# little languages

## lecture 08: Mutability and Lifetimes

Before lecture: **Start VM and pull 590 materials from upstream**.
Then...

```
$ cd comp590-material-<you>
$ git pull upstream master
$ cd 590-material-<you>/lecture/<today>
```

# Learning a new language...

- Getting started in a new language with novel ideas is *difficult*
- The best way I know how is by reading through reference to get some context and trying out lots of little programs
- After you have a baseline understanding of the language, then you can figure out the many details on-demand as needed
- The opening GRQs are about reaching to a baseline understanding of Rust and readings will slow down now that we've seen the fundamentals

# 1. What is the final line of output of this Java program?

Java

```
public class Point {

    int x, y;

    public static void main(String[] args) {
        Point p = new Point(1, 2);
        PointPrinter.print(p);
            // Outputs x: 1, y: 2
        p.x++;
        p.y++;
        PointPrinter.print(p);
            // Outputs: ?
    }

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

# 2. What is the final line of output of this Java program?

Java

```java
public class Point {

    int x, y;

    public static void main(String[] args) {
        Point p = new Point(1, 2);
        PointPrinter.print(p);
          // Outputs x: 1, y: 2
        System.out.println("x:" + p.x + " y:" + p.y);
          // Outputs x: 1, y: 2
          // Wait one millisecond...
        try{Thread.sleep(1);}catch(Exception e){}
        System.out.println("x:" + p.x + " y:" + p.y);
         // Outputs: ???
    }

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

# "[In Rust,] a function’s signature always exposes the body’s behavior."

- While these scenarios are clearly fabricated for demonstration, bugs whose root causes are possible due to these properties are real in most popular languages.

- In Rust, suppose the helper function had the following signature:

  ```
  fn point_printer<'f>(p: &'f Point) -> () {}
  ```

- The ***immutable reference*** promises you this function *will not mutate* its referent.
  - This guarantee would have prevented surprise in the first Java example.

- The ***lifetime annotation*** promises you this function *will not store/use* the reference ***p*** anywhere that outlives the lifetime of a call to *point_printer*
  - The lifetime of a call starts when control jumps into a function and ends when it returns.
  - Even if the signature of the second example was a mutable reference instead of immutable, this guarantee would have prevented compilation of the second Java example.
  - The annotations in the example above are optional and the default lifetimes.

