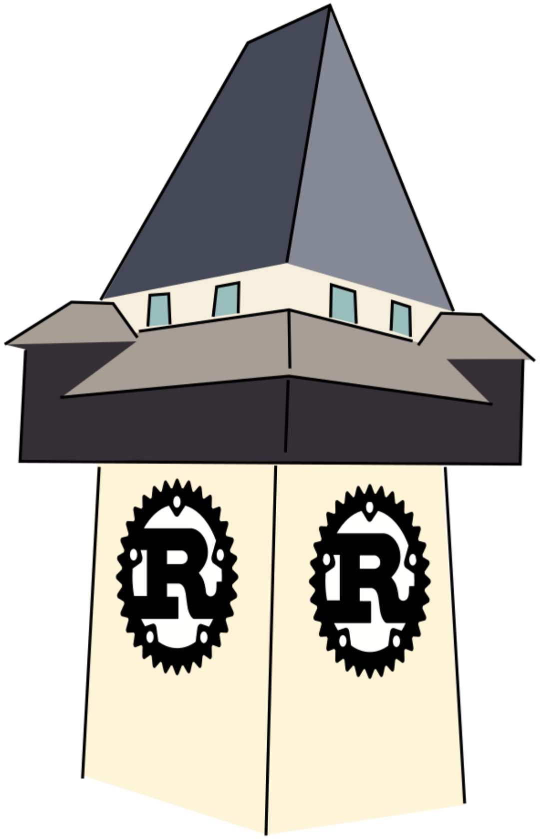
RUST GRAZ – 04 REFERENCES AND BORROWING

Lukas Prokop

30th of October, 2019



Stack, heap, references and borrowing

30th of October 2019

Rust Graz, lab10

PROLOGUE

CLARIFICATION 1: FLOAT RANGES

→ std::ops::Range. Reminder:

```
assert_eq!((3..5), std::ops::Range { start: 3, end: 5 });
assert_eq!(3 + 4 + 5, (3..6).sum());

let arr = [0, 1, 2, 3, 4];
assert_eq!(arr[ ..  ], [0,1,2,3,4]);
assert_eq!(arr[ .. 3], [0,1,2    ]);
assert_eq!(arr[ ..=3], [0,1,2,3  ]);
assert_eq!(arr[1..  ], [  1,2,3,4]);
assert_eq!(arr[1.. 3], [  1,2    ]);
assert_eq!(arr[1..=3], [  1,2,3  ]);
```

CLARIFICATION 1: FLOAT RANGES

→ std::ops::Range

start: Idx ⇒ The lower bound of the range (inclusive).

end: Idx ⇒ The upper bound of the range (exclusive).

- `pub fn contains<U>(&self, item: &U)`
- `pub fn is_empty(&self) -> bool` [nightly

But e.g. one of the implementations is

`impl<A> Iterator for Range<A> where A:`

```
let mut numbers = 1..100;
assert_eq!(numbers.nth(42), Some(42));
```

CLARIFICATION 1: FLOAT RANGES

Exhaustive list of implementors of `std::iter::Step`:

- `i8`, `u8`
- `i16`, `u16`
- `i32`, `u32`
- `i64`, `u64`
- `i128`, `u128`
- `isize`, `usize`

No floats 😔

CLARIFICATION 1: FLOAT RANGES

```
error[E0277]: the trait bound `{float}: std::iter::Step` is no
 --> range.rs:2:14
  |
2 | for i in 2.7 .. 3.1 {
  |          ^^^^^^^^^^ the trait `std::iter::Step`
  |                 is not implemented for `{float}`
  |
  = help: the following implementations were found:
    <i128 as std::iter::Step> <i16 as std::iter::Step>
    <i32 as std::iter::Step>  <i64 as std::iter::Step> and 8 o
  = note: required because of the requirements on the impl
    of `std::iter::Iterator` for `std::ops::Range<{float}>`

error: aborting due to previous error
```

CLARIFICATION 1: FLOAT RANGES

You can print them though!

```
println!("{}", 2.7 .. 3.1);
println!("{}", std::ops::Range { start: 2.7, end: 3.1 });
```

gives

```
2.7..3.1
2.7..3.1
```

CLARIFICATION 1: FLOAT RANGES

Custom float range implementation:

```
#[derive(Debug,Clone,Copy)]
struct IterableFloat {
    start: f64,
    end: f64,
    step: f64,

    init: bool,
    value: f64,
}
```

CLARIFICATION 1: FLOAT RANGES

```
impl IterableFloat {
    fn default() -> IterableFloat {
        IterableFloat {
            start: 2.7, end: 3.1, step: 0.1,
            init: true, value: 2.7,
        }
    }
    fn from_to(start: f64, end: f64, step: f64)
                                    -> IterableFloat {
        IterableFloat {
            start: start, end: end, step: step,
            init: true, value: start,
        }
    }
}
```

