Safe and Unsafe Rust
PLEASE COMPLETE COURSE EVALS!
CREDITS: MARK MANSI

- Developed first version of these slides
- Graduate student in our department
- Active in Rust development

*If you want to know more, talk to Mark!*
FOUNDATIONS

• What does Rust actually guarantee?
• Introducing unsafe
• Unsaftety and Invariants
• Using Abstraction
GETTING STARTED WITH UNSAFE RUST

- Working with raw pointers
- Allocating and deallocating memory
- Links to further reading
WHAT DOES RUST GUARANTEE?
GOAL: FEW BUGS, FASTER PROGRAMS

- Avoid doing non-sensical or wrong things...
- ... and find out when we do.
- Enable compiler optimizations.
LANGUAGE SPEC

Defines allowed, disallowed, and unspecified behaviors.

- Examples of disallowed:
  - dereference null pointer
  - have a bool that is not true or false
  - access array out of bounds

- Examples of unspecified:
  - In C/C++: \( a = f(b) + g(c) \)
  - which is first: \( f \) or \( g \)?
**UNDEFINED BEHAVIOR (UB)**

*there are no restrictions on the behavior of the program.*

Compilers are not required to diagnose undefined behavior (although many simple situations are diagnosed),

*and the compiled program is not required to do anything meaningful.*
IMPLICATIONS OF UB

- Correct programs don’t invoke UB
- UB can be hard to debug
- Compilers can assume no UB when optimizing
EXAMPLE FROM C++

```cpp
char *p = "I'm a string literal";
p[3] = 'x';
```

ISO C++ forbids mutating string literals (ISO C++ §2.13.4p2)
Deferencing an invalid pointer is forbidden (ISO C §6.5.3.2p4)
SAFETY IN RUST

“Safety” means no UB

- Memory safety
  - e.g. accesses are to valid values only
  - e.g. prohibiting mutable aliasing pointers
- Thread safety
  - e.g. mutable aliasing state
- Enforced by type system
```rust
let x = Vec::new();  // Empty Vec
println!("Out of bounds: {}", x[2]);  // Panic, not UB!

fn foo() -> &usize {
    let x = 3;
    &x  // Return reference to stack variable (allowed in C)
    // Doesn't compile (borrow checker error), not UB!
}
```
UB in (Unsafe) Rust

- Dereferencing null, dangling, or unaligned pointers
- Reading uninitialized memory
- Breaking the pointer aliasing rules
- Producing invalid primitive values:
  - dangling/null references
  - null fn pointers
  - a bool that isn’t true or false
MORE UB IN (UNSAFE) RUST

- Producing invalid primitive values:
  - an undefined enum discriminant
  - a char outside the ranges \([0x0, 0xD7FF]\) and \([0xE000, 0x10FFFF]\)
  - A non-utf8 str
- Unwinding into another language
- Causing a data race
WHAT DOES RUST NOT GUARANTEE?
struct Foo(Option<Arc<Mutex<Foo>>>);

impl Drop for Foo {
    /// Implement a destructor for `Foo`
    fn drop(&mut self) {
        // <do some clean up>
    }
}
```rust
fn do_the_foo_thing() {
    let foo1 = Arc::new(Mutex::new(Foo(None)));
    let foo2 = Arc::new(Mutex::new(Foo(None)));

    // Reference cycle
    foo1.lock().unwrap().0 = Some(Arc::clone(&foo2));
    foo2.lock().unwrap().0 = Some(Arc::clone(&foo1));

    // `foo1` and `foo2` are never dropped!
    // Memory never freed. Foo::drop never called. No UB!
}
```
SAFE RUST CAN STILL...

- Panic ("graceful" crashing)
- Deadlock (two threads both waiting for each other)
- Leak of memory and other resources (never freed back to the system)
- Exit without calling destructors (never clean up)
- Integer overflow (MAX_INT + 1)
A DILEMMA
EXAMPLE

In my program (Rust):

```rust
/// Read from file `fd` into buffer `buf`.
fn read_file(fd: i32, buf: &mut [u8]) {
    let len = buf.len();
    libc::read(fd, buf.as_mut_ptr(), len);
}
```

In libc (C):

```c
ssize_t read(int fd, void *buf, size_t count) {
    // oops bug accidentally overflows `buf`
}
```
RESTORING SAFETY

Compiler error: no unsafe C from safe Rust!

```rust
/// Read from the file descriptor into the buffer.
fn read_file(fd: i32, buf: &mut [u8]) {
    let len = buf.len();
    libc::read(fd, buf.as_mut_ptr(), len); // Compile error!
}
```

Ok, but how do we call C libraries or the OS?
unsafe

- Sometimes need to do something potentially unsafe
  - system calls
  - calls to C/C++ libraries
  - interacting with hardware
  - writing assembly code
  - ...