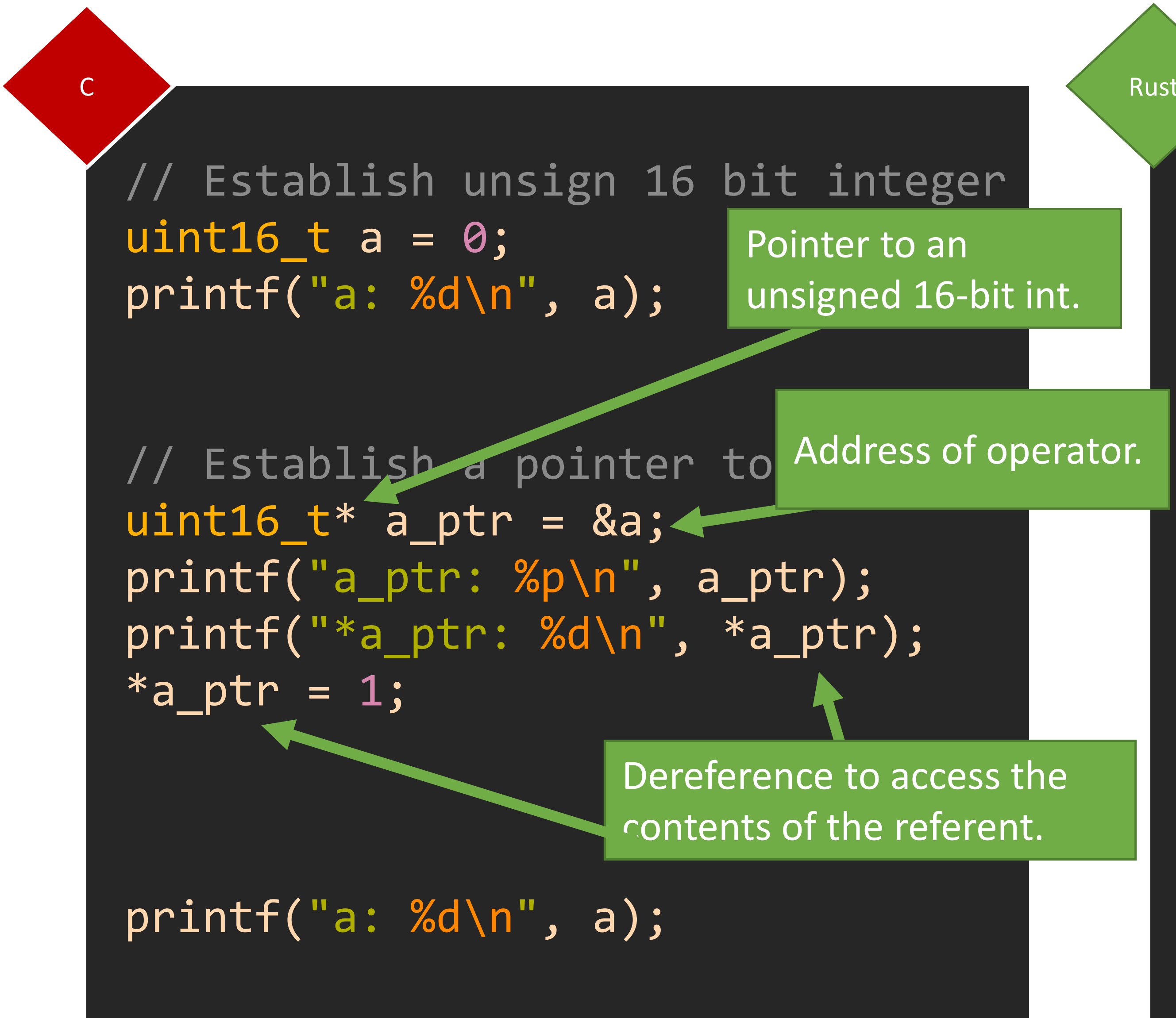
# Systems Programming in C

- C is the *de facto* systems programming language since the '70s

- Some consider the C programming language to be a little language
  - The case for:
    - Its syntax and feature set is tiny compared to most general purpose programming languages (GPLs)
    - A basic C interpreter / compiler would be much easier to implement than most GPLs
  - The case against:
    - It is very much a GPL!
    - For our purposes we're defining *little languages* as *domain-specific languages*
      - Regular expressions are specific to the domain of text pattern matching
      - VIM's command language is specific to text editing

- In 590 we will explore C through the lens of Rust
  - This will help motivate the reasons for features in Rust like lifetimes and mutability
  - This will also help you be more cognizant of dangerous scenarios in C
  - After 590, if you find yourself writing C, you should:
    1. Ask "could I do this in Rust?" If the answer is *no*, then there are reasons *you should not* do it in C!
    2. Ask "how would this be architected in Rust?" Try to mimic that architecture in C!

# Pointers in C and their counterparts in Rust



C

```c
// Establish unsign 16 bit integer
uint16_t a = 0;
printf("a: %d\n", a);


// Establish a pointer to
uint16_t* a_ptr = &a;
printf("a_ptr: %p\n", a_ptr);
printf("*a_ptr: %d\n", *a_ptr);
*a_ptr = 1;


printf("a: %d\n", a);
```