CLARIFICATION 1: FLOAT RANGES

```rust
impl Iterator for IterableFloat {
    fn next(&mut self) -> Option<IterableFloat> {
        if !self.init {
            return None
        }
        if self.value + self.step > self.end {
            return None
        }
        self.value += self.step;
        Some(self.clone())
    }
}
```

CLARIFICATION 1: FLOAT RANGES

Using this iterator:

```rust
fn main() {
    let ifloat = IterableFloat::default();
    //println!("{:?}", ifloat.next().unwrap());

    for f in ifloat {
        println!("{:?}", f);
    }

    for f in IterableFloat::from_to(2.7, 3.1, 0.1) {
        println!("{:?}", f);
    }
}
```

CLARIFICATION 2: NEGATIVE RANGES

What does it print?

```
fn main() {
    for i in 1..-5 {
        println!("{}", i);
    }
}
```

CLARIFICATION 2: NEGATIVE RANGES

What does it print?

```
fn main() {
    for i in 1..-5 {
        println!("{}", i);
    }
}
```

Nothing.

CLARIFICATION 2: NEGATIVE RANGES

Simply unsupported.

CLARIFICATION 3: STEP SIZED RANGES

```
fn main() {
    for i in (3..10).step_by(2) {
        println!("{}", i);
    }
}
```

BTW, `step_by(0)` is a runtime error (also in release mode).

CLARIFICATION 3: STEP SIZED RANGES

```
fn main() {
    for i in (1..-5).step_by(-1) {
        println!("{}", i);
    }
}
```

CLARIFICATION 3: STEP SIZED RANGES

```
error[E0600]: cannot apply unary operator `-` to type `usize`
 --> step_by.rs:2:30
  |
2 |     for i in (1..-5).step_by(-1) {
 cannot apply unary operator `-` ^^
  |
   = note: unsigned values cannot be negated

error: aborting due to previous error

For more information about this error, try `rustc --explain E0
```

CLARIFICATION 4: HOW TO RETURN NON-ZERO EXIT CODES

```rust
fn main() {
    std::process::exit(3);
}
```

Note that because this function never returns, and that it terminates the process, no destructors on the current stack or any other thread's stack will be run. If a clean shutdown is needed it is recommended to only call this function at a known point where there are no more destructors left to run.

CLARIFICATION 5: THE NEVER DATA TYPE

```
pub fn exit(code: i32) -> !
```

- `!` is called never
- `break`, `continue` and `return` expressions have type `!`
- What does `Result<!, E>` represent?

CLARIFICATION 5: THE NEVER DATA TYPE

From the library of cryptic source code (compiles only in nightly):

```
#![allow(unused)]
#![feature(never_type)]
fn main() {
  fn foo() -> u32 {
    let x: ! = {
      return 123
    };
  }
}
```

ADDENDUM 6: VECTOR PREALLOCATION

My own question: How to preallocate a Vec?

```go
// Go, preallocate a byte slice
data := make([]byte, 0, 10)
fmt.Printf("%d\n", len(data))
fmt.Printf("%d\n", cap(data))
```

```rust
// rust, preallocate a vector
let mut data = Vec::with_capacity(10);
assert_eq!(data.len(), 0);
for i in 0..10 {
    data.push(i);
}
data.push(11);  // might reallocate
```

DIALOGUE: BORROWING AND REFERENCES

OS/processor support ⇒ memory allocation ⇒ memory safety ⇒ borrowing

CPU ARCHITECTURES

- x86/i386/IA-32 (Intel 8086, 1978)
- amd64/x86_64 (>95% desktop computers, 2000)
- ARM Cortex (Raspberry PI, smartphone)
- RISC-V (first, major open source CPU architecture)

We can build CPUs using AND/XOR/NOT/… gates. FlipFlops allows us to store state temporarily. A CPU consists of ALUs, state machines, …. A program is stored in main memory and an instruction pointer points to the currently executed instruction.

CPU ARCHITECTURES

Awesome fun fact:

Current state of the art is 5 nm technology – as of October 2018, the average half-pitch of a memory cell expected to be manufactured circa 2019-2020

History: 90nm → 65nm → 45nm → 32nm → 22nm → 14nm → 10nm → 5nm → 3nm

INSTRUCTION SET ARCHITECTURE (ISA)

- An ISA specifies the instructions understood by the CPU (interface between hardware & software)
- ISAs/CPUs compete in feature-richness, implementation complexity and speed
- Compilers (rustc, gcc, …) take your high-level program and convert it into CPU instructions.
- RISC (reduced instruction set computer) versus CISC (complex instruction set computer)

CPU ARCHITECTURES

RISC → e.g. RISC-V, PowerPC
CISC → e.g. x86

Example instruction: **MOV** - Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers via x86_64 ISA (4922 pages)

