Rust programming

Module F: safe and unsafe rust

Unsafe: Learning objectives

- when to reach for `unsafe` code
- reason about undefined behavior
- familiarity with raw pointers in rust
- practical experience with raw pointers, C strings and untagged unions

Content overview

- Why is unsafe needed?
- Undefined behavior and optimizations
- Break
- common types in unsafe code
- examples of unsafe usage

Rust guarantees that references are valid

for any `&T` or `&mut T`

- the address is not `NULL`
- the address is well-aligned for type `T`
- it points into memory belonging to the process

These guarantees make rust memory safe

The borrow checker

The borrow + type checker ensures that these conditions are met

- We want a 100% guarantee that when the compiler says that really means our program is correct
- An analysis that is wrong in 1 out of 100 cases is worthless

The borrow checker is conservative

- "if it is not a hell yes, it’s a no"

fail-proof borrow checker

```
fn borrow_checker<P>(program: P) -> bool {
    false
}
```

all programs that my borrow checker accepts are memory safe!

Hence

There are (many) correct programs that the rust borrow checker does not accept

Hence

There are (many) **useful** programs that the rust borrow checker does not accept

- interacting with other languages (FFI)
- interacting with the OS/hardware
- optimization

Unsafe, morally

In rust "unsafe" means "I, the programmer, am responsible for checking the correctness of this code"

The type and borrow checker are still in fully effect. But we can use types like raw pointers on which the conditions that the type/borrow checker places are less strict.

Unsafe in code

- unsafe blocks: "programmer must check the rules"

```
// BAD!
let reference: &u8 = unsafe {
    let ptr = 0usize as *const u8;

    &*ptr
};
```

- unsafe functions: "programmer must check the preconditions"

```
unsafe fn foobar() {
    ...
}
```

- unsafe impl: "programmer must check impl is valid"

```
unsafe impl Send for MyType {}
```

Undefined Behavior & Optimizations

```
1   // std::hint::unreachable_unchecked
2   pub const unsafe fn unreachable_unchecked() -> !
```

- `unsafe fn`: to call this function, the programmer has to check the preconditions
- returns "never", the type of an infinite loop (diverging computation)

Undefined Behavior & Optimizations

```
if expensive_pure_computation() == 0 {
    println!("hello there");
    unsafe { std::hint::unreachable_unchecked() }
} else {
    different_computation()
}
```

- that print is unreachable if the rest of the branch is unreachable

Undefined Behavior & Optimizations

```
1   if expensive_pure_computation() == 0 {
2       unsafe { std::hint::unreachable_unchecked() }
3   } else {
4       different_computation()
5   }
```

- actually the whole branch is unreachable

Undefined Behavior & Optimizations

```
1   expensive_pure_computation() == 0;
2   different_computation()
```

- actually that whole condition does not need to be computed

Undefined Behavior & Optimizations

```
if expensive_pure_computation() == 0 {
    println!("hello there");
    unsafe { std::hint::unreachable_unchecked() }
} else {
    different_computation()
}
```

becomes just

```
different_computation()
```

- but if the condition turns out to be reachable, behavior is confusing

Undefined Behavior & Optimizations

- misusing `unreachable_unchecked` is very explicit. There are many more subtle ways to introduce UB

```
// BAD!
let reference: &u8 = unsafe {
    let ptr = 0usize as *const u8;

    &*ptr
};
```

- the rust compiler assumes that references are valid, so this snippet contains UB!
- LLVM encodes and exploits assumptions like `nonnull` or `noalias`

```
define internal fastcc void @str.RocStr.reallocate(
    %str.RocStr* noalias nocapture nonnull %arg,
    %str.RocStr* nocapture nonnull readonly align 8 %arg1,
    i64 %arg2
)
```

`transmute`

e.g. bitcast a `i64` into a `f64`:

```
std::mem::transmute::<i64, f64>(42i64)
```

there are still checks! `transmute` errors when the size does not correspond

```
error[E0512]: cannot transmute between types of different sizes, or dependently-sized types
 --> src/main.rs:2:12
  |
2 |   unsafe { std::mem::transmute::<i64, f32>(42i64) };
  |            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: source type: `i64` (64 bits)
  = note: target type: `f32` (32 bits)
```

`transmute`

Only some bit patterns are valid for a type. Creating an invalid bit pattern is UB!