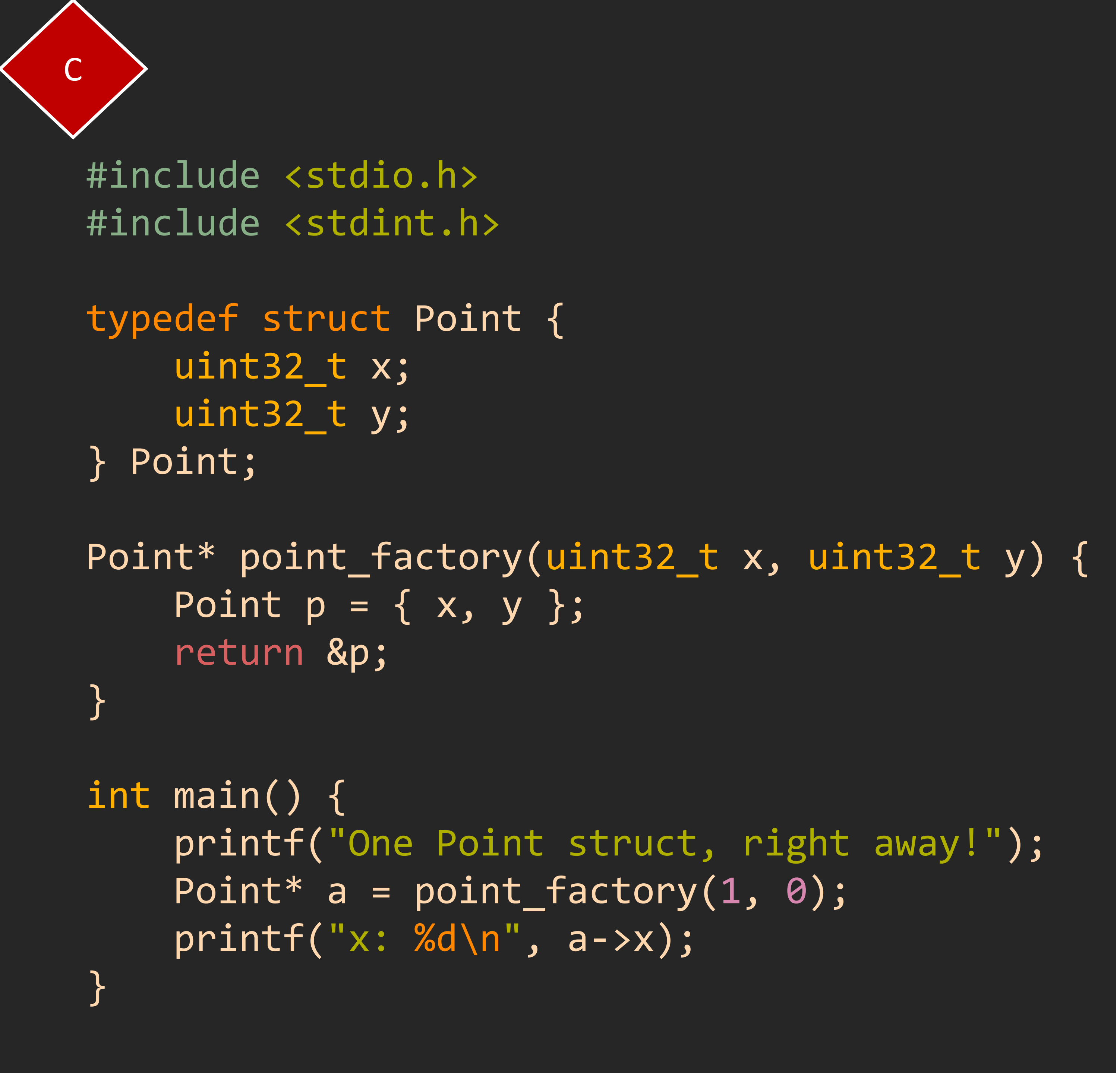
Pointer to an unsigned 16-bit int.

Address of operator.

Dereference to access the contents of the referent.