CPUS AND MEMORY

- An ISA defines how to address registers, main memory and I/O
- OS: every process assumes it has the entire memory available
    - my machine: 16GB
    - separation in physical/virtual addresses
    - is an important security mechanism
    - memory is organized in pages, 4kB on Linux in my case

X86_64

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b

x86_64 registers via stackoverflow

X86_64

- Data Transfer Instructions
- Binary Arithmetic Instructions
- Decimal Arithmetic Instructions
- Logical Instructions
- Shift and Rotate Instructions
- Bit and Byte Instructions
- Control Transfer Instructions
- String Instructions
- I/O Instructions …

In total: between 981 and 3683 instructions (x86_64)

X86_64

X86_64

Fun fact:

mov is Turing-complete

⇒ M/o/Vfuscator - The single instruction C compiler

RISC-V

RISC-V ("risk-five") is an open-source hardware instruction set architecture (ISA) based on established reduced instruction set computer principles.

The project began in 2010 at the University of California, Berkeley, but many contributors are volunteers not affiliated with the university.

As of June 2019, version 2.2 of the user-space ISA and version 1.11 of the privileged ISA are frozen, permitting software and hardware development to proceed. A debug specification is available as a draft, version 0.3.

via Wikipedia

RISC-V

RISC-V is a *Load-store architecture*: Arithmetic instructions can only operate on registers, not memory. Thus, instructions fall in two categories:

- *memory access* (load and store between memory and registers)
- *ALU operations* (which only occur between registers)

RV32I, RV32E, RV64I, RV128I. And about 15 extensions in 2.2 release.

RISC-V EXAMPLE

```
# int offset, val, s;
# int *addr;

main:
    ADDI t0, zero, 1     # int step_size = 1;
    ADDI t1, zero, 0     # int i = 0;
    ADDI t2, zero, array # int ref = &array;
    JAL zero, test       # goto test;
```

RISC-V EXAMPLE

```
test:
    LW t6, n                # s = n;
    BLT t1, t6, body        # if (i < s) goto body;
    LW t6, sum              # s = sum;
    SW t6, 0x7FC(zero)      #// write s to stdout
    EBREAK                  #// return main routine

n:      .word 4             # number of elements in n
array: .word 3, 4, 5, 6  # array to sum up
sum:    .word 0             # sum should finally be 18 == 0x12
```

RISC-V EXAMPLE

```
body:
    SLL t3, t1, t0       # offset = i << 1
    SLL t3, t3, t0       # offset = offset << 1
                         # offset in t3 now contains i*4
    ADD t4, t3, t2       # addr = &array[offset]
    LW t4, 0(t4)         # val = *addr;
    LW t6, sum           # s = *sum;
    ADD t6, t6, t4       # s = s + val;
    SW t6, sum           # *sum = s;
    ADDI t1, t1, 1       # i++;
```

RISC-V COMPILATION

Install RISC-V backend:

`rustup target add riscv32imac-unknown-n`

```
error[E0463]: can't find crate for `std`
  |
  = note: the `riscv64imac-unknown-none-elf` target may not be

error: aborting due to previous error

For more information about this error, try `rustc --explain E0
```

😓 I still don't know how to compile for RISC-V.

ASSEMBLING RUST WITH X86_64

Another example:

```rust
fn main() {
    let x = 5;
    let y = 6;
    println!("{}", x + y);
}
```

ASSEMBLING RUST WITH X86_64

```
$ rustc --print target-spec-json
        -Z unstable-options assembly.rs
{ "arch": "x86_64",
  "cpu": "x86-64",
  "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
  "dynamic-linking": true,
… "llvm-target": "x86_64-unknown-linux-gnu",
… "os": "linux",
  "position-independent-executables": true,
… "stack-probes": true,
  "target-c-int-width": "32",
  "target-endian": "little",
  "target-family": "unix",
  "target-pointer-width": "64",
  "vendor": "unknown" }
```

ASSEMBLING RUST WITH X86_64

⇒ Executables are compiled for a specific operating system on a specific CPU architecture

```
% rustc test.rs
% file test
test: ELF 64-bit LSB shared object, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/l,
for GNU/Linux 3.2.0, BuildID[sha1]=3ec8f851df61d55cd16c48b66da
with debug_info, not stripped
% objdump -d test | grep -A 8 main
```

ASSEMBLING RUST WITH X86_64

Remark: There are two assembly notations. You are going to see “AT&T/GAS syntax”, not “Intel syntax”

```
0000000000004140 <main>:
    4140: 50                    push   %rax
    4141: 48 63 c7              movslq %edi,%rax
    4144: 48 8d 3d 45 ff ff ff  lea    -0xbb(%rip),%rdi
    414b: 48 89 34 24           mov    %rsi,(%rsp)
    414f: 48 89 c6              mov    %rax,%rsi
    4152: 48 8b 14 24           mov    (%rsp),%rdx
    4156: e8 15 00 00 00        callq  4170 <_ZN3std2rt10lang_
    415b: 59                    pop    %rcx
    415c: c3                    retq
    415d: 0f 1f 00              nopl   (%rax)
```

