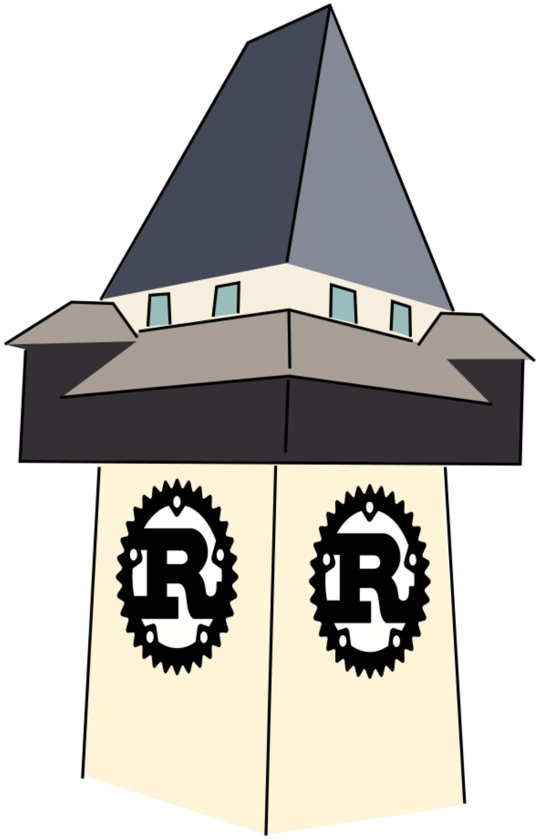
RUST GRAZ – 05 DATA STRUCTURES

Lukas Prokop

27th of November, 2019



Data structures

27th of November 2019

Rust Graz, lab10

PROLOGUE

CLARIFICATION 1: SIZE OF PHYSICAL AND VIRTUAL MEMORY ADDRESSES

On 64-bit machines, we have 64-bit addresses for virtual and physical memory.

In the last years, Intel CPUs used 4-level paging and thus the lower 48 bits virtually to address bytes in memory. Most recently, Intel CPU Ice Lake (Aug 2019) uses 5-level paging and thus uses 57 bits effectively.

CLARIFICATION 1: SIZE OF PHYSICAL AND VIRTUAL MEMORY ADDRESSES

For example, the Objective-C runtime on iOS 7 on ARM64, notably used on the iPhone 5S, uses *tagged pointers*.

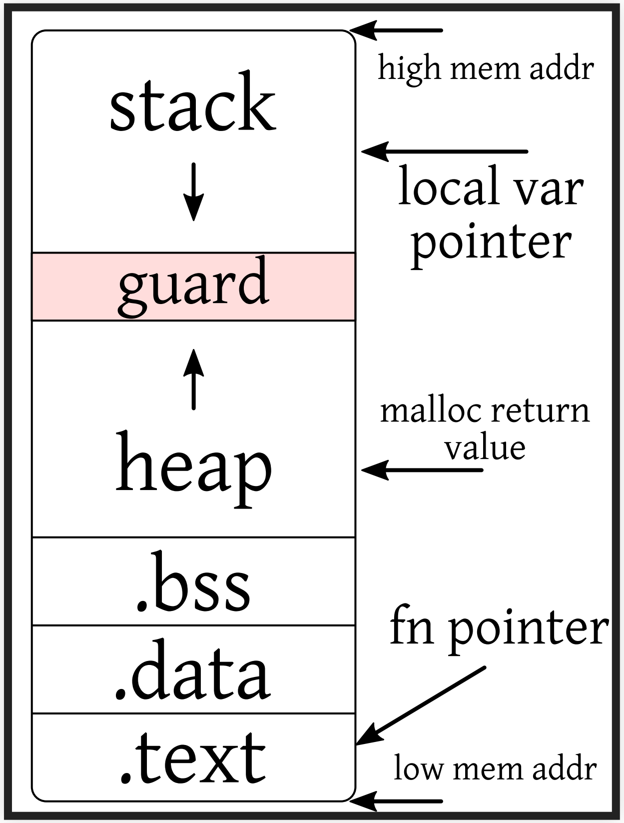
48 bits ⇒ 256 TiB, 57 bits ⇒ 128 PiB

CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?

