)
             :
             : "volatile");
    }
}
```