CS 420 Advanced Programming Languages
Fall Semester, 2022
Doc 3 Rust
Aug 30, 2022

Rust

Version 1.0 Released May 15, 2015

Multi-paradigm, general-purpose programming language

Focus

- Performance
- Type Safety
- Concurrency

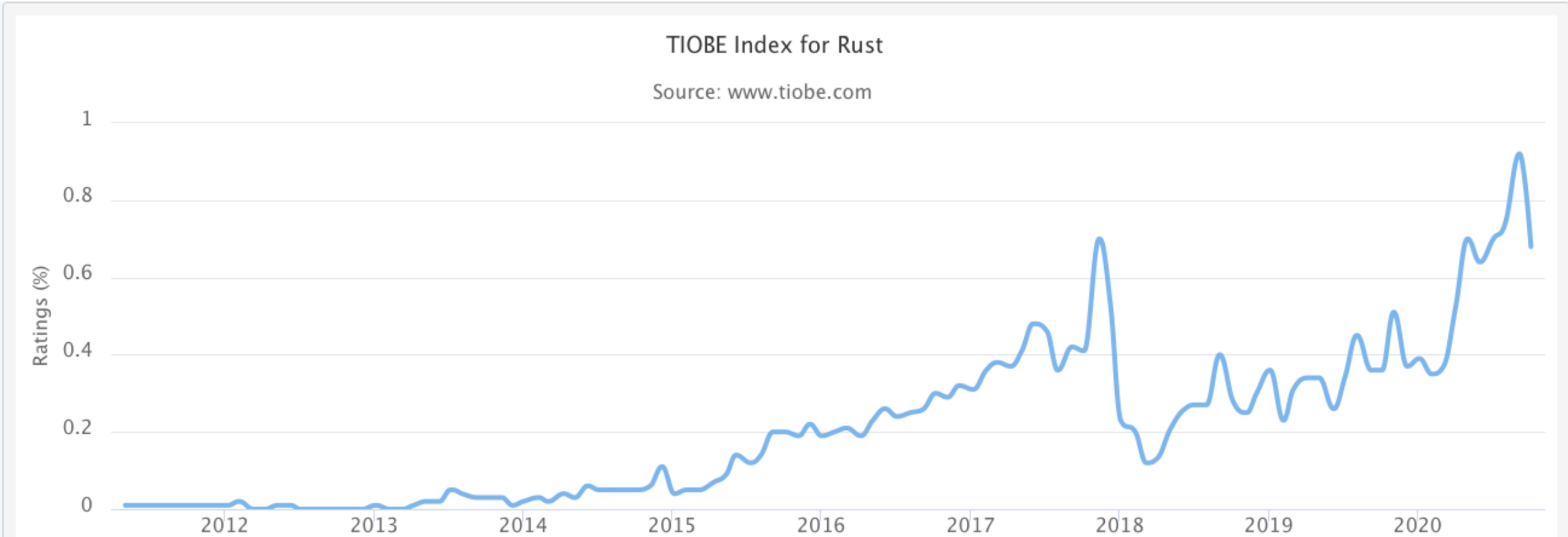
Enforces memory safety

- without a garbage collector or reference counting

Macros & Traits

Stack Overflow Developer Survey

- Most loved programming language every year from 2016 to 2022


TIOBE Index for Rust
Source: www.tiobe.com
Ratings (%)
1
0.8
0.6
0.4
0.2
0
2012
2013
2014
2015
2016
2017
2018
2019
2020

Rust Install & Tools

https://rustup.rs

Follow instructions

Cargo

Compilation manager

Package Manager

rustc

Compiler

rustdoc

Documentation Tool

Clippy

Lint tool

```
 fn main() {
     println!("Hello World");
 }


fn multiply(a: i64, b: i64) -> f64 {
    assert_ne!(b, 0);
    return a * b;
}
```

Basic Types

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Number literals	Example
Decimal	98_222
Hex	FF
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

```
let x = 2.0; // f64
let y: f32 = 3.0; // f32
```

```
let t = true;
let f: bool = false;
```

```
let c = 'z';
let z: char = 'ℤ';
let heart_eyed_cat = '😻';
```

```
let tup = (500, 6.4, 1);
let (x, y, z) = tup;
```