Wikibooks: GAS syntax

ASSEMBLING RUST WITH X86_64

ASSEMBLING RUST WITH X86_64

- `lea`: The LEA (load effective address) instruction computes the effective address in memory (offset within a segment) of a source operand and places it in a general-purpose register.
- `mov`: The MOV (move) and CMOVcc (conditional move) instructions transfer data between memory and registers or between registers
- `callq`: The CALL instruction allows control transfers to procedures within the current code segment (near call) and in a different code segment (far call).

ASSEMBLING RUST WITH X86_64

- `retq`: Transfers program control to a return address located on the top of the stack.
- `nopl`: No operation.

ASSEMBLING RUST WITH X86_64

Main routine again:

```
<main>:
    push   %rax
    movslq %edi,%rax
    lea    -0xbb(%rip),%rdi
    mov    %rsi,(%rsp)
    mov    %rax,%rsi
    mov    (%rsp),%rdx
    callq  4170 <_ZN3std2rt10lang_start17hc195ea030023a472E>
    pop    %rcx
    retq
    nopl   (%rax)
```

ASSEMBLING RUST WITH X86_64

Lessons learned:

- instructions are stored in memory like data
- one instruction after another is executed
- every byte in main memory have an address
- some instructions depend on each other (*data flow dependency* or *control flow dependency*)
- optimizations are applied
- there is some stack (acc. to the spec)

ASSEMBLER

- a stack supports PUSH and POP operations
- PUSH by incrementing a stack pointer (ESP) and initializing a value
- POP by decrementing ESP and copying a value to desired location

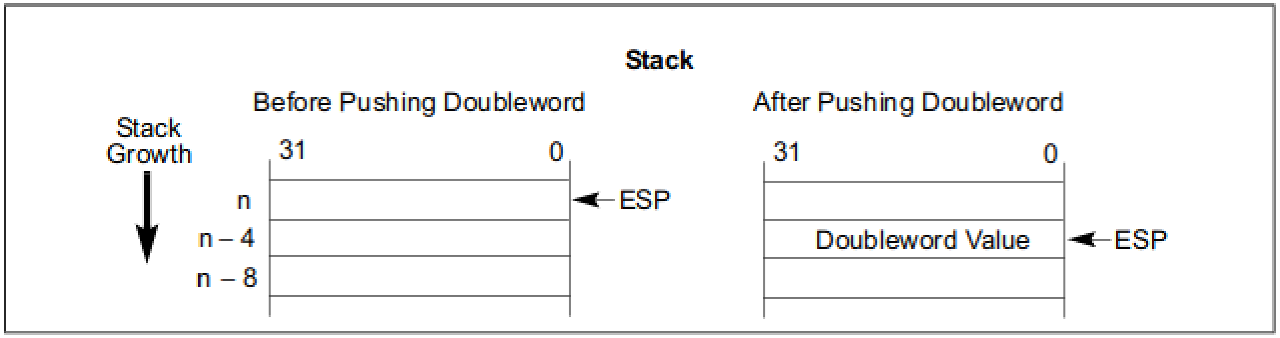


Figure 7-1. Operation of the PUSH Instruction

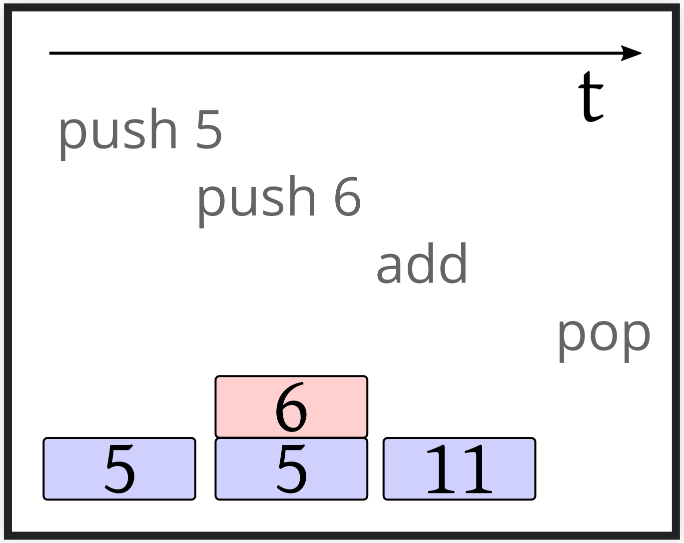
EXAMPLE

Yet another example:

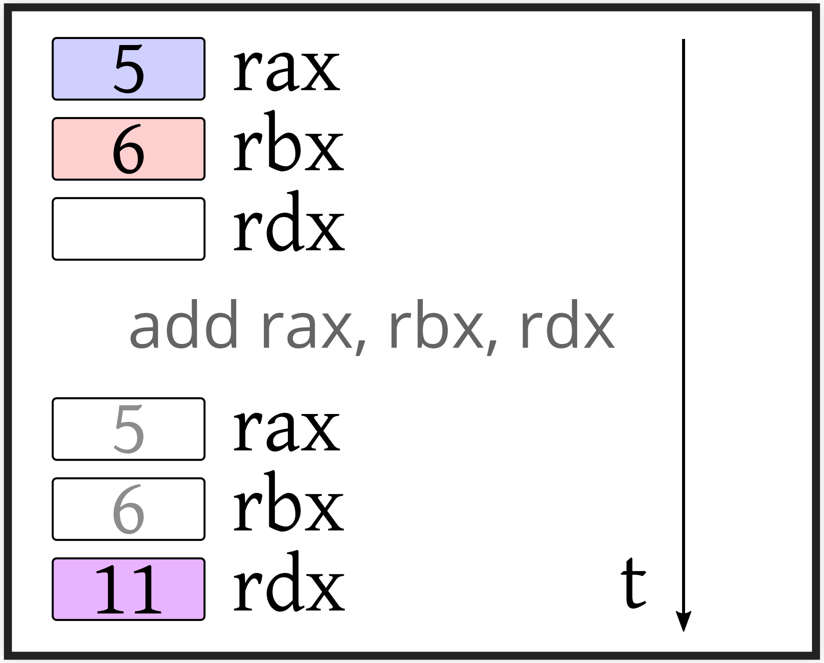
```rust
fn adder(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let _sum = adder(5, 6);
}
```

→ two approaches to execute this

STACK MACHINE



REGISTER MACHINE



WRAP UP

- We have seen a lot of low-level instructions.
- Some use registers. Some use some stack.
- A stack is used to organize the memory.
- We wonder how exectuables are organized and how executables are actually run.

THE TRUTH

ELF § File Structure

.text	where code stands
.data	where global tables, variables, etc. stand
.bss	That's where your uninitialized arrays and variable are
.rodata	that's where your strings go

THE TRUTH

ELF § File Structure

.comment & .note	just comments put there by the compiler/linker toolchain
.stab & .stabstr	debugging symbols & similar information.

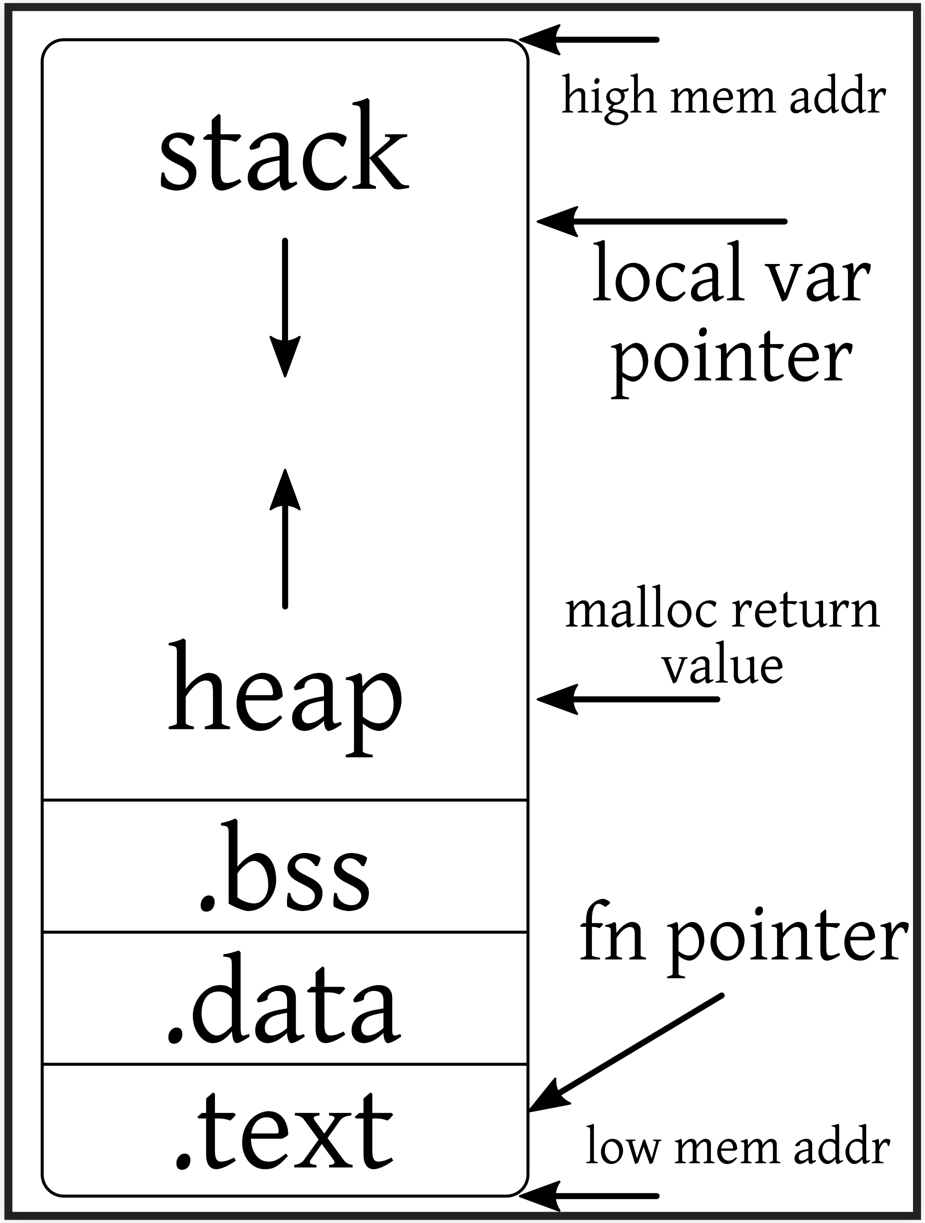
THE TRUTH

```c
#include <stdio.h>
#include <stdlib.h>
const int data = 1;
int global = 1;

void recursive(int count) {
    int local = 42;
    printf("local: %p\n", &local);
    if (count > 0)
        recursive(count - 1);
}

int main() {
    printf("function recursive: %p\n", &recursive);
    printf("data: %p\n", &data);
```