- Collision is called “Clashing”
- There is a guard page between stack and heap
- Access to the guard page triggers a segfault
- 64-bit address space is huge. You have plenty of space.

via Qualys Security Advisory and stackoverflow

CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?



CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?

```rust
fn count_calls(n: u64) -> u64 {
    if n < 1 {
        0
    } else {
        1 + count_calls(n - 1)
    }
}

fn main() {
    println!("{}", count_calls(174470))
}
```

CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?

```rust
fn count_calls(n: u64) -> u64 {
    if n < 1 {
        0
    } else {
        1 + count_calls(n - 1)
    }
}

fn main() {
    println!("{}", count_calls(174470))
}
```

Non-deterministic, but 174470 will crash in ~59/1000 cases.

CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?

- Thinkpad T495s, AMD Ryzen 7 PRO 3700U, 16GB RAM
- 160 bytes per function call frame
- ⇒ 27,915,200 bytes (between 2^{24} and 2^{25}) allocated on stack before stackoverflow

CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?

- Thinkpad T495s, AMD Ryzen 7 PRO 3700U, 16GB RAM
- 160 bytes per function call frame
- $\Rightarrow$ 27,915,200 bytes (between 2^{24} and 2^{25}) allocated on stack before stackoverflow

160 bytes? That much? We debugged it. print and formatting is one reason (34 bytes).

CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?

- Thinkpad T495s, AMD Ryzen 7 PRO 3700U, 16GB RAM
- 160 bytes per function call frame
- $\Rightarrow$ 27,915,200 bytes (between 2^{24} and 2^{25}) allocated on stack before stackoverflow

160 bytes? That much? We debugged it. print and formatting is one reason (34 bytes).

We compiled it optimized. Only a constant was left!

CLARIFICATION 2: STACK & HEAP COLLIDE. WHY A SEGFAULT?

```
thread 'main' has overflowed its stack
fatal runtime error: stack overflow
[1]    26166 abort      ./elf-executable
```

CLARIFICATION 3: ALLOCATE MEMORY INSIDE A FOR-LOOP

It is part of the function call stack frame, right?

```
fn main() {
    let a = 42;
    let b = 2;
    println!("{:p} {:p}", &a, &b);
    for i in 1..10 {
        let c = a + i * b;
        println!("{:p}", &c);
    }
}
```

CLARIFICATION 3: ALLOCATE MEMORY INSIDE A FOR-LOOP

```
0x7ffd4187ff20 0x7ffd4187ff24
0x7ffd4187ffb4
0x7ffd4187ffb4
0x7ffd4187ffb4
0x7ffd4187ffb4
0x7ffd4187ffb4
0x7ffd4187ffb4
0x7ffd4187ffb4
0x7ffd4187ffb4
0x7ffd4187ffb4
```

⇒ Yes

CLARIFICATION 4: DUMP THE STACK

→ Inline assembly

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

- unstable API, only on nightly
- Only in `unsafe { }` blocks

CLARIFICATION 4: DUMP THE STACK

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn dump_stack() {
    let nr_elements = 70;
    for i in 0..nr_elements {
        let offset = nr_elements - i - 1;
        let mut result: u64;
        unsafe {
            asm!("movq %rsp, %rax
                  addq %rbx, %rax
                  movq (%rax), %rcx"
                 : "={rcx}"(result)
                 : "{rbx}"(8 * offset)
                 : "rax", "rbx", "rcx"
```

CLARIFICATION 4: DUMP THE STACK

```
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
#[inline(always)]
fn print_current_address() {
    let mut result: u64;
    unsafe {
        // https://stackoverflow.com/a/52050776
        asm!("leaq (%rip), %rax"
             : "={rax}"(result)
             :
             : "rax"
             :)
    }
    println!("rip = {:016x}", result);
}
```

CLARIFICATION 4: DUMP THE STACK