Variables - Immutable Default

```
fn main() {
    let x = 5;
    x = 6;          // Error X is immutable
}
```

```
fn main() {
    let  mut x = 5;
    x = 6;          // ok
}
```

Sample Program

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

read_line returns Result

- Ok
- Err

Without expect

```
fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess);
    println!("You guessed: {guess}");
}
```

```
    |
12  | /     io::stdin()
13  | |         .read_line(&mut guess);
    | |_____________________________________^
    |
    = note: `#[warn(unused_must_use)]` on by default
    = note: this `Result` may be an `Err` variant, which should be ha
```

Rust is Picky about Types

```
fn divide(a: i64, b: i64) -> f64 {
    assert_ne!(b, 0);
    return a / b;
}
```

```
error[E0308]: mismatched types
 --> src/main.rs:9:12
  |
7 | fn divide(a: i64, b: i64) -> f64 {
  |                              --- expected `f64` because of return
8 |     assert_ne!(b, 0);
9 |     return a / b;
  |            ^^^^^ expected `f64`, found `i64`
  |
help: you can convert an `i64` to an `f64`, producing the floating poi
representation of the integer, rounded if necessary
  |
9 |     return (a / b) as f64;
  |            +     ++++++++
```

Rust is Picky about Types

```
fn divide(a: i64, b: i64) -> f64 {
    assert_ne!(b, 0);
    return (a / b) as f64;
}


#[test]
fn test_divide() {
    assert_eq!(0.5, divide(1, 2))
}
```

```
thread 'test_divide' panicked at 'assertion failed: `(left == right)`
  left: `0.5`,
 right: `0.0`', src/main.rs:14:5
```

Rust is Really Picky about Types

```
fn divide(a: i64, b: i64) -> f64 {
    assert_ne!(b, 0);
    return (a as f64/ b) ;
}
```

```
  |
9 |     return (a as f64/ b) ;
  |                    ^ no implementation for `f64 / i64`
  |
  = help: the trait `Div<i64>` is not implemented for `f64`
```

This works

```
fn divide(a: i64, b: i64) -> f64 {
    assert_ne!(b, 0);
    return (a as f64/ b as f64) ;
}

#[test]
fn test_divide() {
    assert_eq!(0.5, divide(1, 2))
}
```

Tuples

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

let five_hundred = x.0;
let six_point_four = x.1;
let one = x.2;

let (a, b, c) = x;
```

```
let mut  x: (i32, f64, u8) = (500, 6.4, 1);

x.1 = 4.0;
x = (10, 2.2, 9);

println!("{}", x.1);

x = (11.2, 2, 9);                  //Error
x = (1,1.2,1);                     // Error
x.1 = 5.9;                         // Error
```

Arrays

```
let mut a = [1, 2, 3, 4, 5];
a[0] = 4;

println!("{}", a[0]);
```

Arrays are allowed on stack

Size is static

```
#![allow(unused)]
fn main() {
    let a = [3; 5];
}
```

a == [3, 3, 3, 3, 3]

```
fn foo( n: i32) {
    let a = [5;n];       // Error n needs to be a `const`
    println!("{}", a[0]);
}
```

Test for Parameter Passing

```
fn main() {
    let mut a = [1, 2, 3, 4, 5];
    foo(a);

    println!("{}", a[0]);
}

fn foo(mut param: [i32;5]) {
    param[0] = 50;
}
```

Flow Control

if

loop

while

for

```
if number < 5 {
    println!("condition was true");
} else {
    println!("condition was false");
}
```

```
let mut number = 3;

while number != 0 {
    println!("{number}!");

    number -= 1;
}
```

```
let mut counter = 0;

loop {
    counter += 1;
    if counter == 10 {
        break counter * 2;
    }
};
```

```
let a = [10, 20, 30, 40, 50];

for element in a {
    println!("the value is: {element}");
}
```

Flow Control are Expressions

if, loop, while, for are expressions

They return a value

```
let number = if condition { 5 } else { 6 };
```

```
let number;
if condition {
    number = 5;
}
else {
    number = 6;
};
```

Collections

Sequences: Vec, VecDeque, LinkedList

Maps: HashMap, BTreeMap

Sets: HashSet, BTreeSet

Misc: BinaryHeap

Unlike Tuple and Array
These can grow at runtime

Vec

Creating a Vec

