Unsafe Rust

CIS 198 Lecture 12

Unsafe Rust

- Rust's safety checks are convenient, but highly conservative.
- Some Rust programs are completely safe, but the compiler won't accept them.
- Rust has "unsafe" parts to allow you to escape some of the compiler's restrictions.
  - Sometimes you need to "just frob some dang bits." -Gankro

Unsafe Rust

- Unsafe Rust is useful for:
  - Talking to other languages (e.g. C)
  - Writing some low-level abstraction that `std` doesn't provide
  - Using particular types that convey no ownership semantics
  - Writing `std` itself!

Unsafe Rust

- All unsafe code must be delimited with the `unsafe` keyword.
- Unsafe code can be wrapped in an `unsafe` block.
- Entire functions can be marked as unsafe to call.
  - Code which calls unsafe functions must be in `unsafe` blocks.

```
unsafe fn foo() { }

fn main() {
    unsafe {
        foo();
    }
}
```

- Traits can be designated unsafe.
  - Thus, their `impl`s must be also marked as unsafe.

```
unsafe trait Sync { }
unsafe impl Sync for Foo { }
```

Unsafe vs. Safe

- "Safe" operations are ones that don't break any of Rust's ownership, type-checking, or memory safety rules.
- These are dangerous, but are still considered "safe" in Rust:
  - Deadlocks
  - Memory leaks
  - Integer overflow
  - Exiting without running destructors

Unsafe vs. Safe

- "Unsafe" operations are ones that *might* have dangerous effects that the Rust compiler cannot guard against by analysis.
- These are bad, and are only possible in `unsafe` Rust:
    - Data races
    - Dereferencing invalid pointers
    - Reading uninitialized memory
    - Creating invalid primitives
    - etc.

Unsafe Rust

- What does `unsafe` let you do?
  - Dereference raw pointers
  - Mutate `static` variables
  - Implement `unsafe` traits
  - Call `unsafe` functions
- That's it!
- `unsafe` does not disable the borrow checker or otherwise let you do anything that would break regular Rust semantics.
- Everything that's normally safe is still safe in an `unsafe` context.
- Everything else that the compiler would reject is still disallowed.

Where Is `unsafe` Used?

- All over the place!
- Many, many types in `std` are built on `unsafe` in some way.
  - `Vec`
  - `Cell`/`RefCell`
  - `Box`
  - `Rc`/`Arc`
  - `Send`/`Sync`
  - `HashMap`
  - Lots and lots of other types

Correctness of Unsafe Code

- If `std` is built on `unsafe`, how can it possibly be correct?
- `unsafe` is used in very careful ways within `std`.
- There's no formalization of `std`'s safety, so you pretty much need to just trust that `std` is implemented correctly.
  - Rustbelt is an ongoing project to formally verify the encapsulation of unsafe code. Seeking PhD students!

Correctness of Unsafe Code

- As it turns out, using `unsafe` pollutes the safety of everything that can directly modify data that the `unsafe` code depends on.
  - And if not written correctly, safe functions which use `unsafe` will pollute the safety of any code which uses it.
- For example, take this (simplified) definition of `Vec`.
  - `ptr` is a pointer to the data the `Vec` owns.
  - `cap` is the `Vec`'s capacity, `len` is its length.
  - `len` should never be greater than `cap`, but this is not enforced.

```
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

Correctness of Unsafe Code

- Now consider this new method on Vec, written in totally safe code.
- Despite this code's total safety, it does a dangerous thing.
  - The length of the Vec is now potentially longer than its capacity.
- This method allows clients of Vec to access uninitialized memory!

```
impl<T> Vec<T> {
    // ...
    pub fn evil(&mut self) {
        self.len += 2;
    }
}
```

[1] Example taken from The Scope of Unsafe

Borrow Splitting

- Mutual exclusivity of mutable references is safe, but can be very limiting.
- Struct borrows can be split to allow multiple mutable borrows, as the borrow checker understands structs well enough:

```
struct Point {
    x: i32,
    y: i32,
}

let mut p = Point { x: 0, y: 1 };
let x1 = &mut p.x;
let y1 = &mut p.y;
*x1 += 1;
*y1 += 1; // OK
```

Borrow Splitting

- However, arrays and slices cause problems.
- While the borrow checker *could* understand this, more complex examples with trees, etc., are unfeasible.

```rust
let mut arr = [1, 2, 3];
let x = &mut arr[0];
let y = &mut arr[1];
//  ^ cannot borrow `arr[..]` as mutable more than once at a time
let z = &arr[2];
```

Borrow Splitting

- Unsafety can allow us to carefully overcome this limitation.
- Slices can be split into parts and subsequently lent out.
- `std` provides a function `split_at_mut`, that looks like this:

```
fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();
    assert!(mid <= len);
    unsafe {
        (from_raw_parts_mut(ptr, mid),
         from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

- This is a classic example of how to use unsafe.
    - *We* know this function works safely, but the borrow checker can't validate this property.
    - Unsafe code can be used as a careful escape hatch.

