Secure programming languages

Secure Programming Languages

- The design of secure programming languages is an evolving field that addresses how language features and abstractions can help prevent or mitigate security vulnerabilities in software. T
- he goal is to ensure that the language itself enforces, or at least facilitates, the construction of secure softare by design rather than relying entirely on external tools or developer discipline.

Design of secure PL: it is a trend in research to have security by design and constructs to manage security.

Secure Programming Languages: Key issues

in designing secure PL

Memory Safety

Languages like C and C++ allow low-level memory manipulation, which is powerful but dangerous.

Common vulnerabilities: buffer overflows, use-after-free, and null pointer dereferencing.

Use-after-free

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* ptr = malloc(sizeof(int));
    *ptr = 42;
    free(ptr);        // Memory is freed
    printf("%d\n", *ptr); // Use-after-free: undefined behavior
    return 0;
}
```

Null pointer

```
#include <stdio.h>

int main() {
    int* ptr = NULL;
    printf("%d\n", *ptr);
// Null pointer dereference: crashes the program
    return 0;
}
```

Secure Programming Languages: Key issues

Memory Safety

Languages like C and C++ allow low-level memory manipulation, which is powerful but dangerous.

Common vulnerabilities: buffer overflows, use-after-free, and null pointer dereferencing.

Solution: Use of memory-safe languages like **Rust**, **Java**, or **Go**, which manage memory via safe abstractions (e.g., ownership, garbage collection) or static analysis.

Secure Programming Languages: Key issues

• You expect a value and the value is exactly what you expect

Type Safety

Type safety ensures that variables are only used in ways consistent with their data types, preventing a class of errors that can lead to unpredictable behavior or security vulnerabilities.

Format String attacks

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buffer[100];
    if (argc > 1) {
        snprintf(buffer, sizeof(buffer), argv[1]);  // No format specifier!
        printf("You entered: %s\n", buffer);
    }
    return 0;
}
```

The format string is **user-controlled** (argv[1]).

Because snprintf() expects a format string, the user can inject **format specifiers** like %x, %s, or %n to:

Secure Programming Languages: Key issues

Type Safety

Type safety ensures that variables are only used in ways consistent with their data types, preventing a class of errors that can lead to unpredictable behavior or security vulnerabilities.

Solution: Strongly typed languages (e.g., Haskell, Ocaml, Rust) enforce strict type rules at compile time, reducing runtime type errors and related vulnerabilities.

.

Secure Programming Languages: Key issues

(Compositionality: glue everything together using types)

Encapsulation and Modularity

Good language design should support encapsulation to limit access to internal components of a module, thereby reducing the attack surface.

To guarantee isolation

Solution: Use access modifiers (e.g., private, public, protected) and module systems to isolate code and prevent unintended interactions.

Secure Programming Languages: Key issues

Concurrency Safety

Race conditions and other concurrency bugs can lead to unpredictable behavior and security flaws such as data leaks or deadlocks.

Much more powerful type system to work with threads

Solution: Languages like Rust use type systems to enforce safe concurrency, while others like Erlang use actor models for process isolation.

↑ thread isolation

Secure Programming Languages: Key issues

Input Validation and Taint Tracking

Handling untrusted input is a common vector for injection attacks (e.g., SQL injection, XSS).

Solution: Some secure languages support **taint analysis**, marking untrusted inputs and ensuring they are sanitized before use. Others embed DSLs for safe SQL queries (e.g., LINQ in C#).

↑ Domain specific language that performs sanitization

→ LINQ provides a bridge from prog. language to DB to avoid SQL injections

Secure Programming Languages: Key issues

Formal Verification and Static Analysis

There are languages designed to be powerful enough to prove properties about their behavior

Some languages are designed with features that facilitate formal reasoning or verification of program behavior.

Solution: Languages **F*** allow formal proofs about program correctness and security properties.

Secure Programming Languages: Key issues

Default Secure Behaviors

Secure languages tend to choose **secure defaults** over convenient but risky ones—such as immutability, explicit error handling, or safe standard libraries.

↳ supply chain problem: library usage is at your risk

Example: Java and Rust immutable objects.

Challenges

- **Performance vs. Safety**: Low-level languages offer more control but less safety. High-level, secure languages might impose overhead.
- **Ease of Use vs. Strictness**: More secure languages often come with steeper learning curves
- **Legacy Code and Interoperability**: Secure languages need to interoperate with existing, possibly insecure, ecosystems.