```
let mut a = vec![1,5,2]
let b = vec![3;10] //10 elements - all 3

let mut c = Vec::new();
```

Accessing elements

```
for n in 1..10 {
   c.push(n);
}

a[1] = 10;
let d = a[3];
let e : Option<&i32> =  a.get(3)
```

& denotes a reference

```
let a = &3;
let b = &2;
let c = a + b;
assert_eq!(c, 5);
```

More on & in a bit

Illegal Access

```
let a = vec![1,2,3];

a[10];

a.get(20);
```

a[10];

causes program to panic (crash)

a.get(20);

Returns an Option

Option

Contains either

- Some (with the value)
- None

```
fn main() {
    let sam = vec![3;10];
    let fifth : Option<&i32> = sam.get(4);
    let mut ref_or_panic = fifth.unwrap();
    assert_eq!(ref_or_panic, &3);

    ref_or_panic = fifth.expect("fifth element is not 3");
    assert_eq!(ref_or_panic, &3);

    let mut ref_or_zero = fifth.unwrap_or(&0);
    ref_or_zero = match fifth {
        Some(fifth) => fifth,
        None => &0,
    };
    assert_eq!(ref_or_zero,&3);

    let mappped : Option<i32> = fifth.map(|fifth| fifth + 1);
    assert_eq!(mappped, Some(4));
}
```

Optionals Handling Errors

In C

```
int result = foo(x);
if (result =! 0)
    exit(1);
```

Java

```
try {
    foo(x);
} catch (Exception e) {
   do something
}
```

Rust

```
let result = foo(x)
match result {
    Some(value) => handle normal case
    None => handle exception
};
```

Back to Vec

```
fn main() {
    let a : Vec<i32> = vec![1, 2, 3, 4, 5];
    let b = a[0];
    println!("{}",b);
    for x in a {
        println!("{x}");
    }