THE TRUTH

```
function recursive: 0x400520
data: 0x400694
global: 0x601038
heap: 0x1a5d670
local: 0x7ffc91466058
local: 0x7ffc91466038
local: 0x7ffc91466018
local: 0x7ffc91465ff8
local: 0x7ffc91465fd8
local: 0x7ffc91465fb8
local: 0x7ffc91465f98
local: 0x7ffc91465f78
local: 0x7ffc91465f58
local: 0x7ffc91465f38
local: 0x7ffc91465f18
```

FTR, $\texttt{0x7fff2de7d148} > 16*1024^3$


high mem addr
stack
local var
pointer
malloc return
value
heap
.bss
fn pointer
.data
.text
low mem addr

THE TRUTH

- So there are different memory sections.
- One is called stack. One is called heap.
- In particular, the stack is used to handle function calls.
- Function calls consist of: arguments, local variables, return addresses, return values

```rust
fn adder(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let _sum = adder(5, 6);
}
```

THE TRUTH (CONCLUSIO)

Storing state:

- We have registers and main memory to store data.
- We need a memory layout maintenance strategy. Stack and heap is one common way.

Implementing function calls:

- Prefer register machines. Sometimes too small. Then stack machines.
- We push and pop: args, locals, return addr and values

THE TRUTH (CONCLUSIO)

Stack:

- fast allocation
- automatically deallocated
- stack overflow possible

Heap:

- slow allocation (malloc, calloc, realloc, …)
- manual deallocation (free)
- memory leaks possible

AN EXPERIMENT

```c
#include <stdio.h>

void func() {
    int a;
    printf("was %d, setting to %d\n", a, 42);
    a = 42;
}

int main() {
    func();
    func();
    return 0;
}
```

AN EXPERIMENT

```
was 0, setting to 42
was 42, setting to 42
```

AN EXPERIMENT

```
was 0, setting to 42
was 42, setting to 42
```

Why is “was 42” set to 42? (→ thing about the stack)

AN EXPERIMENT (IN RUST)

```
fn func() {
    let a;
    println!("was {}, setting to {}\n", a, 42);
    a = 42;
}

fn main() {
    func();
    func();
}
```

AN EXPERIMENT (IN RUST)

```
warning: value assigned to `a` is never read
 --> stack_experiment.rs:4:5
  |
4 |     a = 42;
  |     ^
  |
  = note: `#[warn(unused_assignments)]` on by default
  = help: maybe it is overwritten before being read?

error[E0381]: borrow of possibly uninitialized variable: `a`
 --> stack_experiment.rs:3:41
  |
3 |     println!("was {}, setting to {}\n", a, 42);
  |                                         ^ use of possibly
```

boring 😒

MEMORY SAFETY ISSUES

- There is uninitialized memory.
- Stack has predictable structure. Heap does not.
- Data is code. Code is data.

→ all these are subject to attacks in IT security

We want countermeasures. rust provides more than C.