Rust

Industry-supported language designed to overcome this tension

try and solve challenges we have designed

Main idea: use a strong type system to prohibit important kinds of unwanted behavior, namely those involving mutation of shared state.

Approach allows many kinds of systems programming errors (e.g, data races, use-after-free) to be detected statically

For data structures that cannot be checked in this way, Rust allows them to encapsulated as “unsafe” within otherwise safe APIs

A strong type system deals with memory safety.

- For legacy code you have "unsafe" keyword

Languages, Memory and Compilers

In C and C++, we notice memory errors by reasoning and testing.

- The language and the compiler can't help us much in order to make it easier to reason about programs,

Rust places some restrictions on the programs you can write

- This can be annoying... but it also gives us some nice guarantees!

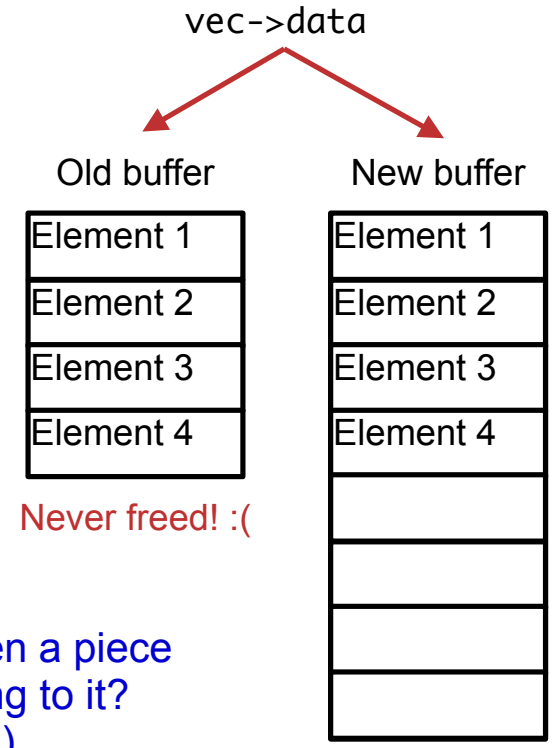
Memory leaks

```
void vec_push(Vec* vec, int n) {
  if (vec->length == vec->capacity) {
    int new_capacity = vec->capacity * 2;
    int* new_data = (int*) malloc(new_capacity);
    assert(new_data != NULL);

    for (int i = 0; i < vec->length; ++i) {
      new_data[i] = vec->data[i];
    }

    vec->data = new_data; // OOP: we forget to free the old data
    vec->capacity = new_capacity;
  }

  vec->data[vec->length] = n;
  ++vec->length;
}
```



Wouldn't it be nice if the compiler noticed when a piece of heap data no longer had anything pointing to it? (and so then it could safely be freed?)

A C++ code

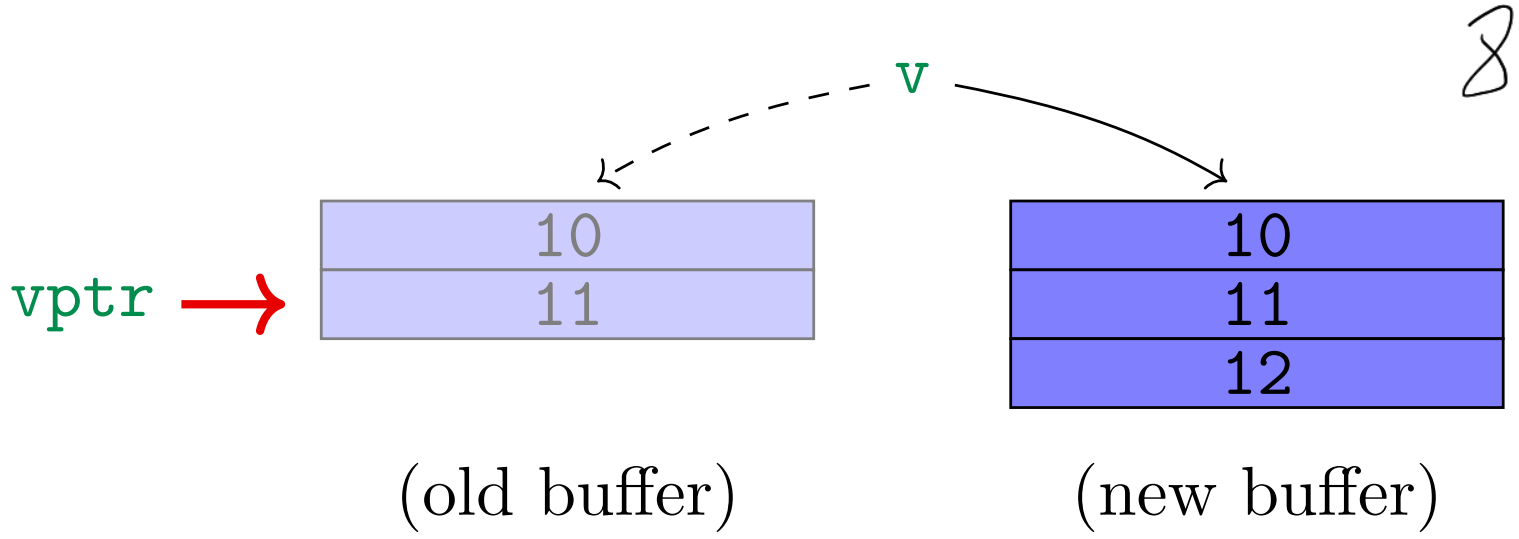
```
1 std::vector<int> v { 10, 11 };
2 int *vptr = &v[1]; // Points *into* ‘v‘.
3 v.push_back(12);
4 std::cout << *vptr;
```