    let mut c = Vec::new();
    for n in 1..10 {
        c.push(n);
    }

    let fifth : Option<&i32> = c.get(4);
    match fifth {
        Some(fifth) => println!("The fifth element is {}", fifth),
        None => println!("There is no fifth element."),
    }
    print!("{:?}", fifth);
}
```

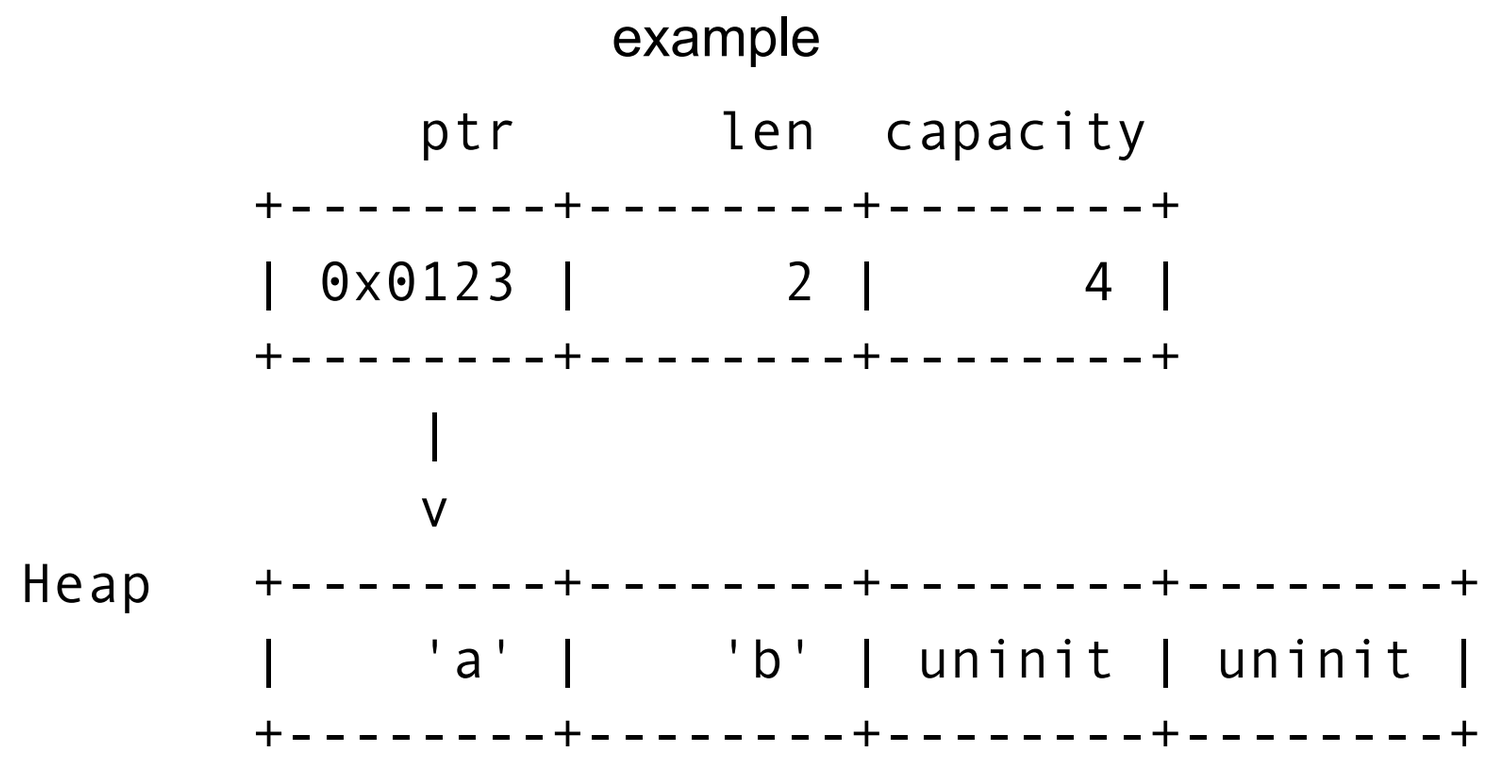
Vec, Heap, Growing

A Vec has:

- Pointer - to actual values
- Length - number of values
- Capacity - How many values it can hold