*Compiler can’t check these: Be careful!*
EXAMPLE

```rust
/// Read from the file descriptor into the buffer.
fn read_file(fd: i32, buf: &mut [u8]) {
    let len = buf.len();
    unsafe {
        libc::read(fd, buf.as_mut_ptr(), len);
    }
}
```

Rust compiles, but C code may do something bad: Be careful!
WHAT DOES unsafe MEAN?
“AUDIT unsafe BLOCKS”

From libstd Vec. Consider `set_len`:

```rust
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl Vec {
    /// Sets the length of the vector to `new_len`.
    pub fn set_len(&mut self, new_len: usize) {
        self.len = new_len;
    }
}
```
fn main() {
    let mut my_vec = Vec::with_capacity(0); // empty vector
    my_vec.set_len(100);

    my_vec[30] = 0; // UB!
}
unsafe fn

impl Vec {
    /// Sets the length of the vector to `new_len`.
    pub unsafe fn set_len(&mut self, new_len: usize) {
        self.len = new_len;
    }
}

Can only be called in an unsafe block!
But why is it possible in the first place?
UB AND INVARIANTS

- **Language Invariant**: something assumed by Rust
  - breaking a language invariant is (by definition) UB
  - e.g. `bool` is always **true** or **false**
  - e.g. all references are valid to dereference
UB AND INVARIANTS

- **Program Invariant**: something that is always true according to the *program spec*
  - e.g. the server must write results to the log before responding to the client
- *In the presence of* **unsafe**, breaking **program invariants** can break lang. invariants, leading to **UB**
UB AND INVARIANTS

```rust
code
pub struct Vec<T> {
    buf: RawVec<T>, // `unsafe` in `RawVec`
    len: usize,
}
```
UB AND INVARIANTS

unsafe: someone promises to uphold invariants!

“Promise” is called a proof obligation.
fn read_file(fd: i32, buf: &mut [u8]) {
    let len = buf.len();

    // `read_file` promises to respect buffer length
    unsafe {
        libc::read(fd, buf.as_mut_ptr(), len);
    }
}

// Caller of `set_len` promises to uphold `Vec` invariants!
pub unsafe fn set_len(&mut self, new_len: usize) {
    self.len = new_len;
}
DIFFERENT USES OF `unsafe`

Whose job to check?

- `unsafe { ... }` blocks
  - Enclosing function is responsible
- `unsafe fn`
  - Caller responsible when calling function
  - Impl. responsible when calling other `unsafe`
- `unsafe trait` and `unsafe impl`
  - Implementor is responsible
HOW TO PLAY WITH FIRE🔥
SAFE ABSTRACTIONS

Idea: Abstraction hides unsafe

- Users of the abstraction have no way to cause UB
- Language features make unsafe parts inaccessible
  - Private struct/enum fields
  - Private modules/types
- Use `unsafe` to expose dangerous interfaces
- Can reason about correctness modularly
**EXAMPLE: Vec**

Using only *safe* methods of Vec, it is *impossible* to cause UB, even though Vec uses *unsafe* internally.

- The safe methods of Vec all uphold invariants.
- Methods that could violate invariants are *unsafe* (e.g. set_len)
**EXAMPLE: READING FILES**

```rust
fn main() -> std::io::Result<()> {
    // Open: call libc and OS. Safely!
    let file = File::open("foo.txt")?
    let mut buf_reader = BufReader::new(file);
    let mut contents = String::new();
    // Read: call libc and OS. Safely!
    buf_reader.read_to_string(&mut contents),ev?
    assert_eq!(contents, "Hello, world!");
    Ok(())
    // Close: call libc and OS. Safely!
}
```

*File, BufReader are safe abstractions that uphold invariants about files, memory, etc.*
CAUTION: FIRE IS HOT
RUST HAS LOTS OF INVARIANTS

- Variance
- Rust ABI
- Memory layout of types
  - Zero-sized types, uninhabited types
  - #[repr(C)] and #[repr(packed)]
- Type-based optimizations
- Reordering, memory coherence, and optimizations
- Many more in the Rustonomicon
PRACTICAL FIRE TWIRLING 101
EXAMPLE: `Vec`

- Caution: will ignore lots of concerns
- Can find real implementation on [GitHub](https://github.com)
FIRST: RAW POINTERS