Rust

```rust
let mut a: u16 = 0;
println!("a: {}", a);

{
    // Take mutable ref inside block...
    let a_ptr: &mut u16 = &mut a;
    println!("a_ptr: {:p}", a_ptr);
    println!("*a_ptr: {}", *a_ptr);
    *a_ptr = 1;
    // Close block to read a again
}

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



C

```c
#include <stdio.h>
#include <stdint.h>

typedef struct Point {
    uint32_t x;
    uint32_t y;
} Point;

Point* point_factory(uint32_t x, uint32_t y) {
    Point p = { x, y };
    return &p;
}

int main() {
    printf("One Point struct, right away!");
    Point* a = point_factory(1, 0);
    printf("x: %d\n", a->x);
}
```

# Given this C code...

- Does it compile?
- Does it run?
- What is its output?

# The Lifetime of Stack Values

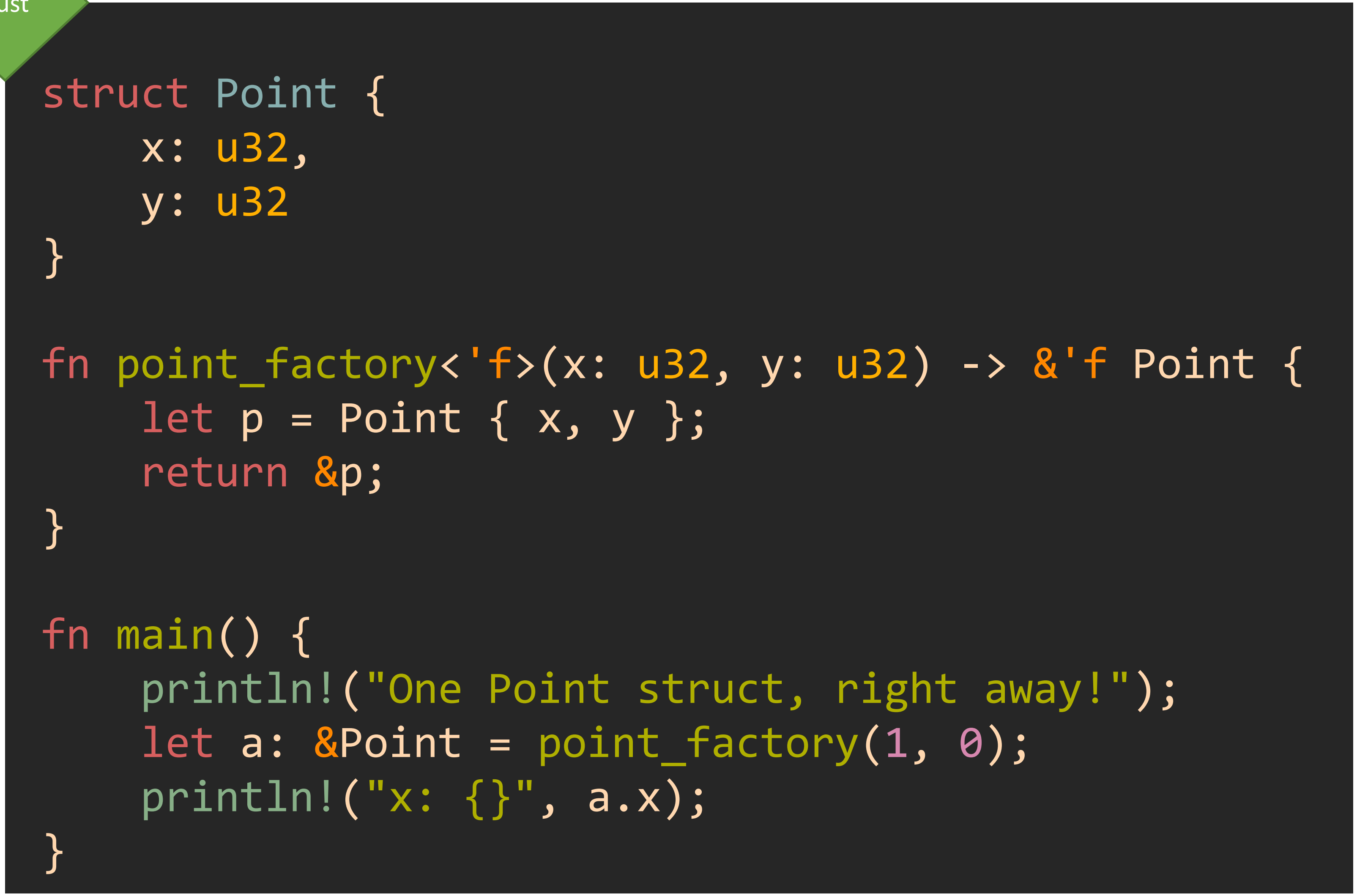
- The lifetime of a stack frame is the lifetime of a function call