A C++ code

```
1 std::vector<int> v { 10, 11 }; //a resizable array of integers
2 int *vptr = &v[1]; // Points *into* 'v'. A partial overlap
3 v.push_back(12); // Here no more space for an additional element
                   // a new buffer is allocated
                   // all the existing elements are moved over.
4 std::cout << *vptr;
```

Why is this case interesting?

Memory usage: an interesating case



both pointers were aliasing: an action through a pointer (v) will in general also affect all its aliases (vptr)

vptr becomes a dangling reference!!!

Rust

Started in 2006 by Graydon Hoare

Sponsored by Mozilla in 2010

Now: most loved programming language in Stack Overflow

Key properties:Type safety despite use of concurrency and manual memory management

Main Features

- Ownership, Borrowing and Lifetime: Key notions for ensuring memory safety
- Traits as core of object(-like) system
- Variable default is immutability
- Data types and pattern matching
- Type inference
- Generics (parametric polymorphism)
- First-class functions

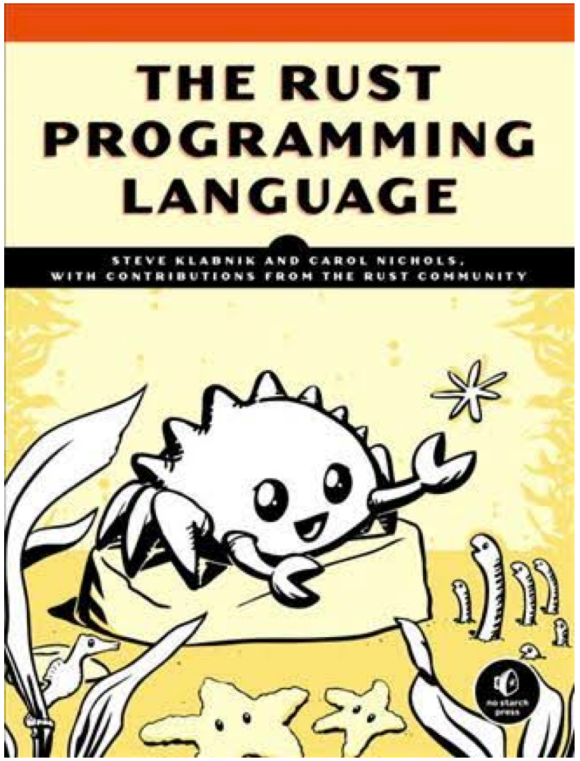
Rust: Design goals aka Shift left!!

Ensure that Rust programs never have undefined behavior.

- We know that about 70% of reported security vulnerabilities in low-level systems are caused by memory corruption, which is one form of undefined behavior.

Avoid undefined behavior at compile-time instead of run-time.

- Catching bugs at compile-time means improving the reliability of software.
- Catching bugs at compile-time means fewer runtime checks for those bugs, improving the performance of software.