```
fn sub(sub_arg: u64) -> u64 {
    print_current_address();
    let sub_local = 0xDEAD_C0DE;
    let sub_sum = sub_arg + sub_local;
    dump_stack();
    sub_sum
}
fn main() {
    let _main_a: u64 = 0xDEAD_BEEF;
    let main_arg: u64 = 0xFEED_C0DE;
    print_current_address();
    let main_ret = sub(main_arg);
    assert_eq!(main_ret, 0x0000_0001_DD9B_81BC);
    let s = sub as *const ();
    let m = main as *const ();
```

CLARIFICATION 4: DUMP THE STACK

```
rip = 00005639782d660f             early main instruction
rip = 00005639782d650a             early sub instruction
sp+552 ⇒ value 00000000feedc0de  main_arg local var
sp+544 ⇒ value 00000000deadbeef  _main_a local var
sp+536 ⇒ value 00005639782f8f18
sp+528 ⇒ value 0000000000000002
sp+520 ⇒ value 0000563979f16920
sp+512 ⇒ value 0000563979f16920
sp+504 ⇒ value 0000563979f16920
sp+496 ⇒ value 0000563979f16971
sp+488 ⇒ value 0000000000000000
sp+480 ⇒ value 0000000000000000
sp+472 ⇒ value 0000000000000000
sp+464 ⇒ value 0000000000000000
sp+456 ⇒ value 0000000000000000
```

DIALOGUE

TODAY

- Attributes and Feature guards
- Just for fun
- Recap → Quiz
- A tiny I/O example
- Stack
- Stack with String
- On pointers and references
- Graph

ATTRIBUTES AND FEATURE GUARDS

An *attribute* is metadata applied to some module, crate or item. This metadata can be used to/for:

- conditional compilation of code
- set crate name, version and type (bin/lib)
- disable lints (warnings)
- enable compiler features (macros,etc.)
- link to a foreign library
- mark functions as unit tests
- mark functions that will be part of a benchmark

ATTRIBUTES AND FEATURE GUARDS

```rust
// syntaxes
#[attribute = "value"]
#[attribute(key = "value")]
#[attribute(value)]
// multi-value attributes
#[attribute(value, value2)]
#[attribute(value, value2, value3,
            value4, value5)]
```

via rust by example

ATTRIBUTES AND FEATURE GUARDS

```
// ! means “scope is global = crate”
#![crate_type = "lib"]
// no ! means “scope is local = module/item”
#[allow(dead_code)]
```

ATTRIBUTES AND FEATURE GUARDS

Examples:

```rust
#![crate_type = "lib"]  // crate is library, not binary
#![crate_name = "rary"] // named “rary”

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
#[inline(always)]
fn arch_specific_code() {}

#[allow(dead_code)]
fn unused_function() {}
```

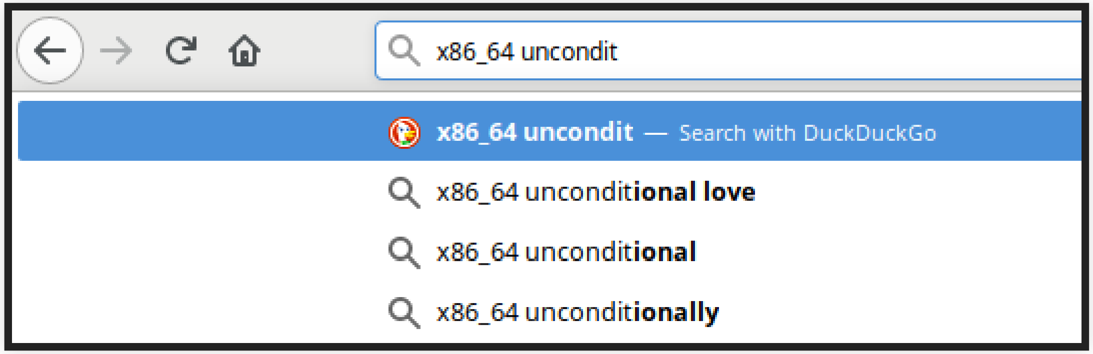
ATTRIBUTES AND FEATURE GUARDS:

```rust
#![feature(box_syntax)]

#[allow(unused_variables)]
fn main() {
    let b = box 5;
}
```

… on unstable, instead of …

```rust
#[allow(unused_variables)]
fn main() {
    let b = Box::new(5);
}
```

JUST FOR FUN


x86_64 uncondit
x86_64 uncondit — Search with DuckDuckGo
x86_64 unconditional love
x86_64 unconditional
x86_64 unconditionally

JUST FOR FUN

Does it compile?