  - Starts when control *jumps into* the function
  - Inclusive of nested function calls (each has its own frame)
  - Ends when control *returns from* the function

- Parameters' and local variables' lifetimes are same as their owning stack block's
  - Usually their stack block is same lifetime as stack frame, but in nested blocks shorter than fn
  - Reminder: the *owning block of a value* and the *owning block of a reference to the value* are only useful when they're not the same owner

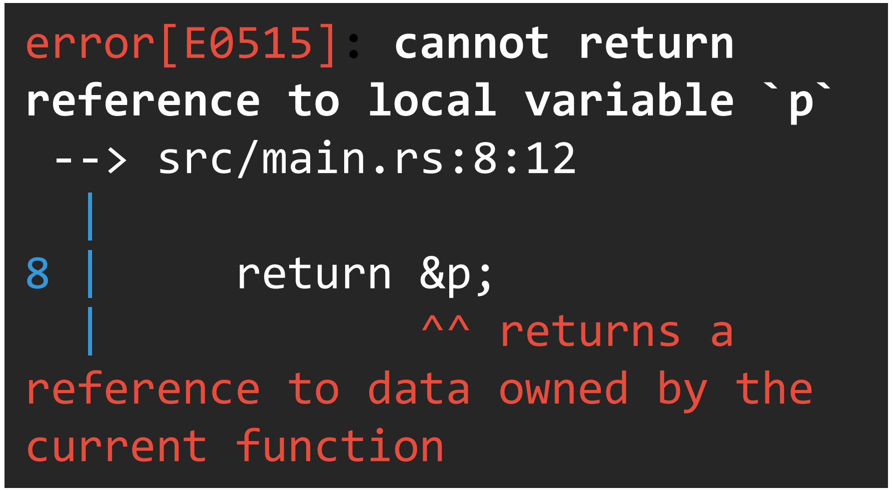
- When function call returns its parameters and local variables are dropped
  - If those values were *owned by* the function's stack frame, they're *dropped (freed from memory)*
  - Exception: when a value is returned its ownership (and lifetime) transfers to the caller's frame

# The Previous Example in Rust with Lifetimes

Rust