*const T and *mut T

• Like C pointers
• Not borrow checked, unsafe to dereference
• Utilities in std::ptr
• Helpful tools in libstd
  ▪NonNull
impl Vec

pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

pub struct RawVec<T> {
    ptr: *mut T, // ptr to allocated space
    cap: usize,  // amount of allocated space
}
impl Vec

pub fn new() -> Vec<T> {
    Vec {
        buf: RawVec::new(), // initially, no allocation
        len: 0,
    }
}
impl RawVec

pub fn new() -> Self {
    RawVec {
        ptr: ptr::null_mut(), // null ptr, safe to construct
        cap: 0,
    }
}
impl Vec

pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None // empty vector
    } else {
        unsafe {
            self.len -= 1; // decrement length
            let addr = self.buf.ptr.offset(self.len);

            // raw ptr read at index `val`
            let val = ptr::read(addr);

            Some(val)
        }
    }
}
impl Vec

pub fn push(&mut self, value: T) {
    // Are we out of space?
    if self.len == self.buf.cap {
        self.buf.double(); // alloc more space
    }

    // put the element in the `Vec`
    unsafe {
        // compute address of end of buffer
        let end = self.buf.ptr.offset(self.len);
        ptr::write(end, value); // write data to raw pointer
        self.len += 1; // increase length
    }
}
impl RawVec

pub fn double(&mut self) {
    unsafe {
        let new_cap = self.cap * 2 + 1; // new capacity

        // alloc more memory with system heap allocator
        let res = if self.cap > 0 {
            heap::realloc(NonNull::from(self.ptr).cast(),
            self.cap, new_cap)
        } else {
            heap::alloc(new_cap)
        };
        // ...
    }
}
impl RawVec

pub fn double(&mut self) {
    unsafe {
        // ...
        match res {
            Ok(new_ptr) => {
                // update pointer and capacity
                self.ptr = new_ptr.cast().into();
                self.cap = new_cap;
            }
            Err(AllocErr) => {
                // handle out of memory
                out_of_memory();
            }
        }
    }
}
OTHER unsafe TOOLS

- Type memory layout: `#[repr(...)]`
- Mixed-language projects
  - `extern fn`
  - Strings, variadic fns (e.g. `printf`), extern types
  - `rust-bindgen`
  - `cbindgen`
EXTRA RESOURCES

- The Rustonomicon
- Ralf Jung’s Blog
- Alexis Beingessner
  - Notes on Type Layouts and ABI
  - Only in Rust
  - The Kinds of Implementation-Defined
EXTRA EXTRA RESOURCES

- Various IRLO discussions:
  - UB and uninitialized memory
  - What do “memory safety”/“thread safety” mean?
  - Taming UB in LLVM
- Guide to UB
WHERE WE’VE BEEN
FIRST HALF: HASKELL

- Pure, functional language
- Rich type system
  - Algebraic datatypes
  - Polymorphism and typeclasses
- Monads and effects
SECOND HALF: RUST

- Safe, imperative language
- Ownership: memory management without GC
- Borrowing: control aliasing at all costs
- “Fearless concurrency”
DIFFERENT, YET SIMILAR

- Very strong compile-time checks
  - Haskell: typechecking
  - Rust: ownership and borrowing
- Rich type systems
  - Algebraic datatypes, sums and products
  - Typeclasses and traits
  - Rust: Mutable and immutable references
- Functional (features)
  - Closures, iterators
  - Patterns: map, fold, etc.
CORE LANGUAGES

- Simply typed lambda calculus
  - Model of functional languages
- While language
  - Model of imperative languages
- Process calculus
  - Model of message-passing languages
LANGUAGE DESIGN IS REALLY HARD
WHAT REALLY MATTERS?

- It turns out, a lot
- PL design is still a obscure art
  - Not clear how to teach design
  - Requires wisdom, and a ton of experience
- Graydon Hoare has good *thoughts* on this
  - Original inventor of Rust
  - Also invented Monotone, before Git
CORE TECHNICAL CONCERNS

- Literally “what works”
  - How fast is the code?
  - How fast is the compiler?
  - How well does it scale?
  - How compact is the code?
  - Can we build a lazy language?
TRADEOFFS AND WEIGHTING

• Can’t have the best of all worlds
  ▪ Peak performance
  ▪ Correctness
  ▪ Compilation speed
  ▪ Language complexity
  ▪ ...