```
#include <stdio.h>
int main() {
  printf("Hello %s!\n",
    #include "/dev/stdin"
  );
  return 0;
}
```

JUST FOR FUN

```
meisterluk@gardner ~ % cat input
"RustGraz"
meisterluk@gardner ~ % cat input | gcc test.c
meisterluk@gardner ~ % ./a.out
Hello RustGraz!
```

Yes on gcc, no on clang. via Poita_ on twitter

RECAP (LAST SESSION)

- `&a` is a (shared) reference. `&mut a` is a mutable reference.
- `*a` dereferences a variable (sometimes auto-dereferencing happens)
- Every variable has an owner. Owner can borrow ownership.
- Only owners can destroy the variable.
- References borrow ownership, they don't move ownership.

The ultimate rule: **aliasing XOR mutation**

QUIZ

```rust
fn sub(s: &mut String) {
    s.push('o');
}

fn main() {
    let mut base = String::from("f");
    sub(&mut base);
    sub(&mut base);
    println!("{} bar", base);
}
```

Compiles?

QUIZ

```
fn sub(s: &mut String) {
    s.push('o');
}

fn main() {
    let mut base = String::from("f");
    sub(&mut base);
    sub(&mut base);
    println!("{} bar", base);
}
```

Compiles? Yes.

QUIZ

```rust
fn sub(s: &mut String) {
    s.push('o');
}

fn main() {
    let mut base = String::from("f");
    let c = &mut base;
    sub(c);
    let d = &mut base;
    sub(d);
    println!("{} bar", base);
}
```

Compiles?

QUIZ

```rust
fn sub(s: &mut String) {
    s.push('o');
}

fn main() {
    let mut base = String::from("f");
    let c = &mut base;
    sub(c);
    let d = &mut base;
    sub(d);
    println!("{} bar", base);
}
```

Compiles? Yes.

QUIZ

```rust
fn sub(s: &mut String) {
    s.push('o');
}

fn main() {
    let mut base = String::from("f");
    let c = &mut base;
    let d = &mut base;
    sub(c);
    sub(d);
    println!("{} bar", base);
}
```

Compiles?

QUIZ

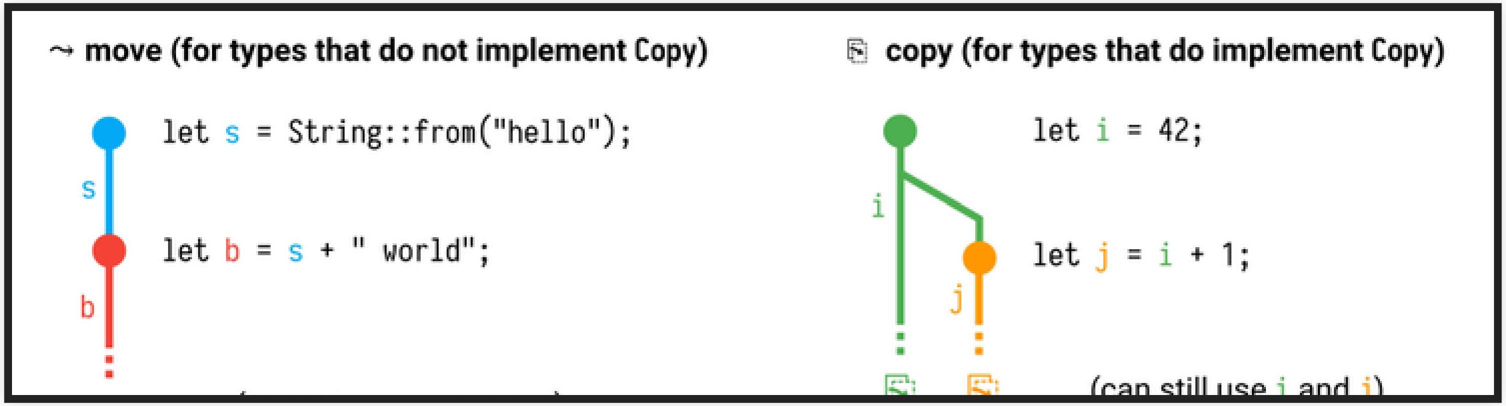
```rust
fn sub(s: &mut String) {
    s.push('o');
}