- Rust book free online
  - https://doc.rust-lang.org/book/
  - **We will follow it in these lectures**
- More references via Rust site
  - https://www.rust-lang.org/en-US/documentation.html
- Rust Playground (REPL)
  - https://play.rust-lang.org/

RUST BASICS

- Variables are bound with `let`:

```
let x = 17;
```

- Bindings are implicitly-typed: the compiler infers based on context.
- The compiler can't always determine the type of a variable, so sometimes you have to add type annotations.

```
let x: i16 = 17;
```

- Variables are inherently immutable:

```
let x = 5;
x += 1; // error: re-assignment of immutable variable x
let mut y = 5;
y += 1; // OK!
```

- (Almost!) everything is an expression: something which returns a value.
  - Exception: variable bindings are not expressions.

- Primitive types
  - bool, char, numerics (i8, i16, …, u8, u16, …)
- Arrays
    - Arrays are generically of type [T; N].
    - N is a compile-time *constant*. Arrays cannot be resized.
    - Array access is bounds-checked at runtime.
    - Arrays are indexed with [] as in C

- Strings
- Tuples (fixed-sized heterogenous lists)
- Structs, unions, …
- Unit type (contains a single value - ())
- No void type

FUNCTIONS

```
// comment
fn main() {
    println!("Hello, world!");
}
```

Hello, world!

Recursive functions

```
fn fact(n:i32) -> i32
{
  if n == 0 { 1 }
  else {
    let x = fact(n-1);
    n * x
  }
}

    fn main() {
      let res = fact(6);
      println!("fact(6) = {}",res);
    }
```

```
fact(6) = 720
```

OCAML VS RUST

```
let f (m : int) (n : int) : int =
  let mm = 2 * m in
  mm + n    ;;

let main () =
  Printf.printf "%d" (f 3 5)  ;;
```

```
fn f(m: i32, n: i32) -> i32 {
  let mm = 2 * m;
  mm + n
}

fn main() {
  println!("{}", f(3, 5));
}
```

RUST is is expression-oriented.
The semicolon in **let x = y; z** has basically the role of the **in** keyword in Ocaml:
it defines an expression, not a statement as in C

IF expressions (not statements)

```
fn main() {
    let n = 5;
    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }
}
```

5 is positive

LET

- By default, Rust variables are immutable
  - Usage checked by the compiler
- **`mut`** is used to declare a resource as mutable.

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

Compile error

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

LET BY EXAMPLES

```
{
  let x = 37;
  let y = x + 5;
  y
}//42
```

```
{
  let x = 37;
  x = x + 5;//err
  x
}
```

```
{ //err:
  let x:u32 = -1;
  let y = x + 5;
  y
}
```

```
{
  let x = 37;
  let x = x + 5;
  x
}//42
```

Redefining a variable *shadows* it (like OCaml)

```
{
  let mut x = 37;
  x = x + 5;
  x
}//42
```

Assigning to a variable only allowed if **`mut`**

```
{
  let x:i16 = -1;
  let y:i16 =
x+5;
  y
}//4
```

Type annotations must be consistent (may override defaults)

QUIZ: What does this evaluate to?

```
{ let x = 6;
  let y = "hi";
  if x == 5 { y } else { 5 };
  7
}
```

A. 6

B. 7

C. 5

D. Error

QUIZ #1: What does this evaluate to?

```
{ let x = 6;
  let y = "hi";
  if x == 5 { y } else { 5 };
  7
}
```

A. 6

B. 7

C. 5

D. Error – if and else have incompatible types

Mutation and Loop

```
fn fact(n: u32) -> u32 {
  let mut x = n;
  let mut a = 1;
  loop {
    if x <= 1 { break; }
    a = a * x;
    x = x - 1;
  }
  a
}
```

infinite loop
(break out)

Looping constructs

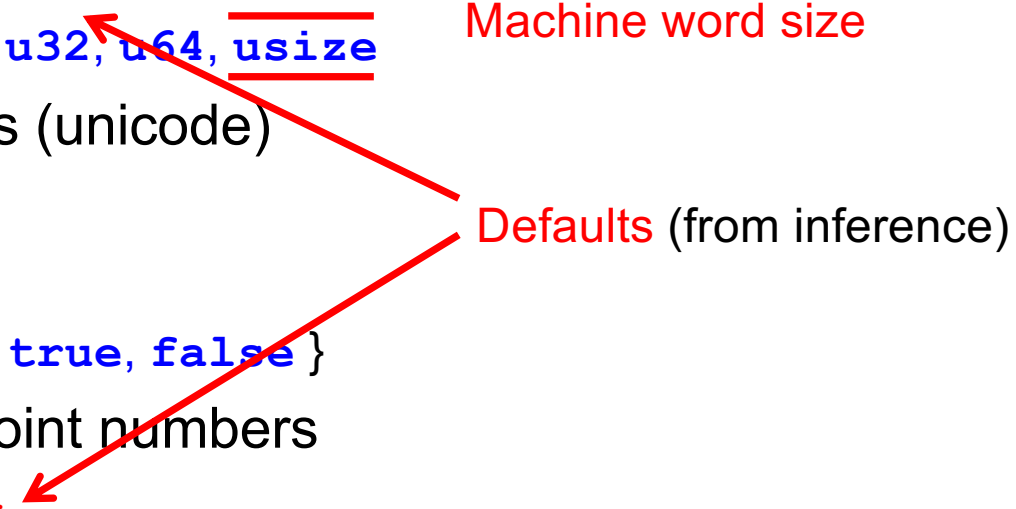
- **while ….**, **for** …, **loop** …
- These looping constructs are ***expressions***
  - They return the final computed value
  - unit, if none