• How to balance these tradeoffs?
QUALITY OF IMPLEMENTATION

- Languages involve implementation
  - How many bugs are in the compiler?
  - How quickly are bugs fixed?
  - How many people are working on tooling?
  - How is the effort funded?
  - Where are the engineers coming from?
  - Deliver quality on schedule?
  - How is the project managed and organized?
COGNITIVE LOAD

• PL is a human computer interface
• Computer side is easier to measure
• Human side is very poorly understood
  ▪ How hard is it to work in the language?
  ▪ How predictable/intelligible is the compiler?
  ▪ How hard is it to understand certain features?
  ▪ How much can a person “hold in their head”?
HUMAN/CULTURAL CONTEXT

- Languages are used by humans
  - Which libraries are better?
  - Which libraries are worse/missing?
  - How is the documentation?
- What is this language “for”? Who will want to use it?
  - Often depends on cultural context at the start
What technologies does the language work with?
Many of these are not feasible to change
  - Operating systems
  - Foreign function interface
  - Networking, databases
  - Standards: floating point, unicode, ...
How to adapt to these requirements?
WHAT'S NEXT?
LOTS OF ROOM FOR BETTER LANGUAGES

• PL features take a very long time to mature
  ■ Haskell has been around for 30 years
  ■ Rust is young, but builds on decades of PLs
• A good list of promising features
Most languages don’t have module systems
- Or: just use modules for namespaces
- Mostly: combine modules by “including”

Richer module systems in SML/OCaml
- Decompose code into separate parts

Fancier ways to combine whole program units
- Functions that transform modules
- Select between modules at run time
ERROR HANDLING

- No good solutions known, many not-so-good ones
- Exceptions
  - Who should handle exception?
  - At any moment, could jump to handler
- Return error codes
  - Programmers forget to check
- More philosophically
  - What errors should be caught?
  - What errors should simply cause a crash?
  - What is an error?
EFFECT SYSTEMS

- IO in Haskell: any kind of side-effect
- Effect systems: track specific effects
  - “This function reads a file”
  - “This function sends on network”
  - “This function prints to screen”
- In research languages, but still far to go
REFINEMENT/DEPENDENT TYPE SYSTEMS

- Even fancier type systems
- The dream: use types to encode full spec
  - “This function returns a sorted list”
  - “This function finds the minimum element”
  - “This function correctly compiles C to assembly”
- ... and have the compiler check it for you
- Currently: very hard to use
SESSION TYPES

• Types for communicating processes
  ▪ Closely related to process calculus
• Ensure that sender/receiver on same page
  ▪ Avoid deadlocks, wrong messages, etc.
• Long studied, not yet mature
RICHER PATTERNS

- Pattern matching is nice, once you get used to it
- Currently pretty basic: name different parts of data
- Fancier matching behavior?
  - Match the first non-zero element in list
  - Match the last even number, or fail
COST/RESOURCE ANALYSIS

• Fancier types for time and space
  ▪ Describe how long function takes to run
  ▪ Describe how much space function uses
• Catch space leaks, or rare worst-cases
FORMALIZATION

• Languages are still implemented first
• Later on: people try to formalize (sometimes)
• Time and time again: serious design flaws
  ▪ Compilers don’t correctly compile
  ▪ Ambiguous or unclear desired behavior
  ▪ Type systems that don’t guarantee safety
• Currently: formalization is very expensive
NEW KINDS OF HARDWARE

- Not just programming a CPU anymore
  - GPU, TPU, custom chips, etc.
- How to program these very-different platforms?
  - Would like to write just one program
WHAT ELSE IS IN PL?
IMPLEMENTATION (CS 536/701)

- How to implement languages?
  - How do interpreters and compilers work?
- How to make programs go fast?
  - Compiler optimizations? JITs?
- How to make compilers go fast?
  - Incremental compilation?
- How to implement functional languages?
- How does type checking and type inference work?
VERIFICATION (CS 703/704)

- What can even fancier type systems do?
- How to use automated solvers to verify programs?
  - SMT and Horn solvers?
  - Model checking?
- How to verify imperative programs?
- How to verify program correctness
  - At run time? Contracts and dynamic analyses
  - At compile time? Abstract interpretation
SYNTHESIS (CS 703)

- How to write programs automatically?
- How to guide solvers to find correct programs?
- How to do machine learning on open source code?
How to give a more realistic operational semantics?
  - With a stack, control, etc.
How to model concurrency mathematically?
  - Process calculus, Petri nets, ...
How to model memory on multicore machines?
  - Weak memory models
How to design languages for mathematical proofs?
  - Theorem provers and dependent type theories
How to model programs more mathematically?
  - Denotational semantics
THAT’S ALL, FOLKS: REMEMBER TO DO COURSE EVALS!