fn main() {
    let mut base = String::from("f");
    let c = &mut base;
    let d = &mut base;
    sub(c);
    sub(d);
    println!("{} bar", base);
}
```

Compiles? No. “cannot borrow base as mutable more than once at a time”.

QUIZ

Remark: Sometimes borrowing is explained with “not more than one mutable reference allowed in a scope”. It is a little bit more complicated IMHO. Aaron Turon explains it with overlapping regions.

An approach to visualization: rufflewind.com



A TINY I/O EXAMPLE

A TINY I/O EXAMPLE

std::io::Stdin

- `pub fn lock(&self) -> StdinLock`: Locks
  handle to the standard input stream, returning a rea
  guard.
- `pub fn read_line(&self, buf: &mut St`
  `-> Result<usize>`: Locks this handle and reads
  of input into the specified buffer.

`StdinLock` implements the `Read` and `BufRead` traits ⇒ useful methods.

A TINY I/O EXAMPLE

```
use std::io::{self, Write};

fn read_line() -> String {
    let mut input = String::new();
    match io::stdin().read_line(&mut input) {
        Ok(n) => {
            eprintln!("{} bytes, {:?}, {:?}", n,
                      input.into_bytes(), input);
            input
        }
        Err(error) => panic!(error),
    }
}
```

A TINY I/O EXAMPLE

```rust
use std::io::{self, Write};

fn read_line() -> String {
    let mut input = String::new();
    match io::stdin().read_line(&mut input) {
        Ok(n) => {
            eprintln!("{} bytes, {:?}, {:?}", n,
                      input.into_bytes(), input);
            input
        }
        Err(error) => panic!(error),
    }
}
```

Does this code compile?

A TINY I/O EXAMPLE

```
error[E0382]: borrow of moved value: `input`
 --> src/main.rs:7:70
  |
4 | let mut input = String::new();
  |     --------- move occurs because `input` has type
  |     `std::string::String`, which does not implement
  |     the `Copy` trait
...
7 | eprintln!("{} bytes, {:?}, {:?}", n,
  |           input.into_bytes(), input);
  |           -----               ^^^^^
  |             |                 value moved here
  |           value borrowed here after move
```

A TINY I/O EXAMPLE

`pub fn into_bytes(self) -> Vec<u8>`:

Converts a String into a byte vector. This consumes the String, so we do not need to copy its contents.

```
let n = 42;
let mut input = String::new();
eprintln!("{} bytes, {:?}, {:?}", n,
  String::from(input).into_bytes(), input);
```

Does this code compile?

A TINY I/O EXAMPLE

`pub fn into_bytes(self) -> Vec<u8>`:

Converts a String into a byte vector. This consumes the String, so we do not need to copy its contents.