`break` may take an expression argument, which is the final result of the loop

```
let mut x = 5;
let y = loop {
      x += x - 3;
      println!("{}", x);// 7 11 19 35
      x % 5 == 0 { break x; }
};
print!("{}",y); //35
```

Scalar type

- Integers
  - `i8`, `i16`, `i32`, `i64`, `isize`
  - `u8`, `u16`, `u32`, `u64`, `usize`
- Characters (unicode)
  - `char`
- Booleans
  - `bool` = { `true`, `false` }
- Floating point numbers
  - `f32`, `f64`
- Note: arithmetic operators (+, -, etc.) *overloaded*



tuples

Tuples

- n-tuple type `(t1,…,tn)`
  - `unit ()` is just the 0-tuple
- n-tuple expression `(e1,…,en)`
- Accessed by pattern matching or like a record field

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;

    println!("The value of five_hundred is: {}", five_hundred);

    println!("The value of six_point_four is: {}", six_point_four);

   println!("The value of one: {}", one);

}
```

RUN ▶ ··· DEBUG STABLE ···

```
    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;

    println!("The value of five_hundred is: {}", five_hundred);

    println!("The value of six_point_four is: {}", six_point_four);

   println!("The value of one: {}", one);

}

```

Execution

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.57s
     Running `target/debug/playground`
```

Standard Output

```
The value of five_hundred is: 500
The value of six_point_four is: 6.4
The value of one: 1
```