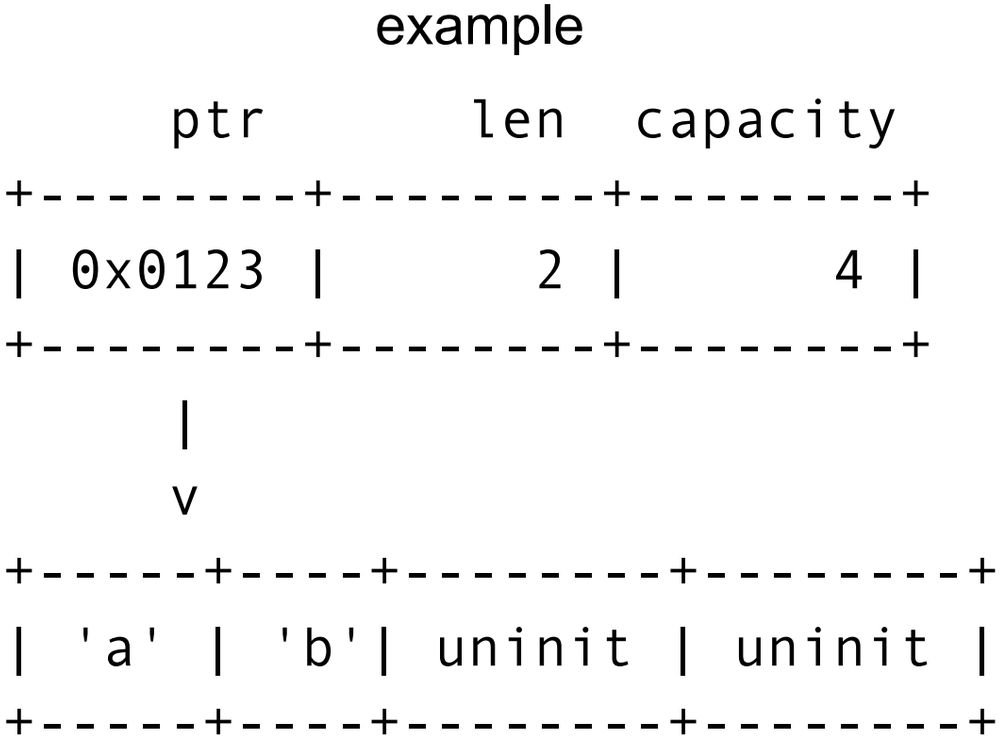
```
fn main() {
    let mut example = Vec::with_capacity(4);
    example.push('a');
    example.push('b');
    assert!(example.len() == 2);
    assert!(example.capacity() == 4);
}
```

Elements in Vec are always on the heap



Vec, Heap, Growing

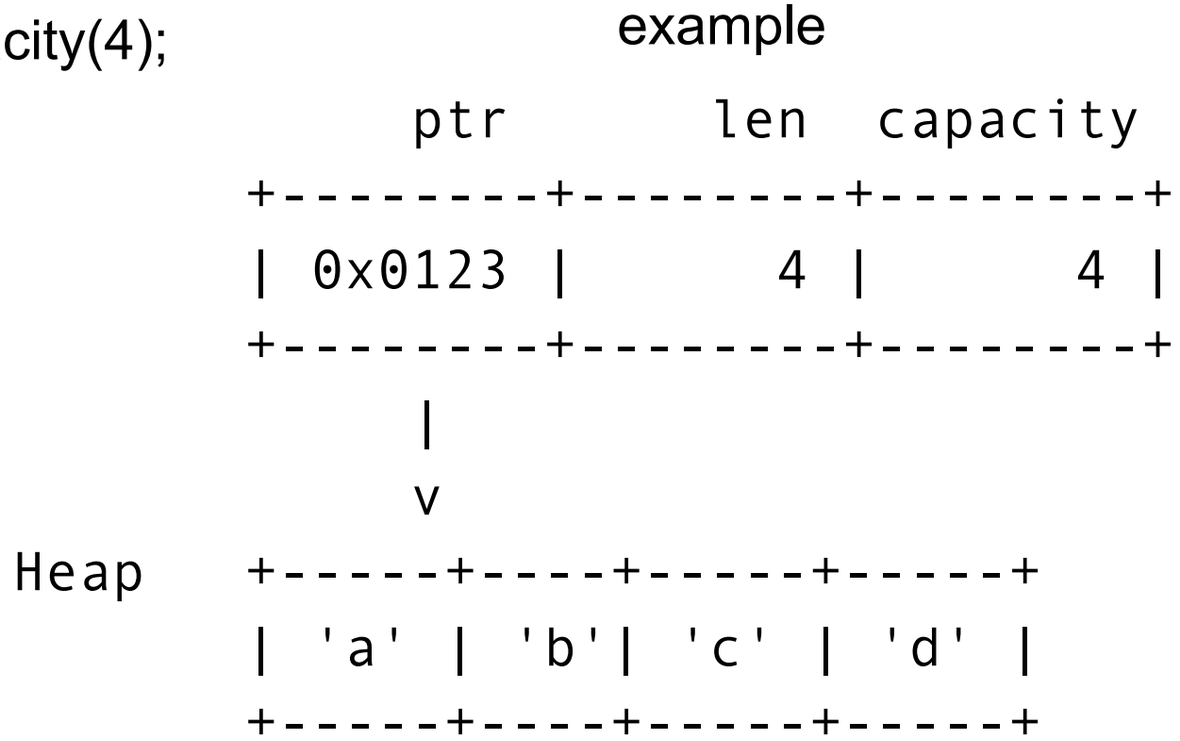
```
fn main() {
    let mut example = Vec::with_capacity(4);
    example.push('a');
    example.push('b');
```



Vec, Heap, Growing