Only `0b0000_0000` and `0b0000_0001` are valid `bool` bit patterns. This code has UB:

```
std::mem::transmute::<u8, bool>(2u8)
```

An `if` statement might be compiled into a jump table

```
const JMP_TABLE: *const u8 = [ 0x1000, 0x1100 ];

// this will fail horribly if `bool_value >= 2`
jmp JMP_TABLE[bool_value as usize];
```

The memory representation of rust values is explicitly undefined! Bitcasting is therefore very unsafe!

So Rust is just as bad as C?

- if memory safety can be broken, how is rust any better than C?

So far

- rust is a systems language: it must provide unrestricted access
  - call code in different languages
  - exploit all capabilities of the OS/hardware
  - optimize
- rust's goals mean restricting access
- unsafe is an escape hatch: great power, but the risk of introducing UB

Part II: common types and examples

Raw Pointers

```
1  let mut x = 0;
2  let y = &mut x as *mut i32;
3  let z = 12;
4
5  unsafe {
6      std::ptr::write(y, z);
7      assert_eq!(std::ptr::read(y), 12);
8  }
```

- raw (mut or const) pointers can alias each other!

NonNull

A `*mut T` that is guaranteed to not be NULL

```rust
use std::ptr::NonNull;

let mut x = 0u32;
let ptr = unsafe { NonNull::new_unchecked(&mut x as *mut _) };

// NEVER DO THIS!!! This is undefined behavior. 
let ptr = unsafe { NonNull::<u32>::new_unchecked(std::ptr::null_mut()) };
```

MaybeUninit

Working with uninitialized memory

```rust
use std::mem::MaybeUninit;

let b: bool = unsafe { MaybeUninit::uninit().assume_init() }; // undefined behavior! 
```

- Useful when working with pointers (which may point to uninitialized data)

```rust
pub const unsafe fn swap<T>(x: *mut T, y: *mut T) {
    // Give ourselves some scratch space to work with.
    // We do not have to worry about drops: `MaybeUninit` does nothing when dropped.
    let mut tmp = MaybeUninit::<T>::uninit();

    // Perform the swap
    // SAFETY: the caller must guarantee that `x` and `y` are
    // valid for writes and properly aligned. `tmp` cannot be
    // overlapping either `x` or `y` because `tmp` was just allocated
    // on the stack as a separate allocated object.
    unsafe {
        std::ptr::copy_nonoverlapping(x, tmp.as_mut_ptr(), 1);
        std::ptr::copy(y, x, 1); // `x` and `y` may overlap
        std::ptr::copy_nonoverlapping(tmp.as_ptr(), y, 1);
    }
}
```

CString

A null-terminated string type

```
use std::ffi::CString;
use libc::strlen;

fn main() {
  let cstring = CString::new("Hello, world!").expect("no NULL bytes");

  // pub unsafe extern "C" fn strlen(cs: *const c_char) -> size_t
  println!("{}", unsafe { strlen(cstring.as_ptr())});
}
```

Examples

- interacting with other languages (FFI)
- interacting with the OS/hardware
- optimization

Using libc functions

```
// pub unsafe extern "C" fn getpid() -> pid_t

use libc;

println!("My pid is {}", unsafe { libc::getpid() });
```

Using libc functions

```
1  // pub fn id() -> u32
2
3  use std::process;
4
5  println!("My pid is {}", process::id());
```