Arrays

Standard operations

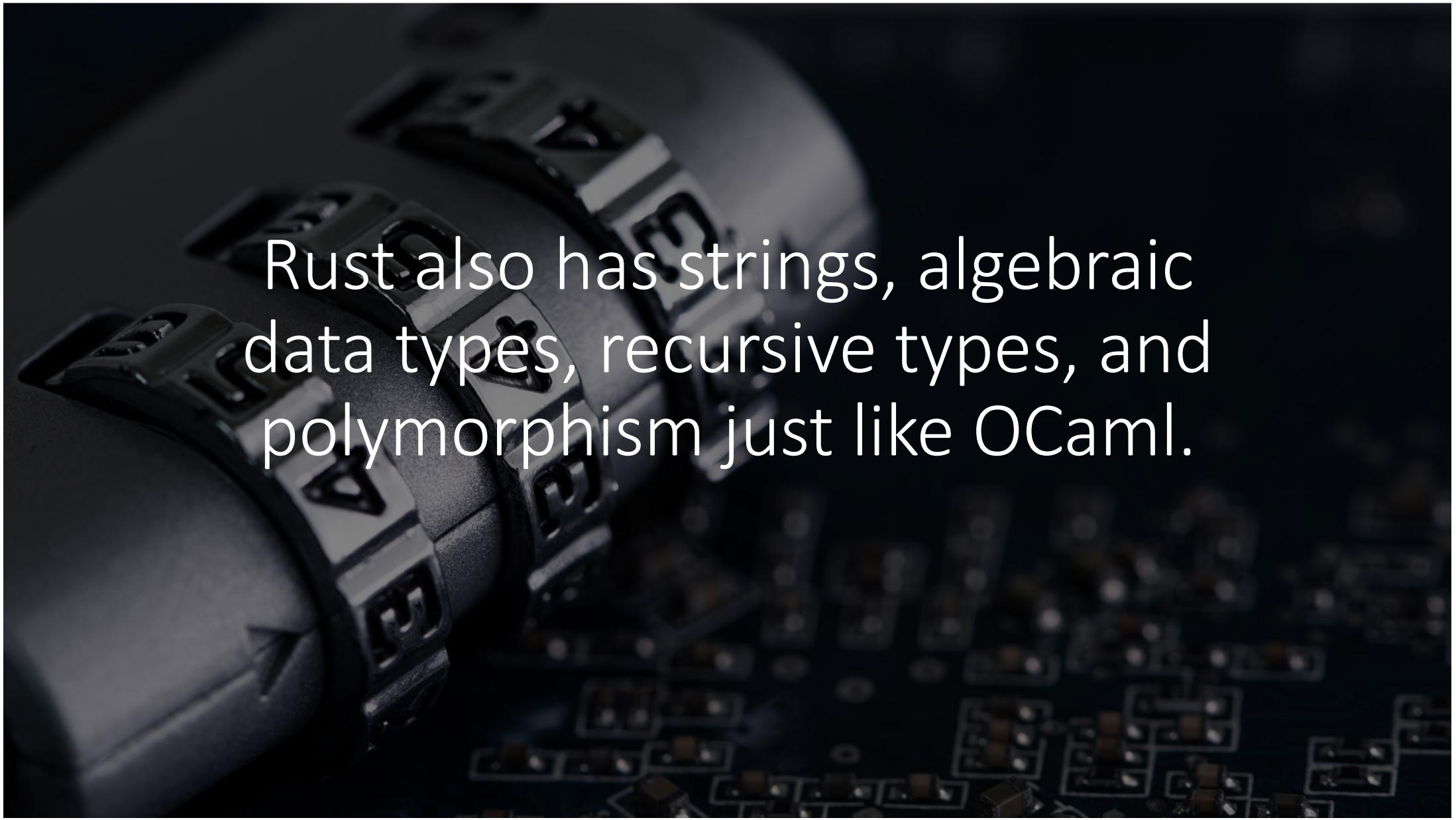
- Creating an array (can be mutable or not)
  - But must be of fixed length
- Indexing an array
- Assigning at an array index

```
let nums = [1,2,3];
let strs = ["Monday","Tuesday","Wednesday"];
let x = nums[0]; // 1
let s = strs[1]; // "Tuesday"
let mut xs = [1,2,3];
xs[0] = 1; // OK, since xs mutable
let i = 4;
let y = nums[i]; //fails (panics) at run-time
```

- Rust provides a way to iterate over a collection
  - Including arrays

```
let a = [10, 20, 30, 40, 50];
for element in a.iter() {
  println!("the value is: {}", element);
}
```

  - `a.iter()` produces an iterator, like a Java iterator


Rust also has strings, algebraic data types, recursive types, and polymorphism just like OCaml.

```
// Polymorphic record for points. Equivalent to type 'a point = {x : 'a; y : 'a}.

struct Point<T> {
  x: T, y: T
}

// Polymorphic function. T: ToString means "all types T where T has a to_string
// method." ToString is a trait, which we will discuss later.

fn point_print<T: ToString>(p: Point<T>) {
  println!("({}, {})", p.x.to_string(), p.y.to_string());
}
```

```
// Polymorphic recursive sum type for lists. Note that we have to wrap the recursive reference in a Box<...>.
// This ensures the type has a known size in memory. We will discuss Box in a few minutes!!
enum List<T> {
  Nil,
  Cons(T, Box<List<T>>)
}

fn main() {
  let p1: Point<i32> = Point { x: 1, y: 2 };
  let p2: Point<f32> = Point { x: 3.1, y: 4.2 };
  point_print(p1); // (1, 2)
  point_print(p2); // (3.1, 4.2)
  // Rust has type inference, so it can infer the type of `l` as List<i32>.
  let l = List::Cons(5, Box::new(List::Nil));
  // Rust has match statements just like OCaml.
  let n = match l {
    List::Cons(n, _) => n,
    List::Nil => -1
  };
  println!("{}", n); // 5
}
```

RUN ▶ ··· DEBUG STABLE ···

```
21    let p1: Point<i32> = Point { x: 1, y: 2 };
22    let p2: Point<f32> = Point { x: 3.1, y: 4.2 };
23    point_print(p1); // (1, 2)
24    point_print(p2); // (3.1, 4.2)
25
26    // Rust has type inference, so it can infer the type of `l` as List<i32>.
27    let l = List::Cons(5, Box::new(List::Nil));
28    // Rust has match statements just like OCaml.
29    let n = match l {
30      List::Cons(n, _) => n,
31      List::Nil => -1
32    };
33    println!("{}", n); // 5
34  }
```

Execution

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 2.77s
     Running `target/debug/playground`
```

Standard Output