```rust
struct Point {
    x: u32,
    y: u32
}

fn point_factory<'f>(x: u32, y: u32) -> &'f Point {
    let p = Point { x, y };
    return &p;
}

fn main() {
    println!("One Point struct, right away!");
    let a: &Point = point_factory(1, 0);
    println!("x: {}", a.x);
}
```



```
error[E0515]: cannot return
reference to local variable `p`
 --> src/main.rs:8:12
  |
8 |     return &p;
  |            ^^ returns a
reference to data owned by the
current function
```

- Without using *unsafe*, Rust *will not* let you compile a program that has a "dangling pointer"

- In this case, the dangling pointer is the result of establishing a local variable in a stack frame and attempting to return its address to the function caller.

- The value p gets dropped when its owning frame returns, but in trying to return &p and give ownership of the &p value back to main its lifetime exceeds its referent's.

# What about **thdc**'s Tokenizer Struct's Lifetime?

```rust
pub struct Tokenizer<'s> {
    chars: Peekable<Chars<'s>>,
}
```

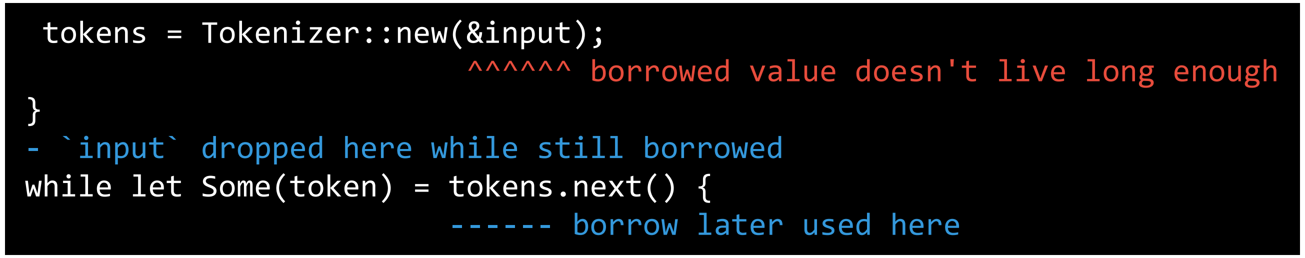
- The lifetime of a `Tokenizer` struct is *dependent upon* the lifetime of the `Peekable Chars` iterator's `&str` input's.
- In other words, a `Tokenizer` is only safe to use for as long as the value of its input `&str` is alive and valid.

```rust
let input = String::from("1 2 +");
let mut tokens = Tokenizer::new(&input);
while let Some(token) = tokens.next() {
    println!("nom nom nom...");
}
```

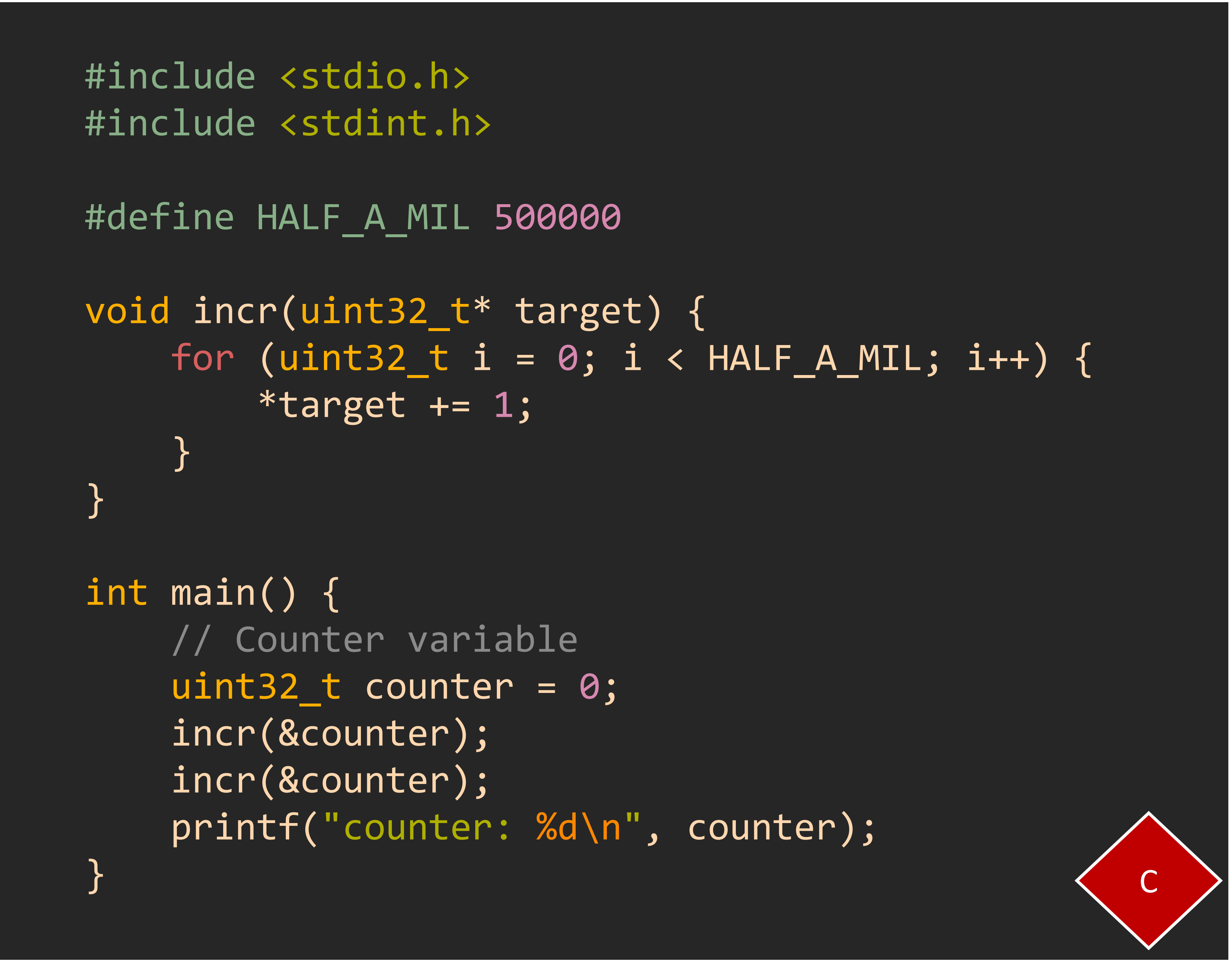
- This usage is **OK** because `&input` is declared in the same lifetime (scope) as the Tokenizer.
- Since *drops* happen in reverse order of *declarations,* the `Tokenizer` will be dropped first, then the input `String`.
- Note: If this is in a function you *could not return* the `Tokenizer`.