Interacting with the OS

```
unsafe fn execve(&self, argv: &[*const c_char], envp: &[*const c_char]) -> c_int {
    match self {
        // ...

        #[cfg(target_family = "windows")]
        ExecutableFile::OnDisk(_, path) => {
            let path_cstring = CString::new(path.to_str().unwrap()).unwrap();

            libc::execve(path_cstring.as_ptr().cast(), argv.as_ptr(), envp.as_ptr())
        }
    }
}
```

Interacting with the Hardware

Using a SIMD intrinsic

```
#[target_feature(enable = "avx")]
unsafe fn vperilps(mut current: __m128, mask: (i32, i32, i32, i32)) -> __m128 {
    let mask = _mm_set_epi32(mask.3, mask.2, mask.1, mask.0);

    std::arch::asm!(
        "vpermilps {a:y}, {a:y}, {m:y}",
        a = inout(ymm_reg) current,
        m = in(ymm_reg) mask,

    );

    current
}
```

Example: Memory consumption of linked lists

```
enum LinkedList {
    Nil,
    Cons(u64, Box<LinkedList>),
}

use LinkedList::*;

impl LinkedList {
    fn range(range: Range<u64>) -> Self {
        let mut list = Nil;
        for value in range.rev() {
            list = Cons(value, Box::new(list));
        }

        list
    }

    fn sum(&self) -> u64 {
        match self {
            Nil => 0,
            Cons(first, rest) => first + rest.sum(),
        }
    }
}
```

Example: Memory consumption of linked lists

```
enum LinkedList {
    Nil,
    Cons(u64, Box<LinkedList>),
}

// could be represented as

struct LinkedList {
    tag: LinkedListTag,
    payload: LinkedListUnion,
}

enum LinkedListTag {
    Nil = 0,
    Cons = 1,
}

union LinkedListUnion {
    nil: (),
    cons: (u64, std::mem::ManuallyDrop<Box<LinkedList>>),
}
```

Example: Memory consumption of linked lists

- what is the memory layout of this type?

```
struct LinkedList {
    tag: LinkedListTag,
    payload: LinkedListUnion,
}

enum LinkedListTag {
    Nil = 0,
    Cons = 1,
}

union LinkedListUnion {
    nil: (),
    cons: (u64, std::mem::ManuallyDrop<Box<LinkedList>>),
}
```

- field order
- alignment
- size

Example: Memory consumption of linked lists

```
struct LinkedList(*const Node);

struct Node {
    first: u64,
    rest: LinkedList,
}

impl LinkedList {
    fn range(range: Range<u64>) -> Self {
        let mut list = LinkedList(std::ptr::null());
        for value in range.rev() {
            let node = Node { first: value, rest: list };
            list = LinkedList(Box::into_raw(Box::new(node)));
        }

        list
    }

    fn sum(&self) -> u64 {
        if self.0.is_null() { 0 } else {
            let node = unsafe { std::ptr::read(self.0) };
            node.first + node.rest.sum()
        }
    }
}
```

Example: Memory consumption of linked lists

```
struct LinkedList(Option<Box<Node>>);

struct Node {
    first: u64,
    rest: LinkedList,
}

impl LinkedList {
    fn range(range: Range<u64>) -> Self {
        todo!()
    }

    fn sum(&self) -> u64 {
        todo!()
    }
}
```

Question: what is the memory layout of LinkedList. What is the size?

Exercises

- Implement functions for the pointer-based `LinkedList`
- Implement a process forwarding program using `execve`
- Implement a custom `Result` variant that matches a specific memory layout

Summary

- rust is a systems language: it must provide unrestricted access
  - call code in different languages
  - exploit all capabilities of the OS/hardware
  - optimize
- rust's goals mean restricting access
- unsafe is an escape hatch: great power, but the risk of introducing UB
- common types in unsafe code: `*const T`, `*mut T`, `CString`, `MaybeUninit`
- examples of unsafe
  - using the `execve` syscall wrapper
  - using custom simd instructions
  - optimizing a linked list with pointer trickery

END