```
fn main() {
    let mut example = Vec::with_capacity(4);
    example.push('a');
    example.push('b');

     example.push('c');
     example.push('d');
```

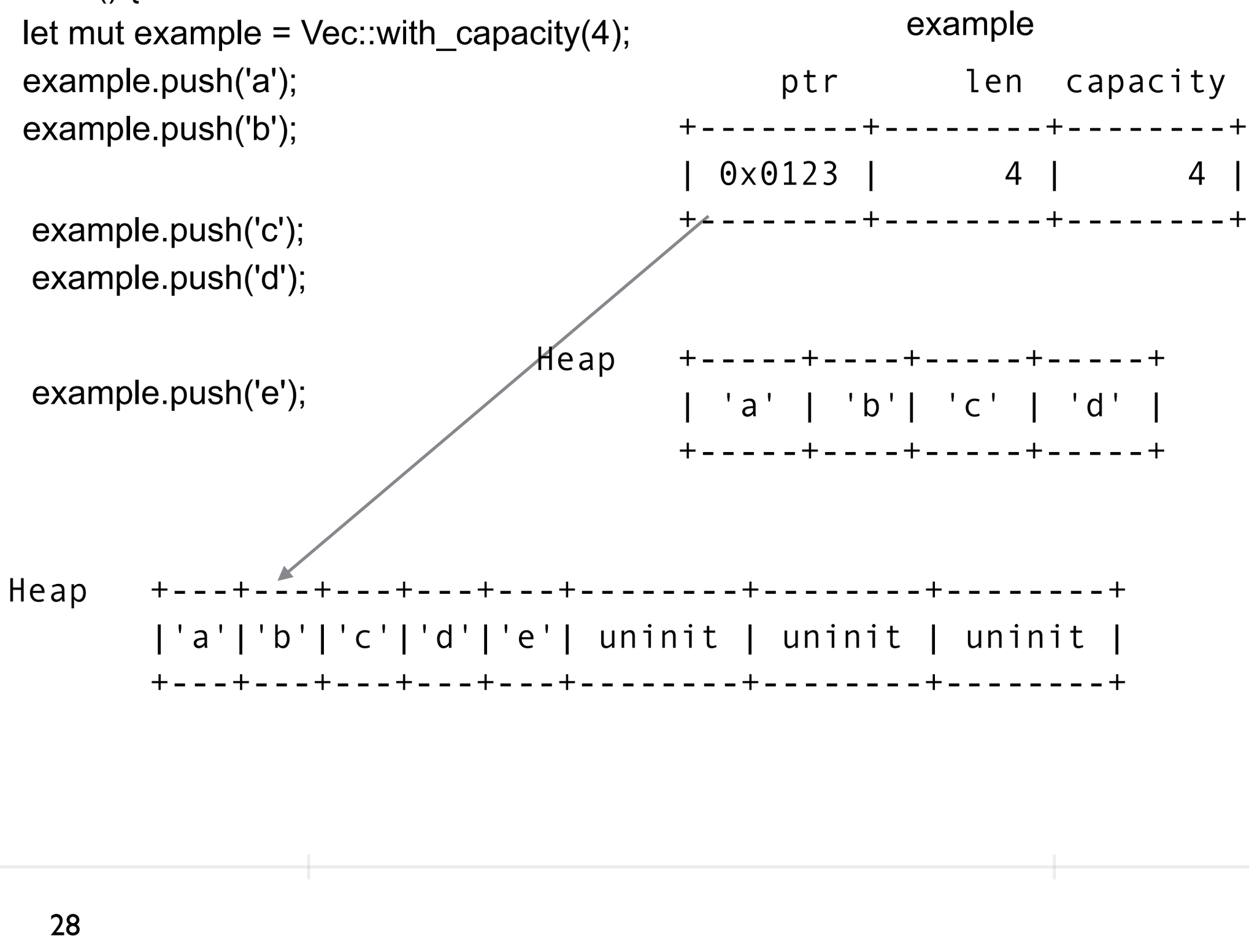


Vec, Heap, Growing

```
fn main() {
    let mut example = Vec::with_capacity(4);
    example.push('a');
    example.push('b');

     example.push('c');
     example.push('d');

    example.push('e');
}
```



What is Printed out?

```
fn main() {
    let  a = vec![1,2,3];
    let  b = a;
    let first = a[0];
    println!("{}", first);
}
```

```
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:9:17
  |
6 |     let  a = vec![1,2,3];
  |          - move occurs because `a` has type `Vec<i32>`, which does not implement the `Copy` trait
7 |     let  b;
8 |     b = a;
  |         - value moved here
9 |     let first = a[0];
  |                 ^ value borrowed here after move
```

What is Printed out?

```
fn main() {
    let mut  a = [1,2, 3];
    let   b = a;
    a[0] = 5;
    print!("{}", b[0]);
}
```

Copy values
- base types - int, float, tuple
- Most stack-based values

Copied when
- assigned
- passed as parameter
- Returned from a function

Non-copy values
- Heap-base values
- Things containing non-copy values
- Things that dynamically grow
- Large things - Vec, String

A value becomes a Copy value by
- Implementing Copy

String & str

str

immutable
Fixed length
UT8 Unicode
Copy

```
let a: str = "Hello, world!";
println!("{}", a);

let hello_in_hindi = "नमस्ते";
let ஹலோ = "hello in Tamil";
```

String

Growable
So owns values on the heap
UT8 Unicode
Not Copy