```rust
let mut tokens: Tokenizer;
{
    let input = String::from("1 2 +");
    tokens = Tokenizer::new(&input);
}
while let Some(token) = tokens.next() {
    println!("nom nom nom...");
}
```

- This usage is ***NOT OK*** because the `Tokenizer's` lifetime is longer than its input `String's`.



```
 tokens = Tokenizer::new(&input);
                         ^^^^^^ borrowed value doesn't live long enough
}
- `input` dropped here while still borrowed
while let Some(token) = tokens.next() {
                        ------ borrow later used here
```



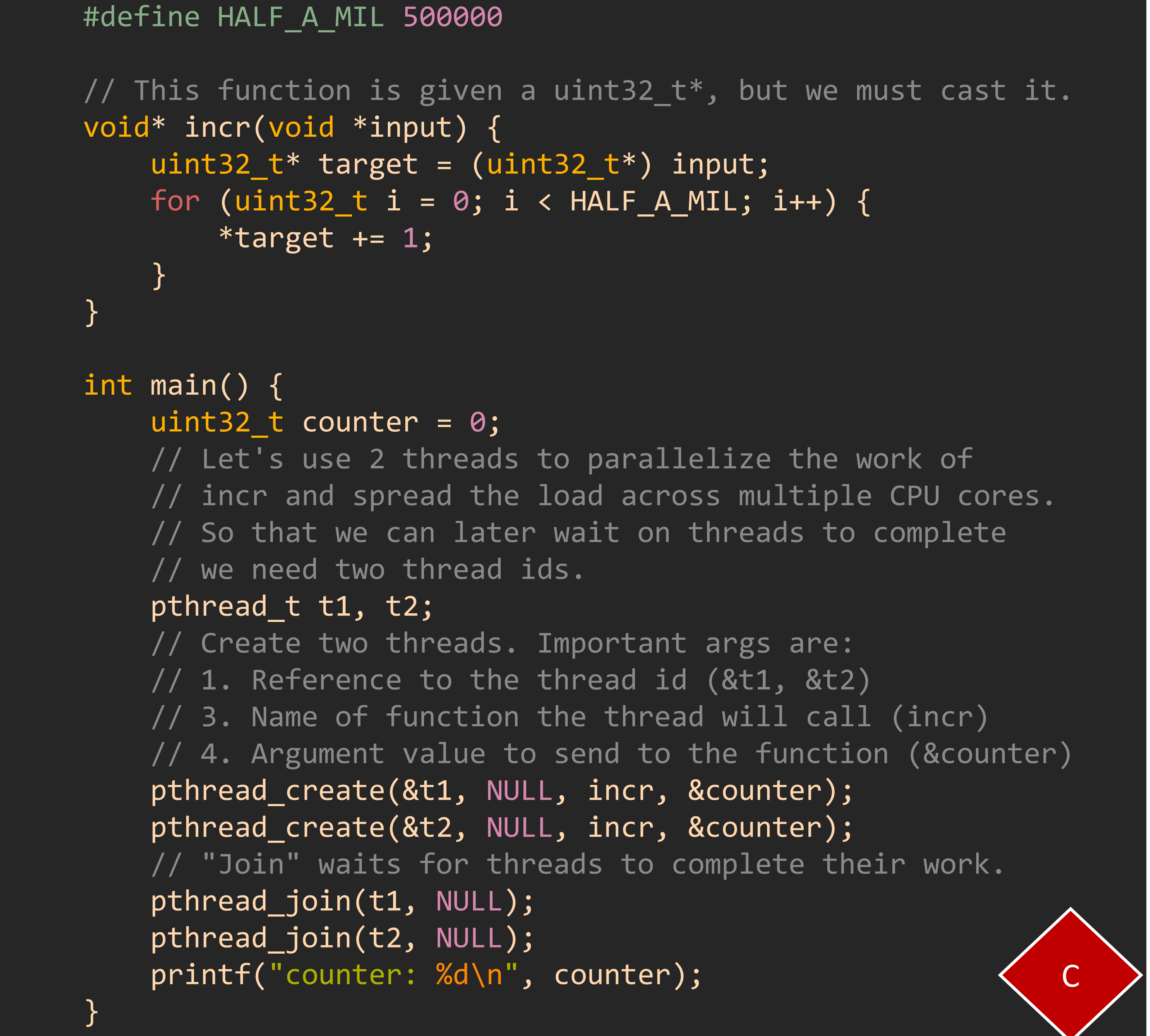
```c
#include <stdio.h>
#include <stdint.h>