```
let n = 42;
let mut input = String::new();
eprintln!("{} bytes, {:?}, {:?}", n,
  String::from(input).into_bytes(), input);
```

Does this code compile? No, still consuming.

A TINY I/O EXAMPLE

```
let n = 42;
let mut input = String::new();
eprintln!("{} bytes, {:?}, {:?}", n,
    String::from(&input).into_bytes(), input);
```

It compiles!

A TINY I/O EXAMPLE

```
fn main() {
    loop {
        print!("⇐ ");
        io::stdout().flush().unwrap();
        let line = read_line();
        if line == "" || line == "0\n" {
            break
        }
        println!("⇒ {}", line)
    }
}
```

⇒ We use it to fetch dynamic data.

DATA STRUCTURE: STACK

DATA STRUCTURE: STACK

- Do you remember stack and heap from last session?
- PUSH and POP operation
- PUSH: add element to the top
- POP: remove element from the top and return it
- `O(1)` time complexity for both operations

DATA STRUCTURE: STACK

```rust
#[derive(Debug)]
struct Stack {
    content: Vec<u64>
}

impl Stack {
    fn new() -> Stack {
        Stack { content: Vec::<u64>::new() }
    }

    fn push(&mut self, element: u64) -> &mut Self {
        self.content.push(element);
        self
    }
```

DATA STRUCTURE: STACK

```
use std::fmt;
use std::io::{self, Write};

impl fmt::Display for Stack {
    fn fmt(&self, f: &mut fmt::Formatter<'_>)
    -> fmt::Result {
        write!(f, "Stack[{}]", self.content
          .iter().map(|uint| { uint.to_string() })
          .collect::<Vec<String>>().join(" "))
    }
}
```

DATA STRUCTURE: STACK

```rust
// Example usage
fn main() {
    let s = Stack::new();
    s.push(1).push(1).push(2).push(3).push(5).push(8);
    println!("{}", s.pop());
}
```

Does it compile?

DATA STRUCTURE: STACK

```rust
// Example usage
fn main() {
    let s = Stack::new();
    s.push(1).push(1).push(2).push(3).push(5).push(8);
    println!("{}", s.pop());
}
```

Does it compile? No.

DATA STRUCTURE: STACK

```rust
// Example usage
fn main() {
    let mut s = Stack::new();
    s.push(1).push(1).push(2).push(3).push(5).push(8);
    println!("{}", s.pop());
}
```

s is now mutable. Does it compile?

DATA STRUCTURE: STACK

```rust
// Example usage
fn main() {
    let mut s = Stack::new();
    s.push(1).push(1).push(2).push(3).push(5).push(8);
    println!("{}", s.pop());
}
```

`s` is now mutable. Does it compile? No.

DATA STRUCTURE: STACK

```rust
// Example usage
fn main() {
    let mut s = Stack::new();
    s.push(1).push(1).push(2).push(3).push(5).push(8);
    println!("{}", s.pop().unwrap());
}
```

`pop()` returns `std::option::Option<u64>`.

DATA STRUCTURE: STACK

```rust
// Example usage
fn main() {
    let mut s = Stack::new();
    s.push(1).push(1).push(2).push(3).push(5).push(8);
    println!("{}", s.pop().unwrap());
}
```

`pop()` returns `std::option::Option<u64>`.

It compiles.

DATA STRUCTURE: STACK

```
// Parse a line and push integer to the stack
fn handle_input(s: &mut Stack, input: &String) {
    let mut provided_integer = 0;
    let mut valid = true;
    let input_trimmed = input.trim();
    match input_trimmed.parse::<u64>() {
        Ok(val) => provided_integer = val,
        Err(_) => {
            println!("… is not an integer");
            valid = false;
        },
    }
    if valid {
        s.push(provided_integer);
    }
```

DATA STRUCTURE: STACK

More advanced:

```
#[derive(PartialEq)]
enum Op {
  Pop,
  Push,
  Print,
}

…
match op {
  Op::Push => { s.push(provided_integer); },
  Op::Pop => { println!("{:?}", s.pop()); },
  Op::Print => { println!("{}", s); },
}
```

DATA STRUCTURE: STACK

```rust
fn handle_input(s: &mut Stack, input: &String) {
  let input_trimmed = input.trim();
  let mut fields = input_trimmed.split_whitespace();

  let op = match fields.next() {
    Some("push") => Op::Push,
    Some("pop") => Op::Pop,
    Some("print") => Op::Print,
    Some(_) | None => {
      eprintln!("unknown operation");
      return;
    },
  };
  …
```

DATA STRUCTURE: STACK