```
(1, 2)
(3.1, 4.2)
5
```

Data allocation in memory: Stack vs Heap

The stack

- constant-time, automatic (de)allocation
- Data size and lifetime must be known at compile-time
  - Function parameters and locals of known (constant) size

The heap

- Dynamically sized data, with non-fixed lifetime
- Slightly slower to access than stack via a reference

References

Reference *types* are written with an `&`: `&i32`.

References can be taken with `&` (like C/C++).

References can be *dereferenced* with `*` (like C/C++).

References are guaranteed to be valid.

- Validity is enforced through compile-time checks!

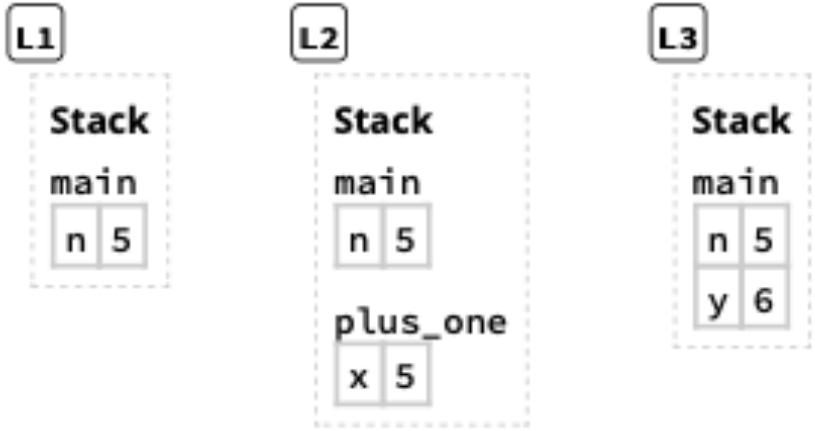
These are *not* the same as pointers!

```
let x = 12;
let ref_x = &x;
println!("{}", *ref_x); // 12
```

Stack allocation

```rust
fn main() {
    let n = 5; L1
    let y = plus_one(n); L3
    println!("The value of y is: {y}");
}

fn plus_one(x: i32) -> i32 {
    L2 x + 1
}
```



Stack allocation

```
let a = [0; 1_000_000]; L1
let b = a; L2
```

L1

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0

L2

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0
b	0	0	0	0	0	0	0	0	0	0	...	0

Stack allocation