MEMORY SAFETY: C VERSUS RUST

C:

- implicit [de]allocation on the stack
- malloc/… and free for the heap

rust:

- implicit [de]allocation on the stack with stronger notion of scopes {} and lifetimes
- explicit heap allocation with Box types living in a scope

BOX TYPE EXAMPLE

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

fn main() {
    let stack_pt: Point =
                    Point { x: 10, y: 10 };
    let heap_pt: Box<Point> =
             Box::new(Point { x: 10, y: 10 });
    println!("{:p}", &stack_pt);
    println!("{:p}", &heap_pt);
}
```

CONCEPT 1: REFERENCES

References are memory addresses. Unlike C, there is no pointer arithmetic in safe rust. You can only print memory addresses. Unlike C, rust does automatic dereferencing.

```
fn func() {}

fn main() {
  let f = &func;
  let local1 = 5;
  let local2 = 6;

  println!("func: {:p}", f);             //func: 0x558670c3c570
  println!("local1: {:p}", &local1); //local1: 0x7ffc4f6f18a8
  println!("local2: {:p}", &local2); //local2: 0x7ffc4f6f18ac
}
```

CONCEPT 1: REFERENCES

We copy 5 and 6 as data. Add them. Return them as data. “Call by value”.

There are no references.

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    println!("5 + 6 = {}", add(5, 6));
}
```

CONCEPT 1: REFERENCES

We provide two references (memory addresses) as argument. rust automatically dereferences them for addition.

```rust
fn add(a: &i32, b: &i32) -> i32 {
    return a + b;
}

fn main() {
    println!("5 + 6 = {}", add(&5, &6));
}
```

CONCEPT 1: REFERENCES (IN C)

```c
#include <stdio.h>

int add(int *a, int *b) {
    return *a + *b;
}

int main() {
    int a = 5;
    int b = 6;
    printf("%d\n", add(&a, &b));
    return 0;
}
```

CONCEPT 1: REFERENCES (IN C)

- &5 and &6 do not work in C
- we need to explicitly dereference a pointer/reference
  - might yield a null pointer dereference
  - in rust usually automatically (you can also write `*a + *b`)

Follow-up question: Are references mutable?

CONCEPT 1: REFERENCES

```rust
fn adder(a: &i32, b: &i32) -> i32 {
    a = b;
    a + b
}
```

CONCEPT X: LIFETIMES

```
error[E0623]: lifetime mismatch
 --> test.rs:2:9
  |
1 | fn adder(a: &i32, b: &i32) -> i32 {
  |             ----     ---- these two types are
  |          declared with different lifetimes...
2 |     a = b;
  |         ^ ...but data from `b` flows into `a` here

error: aborting due to previous error
```

CONCEPT X: LIFETIMES

```
error[E0623]: lifetime mismatch
 --> test.rs:2:9
  |
1 | fn adder(a: &i32, b: &i32) -> i32 {
  |             ----     ---- these two types are
  |          declared with different lifetimes...
2 |     a = b;
  |         ^ ...but data from `b` flows into `a` here

error: aborting due to previous error
```

What are lifetimes? → talk in the future.

CONCEPT 1: REFERENCES

References are immutable unless you make it a `&mut` reference.

```
fn adder(a: &mut i32, b: &mut i32) -> i32 {
    a = b;
    a + b
}

fn main() {
    println!("{}", adder(&mut 5, &mut 6));
}
```

CONCEPT 1: REFERENCES

But they have different semantics:

```
error[E0369]: binary operation `+` cannot be applied to type `
 > test.rs:3:7
  |
3 |     a + b
  |     - ^ - &mut i32
  |     |
  |     &mut i32
  |
  = note: an implementation of `std::ops::Add` might be missin

error: aborting due to previous error

For more information about this error, try `rustc --explain E0
```

CONCEPT 1: REFERENCES

You could do:

```
fn adder(a: &mut i32, b: &mut i32) -> i32 {
    a = b;
    *a + *b
}

fn main() {
    println!("{}", adder(&mut 5, &mut 6));
}
```

→ * explicitly dereferences an address

CONCEPT 1: REFERENCES

… but you still run into lifetimes, again.

```
error[E0623]: lifetime mismatch
 --> ref.rs:2:9
  |
1 | fn adder(a: &mut i32, b: &mut i32) -> i32 {
  |             ---------     --------- these two types are decl
2 |     a = b;
  |         ^ ...but data from `b` flows into `a` here

error: aborting due to previous error
```

CONCEPT 1: REFERENCES

… but you still run into lifetimes, again.

```
error[E0623]: lifetime mismatch
 --> ref.rs:2:9
  |
1 | fn adder(a: &mut i32, b: &mut i32) -> i32 {
  |             --------     -------- these two types are decl
2 |     a = b;
  |         ^ ...but data from `b` flows into `a` here

error: aborting due to previous error
```

→ references can be mutable/immutable. Let's forget about the details for now.

CONCEPT 1: REFERENCES

Dereferencing is defined by the type implementation (→ Deref trait). You cannot arbitrarily dereference values.

```rust
fn main() {
    let a = 5;
    let b = 6;

    let mut c = &a;
    c = &b;
    println!("{}", c);
}
```

CONCEPT 1: REFERENCES

And all operations are scope-based.