```
…
let mut provided_integer = 0;
if op == Op::Push {
    match fields.next() {
        Some(val) => {
            match val.parse::<u64>() {
                Ok(val) => provided_integer = val,
                Err(_) => {
                    eprintln!("… is not an integer");
                    return;
                },
            }
        },
        None => {
            eprintln!("I find this lack of arguments distu
```

DATA STRUCTURE: STACK

```
    …
    match op {
        Op::Push => { s.push(provided_integer) },
        Op::Pop => { println!("{:?}", s.pop()) },
        Op::Print => { println!("{}", s) },
    }
}
```

Does it compile?

DATA STRUCTURE: STACK

```
    …
    match op {
        Op::Push => { s.push(provided_integer) },
        Op::Pop => { println!("{:?}", s.pop()) },
        Op::Print => { println!("{}", s) },
    }
}
```

Does it compile? No, unmatching types in branches

DATA STRUCTURE: STACK

```
    …
    match op {
        Op::Push => { s.push(provided_integer); },
        Op::Pop => { println!("{:?}", s.pop()); },
        Op::Print => { println!("{}", s); },
    }
}
```

Does it compile? Yes, with semicolons.

DATA STRUCTURE: STACK

```
    …
    match op {
        Op::Push => { s.push(provided_integer); },
        Op::Pop => { println!("{:?}", s.pop()); },
        Op::Print => { println!("{}", s); },
    }
}
```

DATA STRUCTURE: STACK

```
fn main() {
    let mut s = Stack::new();
    loop {
        print!("⇐ ");
        io::stdout().flush().unwrap();
        let line = read_line();
        if line == "" || line == "bye\n" {
            break
        }
        handle_input(&mut s, &line);
        println!("⇒ {}", line)
    }
    println!("{}", s);
}
```

DATA STRUCTURE: STACK

```
% cargo run
⇐ push 42
⇒ push 42

⇐ push 73
⇒ push 73

⇐ print
Stack[42 73]
⇒ print

⇐ pop
Some(73)
⇒ pop
```

DATA STRUCTURE: STACK WITH STRINGS

DATA STRUCTURE: STACK WITH STRINGS

Use `String` instead of `u64` as elements. What is the interesting difference?

→ Replace every occurence of `u64` with `String`.

Does it compile?

DATA STRUCTURE: STACK WITH STRINGS

Use `String` instead of `u64` as elements. What is the interesting difference?

→ Replace every occurence of `u64` with `String`.

Does it compile?

Yes.

ON POINTERS AND REFERENCES

Pointers in C/C++:

{NULL Pointer, Dangling Pointer, Generic Pointer, Wild Pointer, Near Pointer, Far Pointer, Huge Pointer, Fat pointer, Shared pointer, Smart pointer, Unique pointer, Weak pointer}

ON POINTERS AND REFERENCES

NULL pointer

- defined e.g. in `stddef.h`
- Tony Hoare “I call it my billion-dollar mistake. It was the invention of the null reference in 1965”
- Use Option/Result instead in rust

ON POINTERS AND REFERENCES

Dangling Pointer

- object A exists
- pointer p points to A
- object A vanishes
- pointer p becomes a Dangling pointer
- Impossible in rust to the best of my knowledge

ON POINTERS AND REFERENCES

Generic Pointer

- aka “void pointer”
- std::ptr::null_mut
- rust has a strong, full-featured type system
- thus only used for FFI in rust

ON POINTERS AND REFERENCES

Wild pointer

- i.e. uninitialized pointer
- `int *ptr;` in C
- `let a: Box<u32>; a = Box::new(5);` in rust
- rust ensure: no read before write!

ON POINTERS AND REFERENCES

Near Pointer

- can only address values in (e.g.) ±32KB data segment

Far Pointer

- can address values outside (e.g.) ±32KB data segment

Non-standard Intel extension (16 bit architecture).

ON POINTERS AND REFERENCES

Huge Pointer

Like far pointer, but points to consecutive memory blocks (→ memory management details)

ON POINTERS AND REFERENCES

Fat Pointer