```
let a = [0; 1_000_000]; L1
let b = a; L2
```

L1

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0

L2

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0
b	0	0	0	0	0	0	0	0	0	0	...	0

Copying a into b causes the main AR to contain 2 million elements.!!!!!

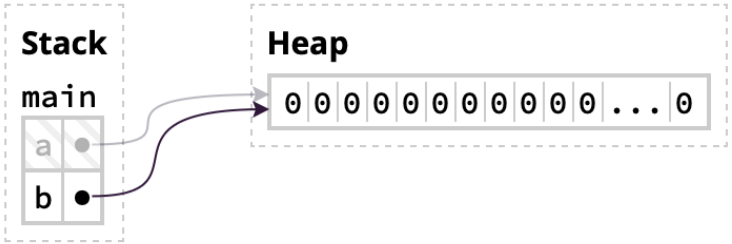
Boxes lives in the Heap

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```

L1

The statement `let b = a` copies the pointer from `a` into `b`, but the pointed-to data is not copied.

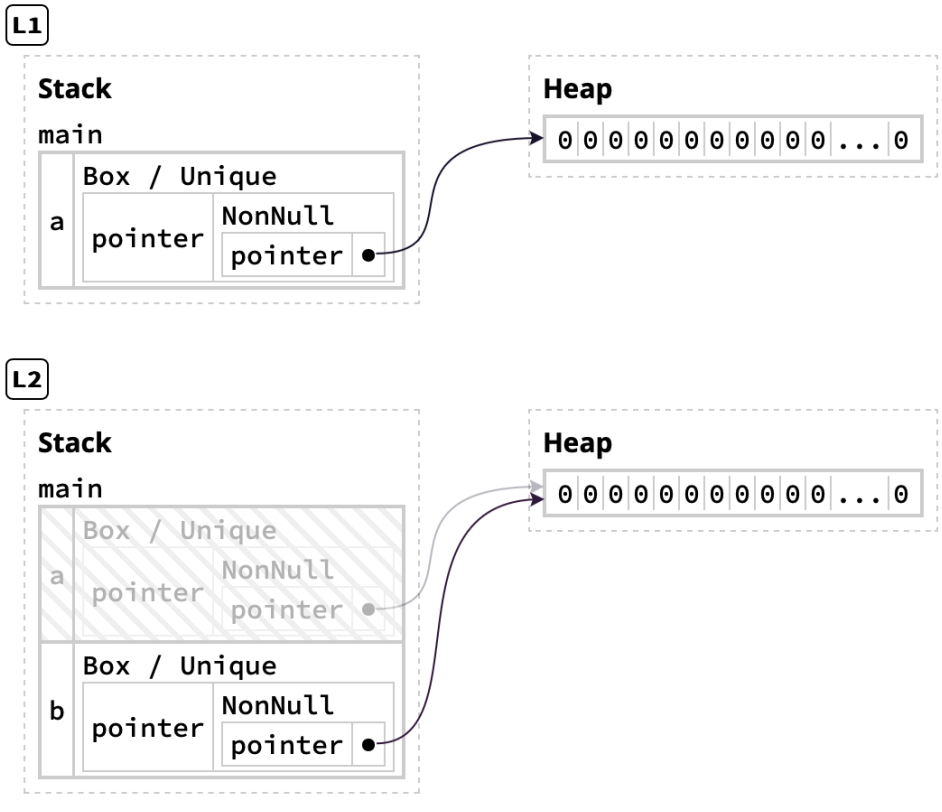
L2



Note that `a` is now grayed out because it has been *moved*.

Boxes lives in the Heap

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```



RUST: safe memory management

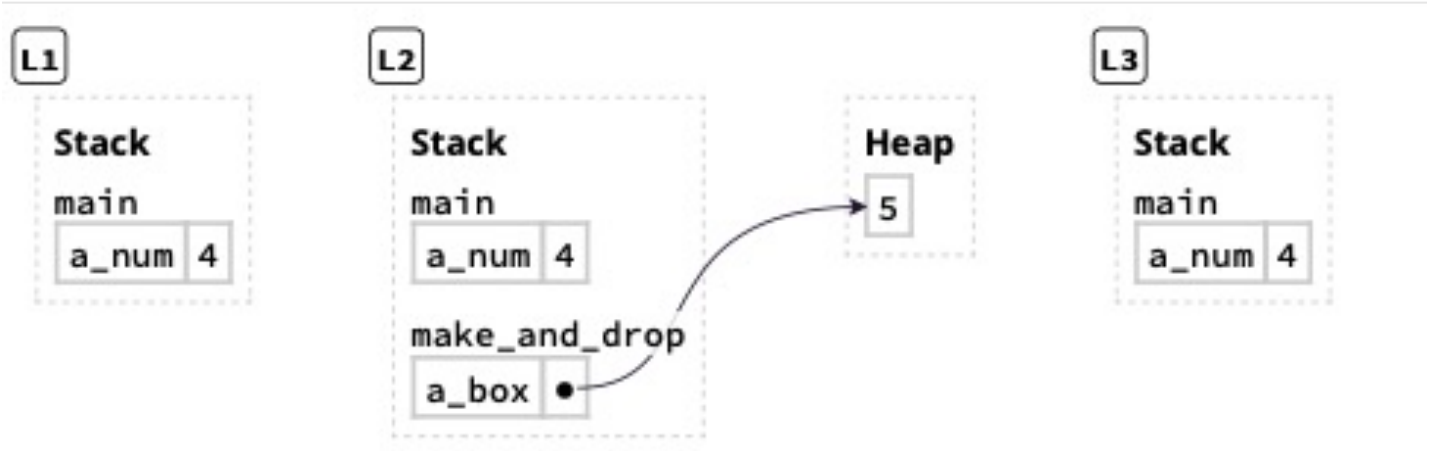
- Rust heap memory management: **without GC**
  - Type checking ensures **no dangling pointers**
  - unsafe idioms are disallowed

Box deallocation principle (almost correct):

If a variable is bound to a box, when Rust deallocates the variable's frame, then Rust deallocates the box's heap memory.

```
fn main() {
    let a_num = 4;
     L1 make_and_drop(); L3
}

fn make_and_drop() {
    let a_box = Box::new(5); L2
}
```



Boxes data work well!! …. What if

```
let a = Box::new([0; 1_000_000]);
let b = a;
```

The boxed array has now been bound to both a and b (alias).

By applying the "almost correct" principle, Rust would try to free the box's heap memory *twice* on behalf of both variables.

That's undefined behavior too!