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

CONCEPT 1: REFERENCES

References to functions are possible …

```
fn fn1() {}
fn fn2() {}

fn main() {
    let mut a = fn1;
    a = fn2;
}
```

CONCEPT 1: REFERENCES

```
error[E0308]: mismatched types
 --> test.rs:6:9
  |
6 |     a = fn2;
  |         ^^^ expected fn item, found a different fn item
  |
  = note: expected type `fn() {fn1}`
             found type `fn() {fn2}`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0
```

… but difficult to get right (→ scoping).

CONCEPT 2: BORROWING

- rust's distinctive memory safety mechanism
- each value in Rust has a variable that’s called its *owner*.
- there can only be one owner at a time.
- when the owner goes out of scope, the value will be dropped.

⇒ “ownership”

CONCEPT 2: BORROWING

b, c and d are aliases of a.

```rust
fn main() {
    let mut a = Vec::new();
    a.push(42);

    let b = &a;
    let c = &a;
    let d = &a;
}
```

CONCEPT 2: BORROWING

b is mutating a.

```
fn main() {
    let mut a = Vec::new();
    a.push(42);

    let b = &mut a;
    b.push(31);
}
```

CONCEPT 2: BORROWING

Borrowing rules:

- one or more references (&T) to a resource,
- exactly one mutable reference (`&mut T`).

Either or, not both!

DATA RACES

> *There is a ‘data race’ when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized.*

Borrowing prevents data races.

CONCEPT 2: BORROWING

A binding that borrows something does not deallocate the resource when it goes out of scope.

```rust
// This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}

// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}
```

CONCEPT 2: BORROWING

```rust
fn main() {
  // Create a boxed i32, and a stacked i32
  let boxed_i32 = Box::new(5_i32);
  let stacked_i32 = 6_i32;

  // Borrow the contents of the box. Ownership is
  // not taken, so the contents can be borrowed again.
  borrow_i32(&boxed_i32);
  borrow_i32(&stacked_i32);
```

CONCEPT 2: BORROWING

```rust
{
    // Take a reference to the data contained
    // inside the box
    let _ref_to_i32: &i32 = &boxed_i32;

    // Error! Can't destroy `boxed_i32` while
    // the inner value is borrowed later in scope.
    eat_box_i32(boxed_i32); // FIXME Comment out

    // Attempt to borrow `_ref_to_i32` after
    // inner value is destroyed
    borrow_i32(_ref_to_i32);
    // `_ref_to_i32` goes out of scope and
    // is no longer borrowed.
}
```

CONCEPT 2: BORROWING

```
  // `boxed_i32` can now give up
  // ownership to `eat_box` and be destroyed
  eat_box_i32(boxed_i32);
}
```

EPILOGUE

QUIZ

Name some CPU architecture.

Which are the (dis)advantages of the stack/heap?

What is a return address?

What is borrowing?

Who defines how to dereference a reference?

QUIZ

Name some CPU architecture.

x86, amd64, ARM, PowerPC

Which are the (dis)advantages of the stack/heap?

What is a return address?

What is borrowing?

Who defines how to dereference a reference?

QUIZ

Name some CPU architecture.

x86, amd64, ARM, PowerPC

Which are the (dis)advantages of the stack/heap?

Stack: speed, no memory leaks; *Heap:* size arbitrary at compile time

What is a return address?

What is borrowing?

Who defines how to dereference a reference?

QUIZ

Name some CPU architecture.

x86, amd64, ARM, PowerPC

Which are the (dis)advantages of the stack/heap?

Stack: speed, no memory leaks; *Heap:* size arbitrary at compile time

What is a return address?

memory location of instruction to execute after function returns

What is borrowing?

Who defines how to dereference a reference?

QUIZ

Name some CPU architecture.

x86, amd64, ARM, PowerPC

Which are the (dis)advantages of the stack/heap?

Stack: speed, no memory leaks; *Heap:* size arbitrary at compile time

What is a return address?

memory location of instruction to execute after function returns

What is borrowing?

Memory safety concept where every value has an owner and the owner can borrow values

Who defines how to dereference a reference?

QUIZ

Name some CPU architecture.

x86, amd64, ARM, PowerPC

Which are the (dis)advantages of the stack/heap?

Stack: speed, no memory leaks; *Heap:* size arbitrary at compile time

What is a return address?

memory location of instruction to execute after function returns

What is borrowing?

Memory safety concept where every value has an owner and the owner can borrow values

Who defines how to dereference a reference?

The type implementation

NEXT SESSION

Wed, 2019/11/27 19:00

Show of hands:

- **Topic 1:** Strings, string types and UTF-8
- **Topic 2:** Type systems and traits
- **Topic 3:** Data structure implementations