```
size_of::<&u32>() = 8
size_of::<&[u32; 2]>() = 8
size_of::<&[u32]>() = 16
```

`&[u32]` must carry around the length of the memory view. Thus needs more space.

via stackoverflow

ON POINTERS AND REFERENCES

Shared pointer

Manages the storage of a pointer, providing a limited garbage-collection facility, possibly sharing that management with other objects.

via C++: shared_ptr

ON POINTERS AND REFERENCES

Smart pointer

> *A smart pointer is an abstract data type that simulates a pointer while providing added features, such as automatic memory management or bounds checking*

via Wikipedia

ON POINTERS AND REFERENCES

Unique pointer

> *`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope.*

via C++: unique_ptr

ON POINTERS AND REFERENCES

Weak pointer

> *`std::weak_ptr` is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by std::shared_ptr. It must be converted to std::shared_ptr in order to access the referenced object.*

via C++: weak_ptr

STD::RC::RC

- A single-threaded reference-counting pointer. 'Rc' stands for 'Reference Counted'.
- The type `Rc<T>` provides shared ownership of a value of type `T`, allocated in the heap. Invoking `clone` on Rc produces a new pointer to the same value in the heap. When the last Rc pointer to a given value is destroyed, the pointed-to value is also destroyed.

via rust doc

DATA STRUCTURE: GRAPH

- A graph consists of vertices and edges
- Vertices are distinctive and edges are tuples (directed) or sets (undirected) of 2 edges
- e.g. DAG – Directed Acyclic Graph
- See code example

VALGRIND TO CHECK FOR MEMORY LEAKS

```
% valgrind ./target/debug/strstack
==4731== Memcheck, a memory error detector
==4731== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Sew
==4731== Using Valgrind-3.14.0 and LibVEX; rerun with -h for c
==4731== Command: ./target/debug/stack
==4731==
⇐ push foo
⇒ push foo

⇐ push bar
⇒ push bar

⇐ push baz
⇒ push baz
```

EPILOGUE

QUIZ

How can we print a pointer with fmt?

What is rust's ultimate borrowing rule?

Which print macro prints to stderr?

How can we parse a string into an integer?

Why was `#[derive(PartialEq)]` required?

QUIZ

How can we print a pointer with fmt?

`{:p}`

What is rust's ultimate borrowing rule?

Which print macro prints to stderr?

How can we parse a string into an integer?

Why was `#[derive(PartialEq)]` required?

QUIZ

How can we print a pointer with fmt?

`{:p}`

What is rust's ultimate borrowing rule?

aliasing XOR mutation

Which print macro prints to stderr?

How can we parse a string into an integer?

Why was `#[derive(PartialEq)]` required?

QUIZ

How can we print a pointer with fmt?

`{:p}`

What is rust's ultimate borrowing rule?

aliasing XOR mutation

Which print macro prints to stderr?

e-prefixed `eprintln!(…)`

How can we parse a string into an integer?

Why was `#[derive(PartialEq)]` required?

QUIZ

How can we print a pointer with fmt?

`{:p}`

What is rust's ultimate borrowing rule?

aliasing XOR mutation

Which print macro prints to stderr?

e-prefixed `eprintln!(…)`

How can we parse a string into an integer?

`stringvar.parse::<u64>()`

Why was `#[derive(PartialEq)]` required?

QUIZ

How can we print a pointer with fmt?

`{:p}`

What is rust's ultimate borrowing rule?

aliasing XOR mutation

Which print macro prints to stderr?

e-prefixed `eprintln!(…)`

How can we parse a string into an integer?

`stringvar.parse::<u64>()`

Why was `#[derive(PartialEq)]` required?

`op == Op::Push` requires it, not `match`

NEXT SESSION

Wed, 2019/12/25 19:00?
→ 2019/12/18 instead

My suggestion:

1. **Short topic:** Strings, string types and UTF-8
2. **Fun:** Hacker jeopardy

January: Traits, generics
Feb+: lifetimes, concurrency, macros, crates, crates, crates …