```
let mut hello = String::from("Hello, ");
hello.push('w');
hello.push_str("orld!");
```

```
let uni_mouse = "How to embed unicode in string \u{1F42D}";
println!("{}",uni_mouse);
```

How to embed unicode in string 🐭

Unicode Makes Strings Complex

```
let foo = "Hi Mom";

foo[1];                                 // Compile Error



let mut a= String::from("\u{65}");
assert_eq!(a.len(), 1);

a.push('\u{301}');
assert_eq!(a.len(), 3);                 // In Swift we would get length 1
```

Ownership

The Rules

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

```
fn main() {
    let  a = 1;
    let  b = vec![1,2,3];
    {
        let  f = vec!['a','b','c'];
    }
    let  c = a;
    let  d = b;     // d now owns the vector, b can not be used to access it
}
```

Moves

```
fn main() {
   let  b = vec![1,2,3];
   let  d = b;     // d now owns the vector, b can not be used to access it
}
```

When ownership of a value changes, Rust says the value moved.

Moves - Examples

```
let  a = String::from("Hello, world!");
let  b = a;

let  c = "Hello, world!";
let  d = c; // str is copy

println!("{}", a);  // Compile Error
println!("{}", c);  // No problem
```

```
let mut example = String::from("Hello, world!");
foo(example);
println!("{}", example);  // Compile Error

fn foo(a:String) {
    a.push_str("foo");
}
```

Moves - Examples

```
let x = vec![10, 20, 30];
if c {
    f(x); // ... ok to move from x here
} else {
    g(x); // ... and ok to also move from x here
}
h(x); // bad: x is uninitialized here if either path uses it
```

Moves - Examples

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// Pull out random elements from the vector.
let third = v[2]; // error: Cannot move out of index of Vec
let fifth = v[4]; // here too
```

Moves - Examples

```
let x = vec![10, 20, 30];
while f() {
    g(x); // bad: x would be moved in first iteration,
          // uninitialized in second
}
```

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);           // move from x
    x = h();        // give x a fresh value
}
e(x);
```

Moves - Examples

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };
let name = knuth.name.unwrap();

println!("{:?}", knuth.name);              // Compile Error
```

Moves - Examples

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };

let mut cs_gods = Vec::new();
cs_gods.push(knuth);

let name = cs_gods[0].name;  //Compile Error
```

This Works - Why?

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };
let name = match knuth.name {
    Some(ref n) => n,
    None => "",
};
println!("{:?}", knuth.name);
```

Borrowing a Value

```
let a = String::from("Hello");
let b = a;
println!("{}", a); // Compile Error
```

```
let a = String::from("Hello");
let b = &a;
println!("{}", a);  //OK
```

Two Types of References

Shared Reference

Read only

A value can have many shared references

&a returns a shared reference

Copy

Mutable reference

Read & Write

Only one reference of any type can be active if the value has a mutable reference

&mut a returns a mutable reference

Not Copy

Borrowing - Examples

```
let mut example = String::from("Hello, world!");
foo(example);
println!("{}", example);  // Compile Error

fn foo(a:String) {
    a.push_str("foo");
}
```

```
let mut example = String::from("Hello, world!");
foo(&mut example);
println!("{}", example);  // OK

fn foo(a:&mut String) {
    a.push_str("foo");
}
```

Borrowing - Examples

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };
let name = knuth.name.unwrap();

println!("{:?}", knuth.name);              // Compile Error
```

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };
let name : &Option<String> = (&knuth.name);

println!("{:?}", knuth.name);
```

Getting Ride of the Option

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };
let name : &Option<String> = (&knuth.name);

println!("{:?}", knuth.name);
```

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };
let name : &String = knuth.name.as_ref().unwrap();

println!("{:?}", knuth.name);
```

Borrowing - Examples

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };

let mut cs_gods = Vec::new();
cs_gods.push(knuth);

let name = cs_gods[0].name;  //Compile Error
```

```
struct Person { name: Option<String>, birth: i32 }

let knuth = Person { name: Some("Knuth".to_string()), birth: 1938 };

let mut cs_gods = Vec::new();
cs_gods.push(knuth);

let name = &cs_gods[0].name;
println!("{:?}", name)
```

Borrowing - Examples

```
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

let third = v[2]; // error: Cannot move out of index of Vec
```

```
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

let third = &v[2];
```