#define HALF_A_MIL 500000

void incr(uint32_t* target) {
    for (uint32_t i = 0; i < HALF_A_MIL; i++) {
        *target += 1;
    }
}

int main() {
    // Counter variable
    uint32_t counter = 0;
    incr(&counter);
    incr(&counter);
    printf("counter: %d\n", counter);
}
```

# Given this C code...

- Does it compile?
- Does it run?
- What is its output?

# Given this C code...

- Does it compile?
- Does it run?
- What is its output?



```
#define HALF_A_MIL 500000

// This function is given a uint32_t*, but we must cast it.
void* incr(void *input) {
    uint32_t* target = (uint32_t*) input;
    for (uint32_t i = 0; i < HALF_A_MIL; i++) {
        *target += 1;
    }
}

int main() {
    uint32_t counter = 0;
    // Let's use 2 threads to parallelize the work of
    // incr and spread the load across multiple CPU cores.
    // So that we can later wait on threads to complete
    // we need two thread ids.
    pthread_t t1, t2;
    // Create two threads. Important args are:
    // 1. Reference to the thread id (&t1, &t2)
    // 3. Name of function the thread will call (incr)
    // 4. Argument value to send to the function (&counter)
    pthread_create(&t1, NULL, incr, &counter);
    pthread_create(&t2, NULL, incr, &counter);
    // "Join" waits for threads to complete their work.
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("counter: %d\n", counter);
}
```

# "I feel the Earth move under my feet..." - Carole King (*on multithreading and mutable references*)

- Multithreading and shared mutable references mix like water and lithium
- Everything you think you've learned about the determinism of programming goes out the door when you have multiple threads that can write to the same location in memory and you do nothing to protect its access.
- A **race condition** is a behavior that's dependent on the *timing* of sequential operations.
  - Race condition was at play in the 2nd Java example and in the threaded counter demo.
  - When you have a race condition you get non-deterministic behavior *at runtime*.
  - These are often ***heisenbugs*** that don't exist when you slow down the program in a debugger.
- Rust's *hard rule* about every value having only one mutable owner at any point in time is what protects you from ever facing race conditions in safe, multithreaded Rust.