Raw Pointers

- Rust defines two types of C-like "raw pointers":
  - `*const T`, an immutable pointer to a `T`
  - `*mut T`, a mutable pointer to a `T`
- Raw pointers are unsafe to dereference.
- Raw pointers confer no ownership semantics, just like C pointers.
- Raw pointers may be initialized to null with `null()` and `null_mut()`, respectively.
  - These are actually safe - only dereferencing is unsafe.

Raw Pointers

- Raw pointers may be (safely) created in several ways:

```
// Reference coercion
let mut x = 0i32;
let const_ptr = &x as *const i32;
let mut_ptr = &mut x as *mut i32; // Allowed to alias pointers

let box_y = Box::new(1);
let raw_y = &*box_y as *const i32; // Does not consume `box_y`
```

Raw Pointers

- Raw pointers may be (safely) created in several ways:

```
// Box consumption
let raw_y_2 = Box::into_raw(box_y);

// ... to properly clean up, later ...
unsafe { drop(Box::from_raw(raw_y_2)); }
```

- Raw pointers may need to be deallocated manually.
  - If you've taken ownership of the pointer, not if you've aliased it.
- Raw pointers *can*, but should not, be used after their lifetime expires.
  - Doing so will cause a segmentation fault.

Unique

- A wrapper around `*mut T` to indicate that the `Unique` struct *owns* the pointer.
- Confers regular Rust ownership semantics, unlike `*mut T`.
- Implies `T` should not be modified without a unique path to `T`.
- Unsafe to create, dereference, etc.
- Useful for building abstractions such as `Box`.
- Currently unstable :(
    - Available on nightly behind a feature gate:

```rust
#![feature(unique)]
use std::ptr::Unique;

let mut x = 0i32;
let mut_ptr = &mut x as *mut i32;
unsafe {
    let unique_ptr = Unique::new(mut_ptr);
}
```

`Shared`

- A wrapper around `*mut T` to indicate that the `Shared` struct has *shared ownership* of the pointer.
- Confers ownership semantics, unlike `*mut T`.
- Unsafe to create, dereference, etc.
- Useful for building abstractions such as `Rc`, `Arc`.
- Also currently unstable :(

`UnsafeCell`

- A wrapper around a `T` that provides interior mutability.
- Implies that `T` may be modified using `unsafe` operations.
- A primitive cell type that is the basis for `Cell`/`RefCell`.

Uninitialized Memory

- If you really, *really* want some uninitialized memory, you can make some with `std::mem::uninitialized<T>()`.
    - This function just pretends to make a value of type `T`.
    - This function is unsafe to invoke.
- **Reading uninitialized memory always has undefined behaviour!**
- Writing to uninitialized memory may also be undefined.
    - The compiler believes the value is initialized, so it may try to drop a value that isn't there, causing a panic.
- The only safe way to initialize uninitialized memory is by using one of the functions from `std::ptr`.
    - `write`, `copy`, or `copy_nonoverlapping`

Leaking Memory

- Leaking memory is a safe operation, and can be done with `std::mem::forget()`.
- You rarely ever want to do this, but you might want to if:
  - you have an uninitialized value that you don't want to get dropped
  - you have two copies of a value, but only want to drop one to avoid a double free
  - you want to transfer a resource across the FFI boundary and into another language
- This function is safe because Rust does not guarantee that destructors are always run.

Transmutation

- Is the type system just too... pesky for you?
- Want to turn any type into any other type as much as you please?
- `mem::transmute` may be right for you![1]

[1]Ask your compiler before using `mem::transmute`. Side effects may include your laundry being eaten by nasal goblins.

Transmutation

- `mem::transmute<T, U>` takes a value of type `T` and reinterprets it to be of type `U`.
- The only restriction on this function is that `T` and `U` must be the same size.
- Arbitrary type transmutation can have wildly undefined behaviour!

Transmutation

- Creating an instance of any type with invalid state causes arbitrary behaviour.
  - e.g. a `Vec` with a capacity smaller than its length.
  - The above may or may not be possible with an actual `Vec`, it's just an example.
- `transmute` can create invalid primitives, e.g. a `bool` that is neither `0` nor `1` underneath.
- `transmute` can turn an `&T` into an `&mut T`, which is *wildly, always* undefined
- `transmute` on a reference without an explicit lifetime produces an *unbounded* lifetime.

Transmutation

- There's also a variant named `transmute_copy` which is *even more unsafe!*
- `transmute_copy<T, U>` copies `size_of<U>` bytes from an `&T` and interprets them as a `U`.
- No more pesky size check like in regular `transmute`.
- `U` may be larger than `T`, which causes undefined behaviour.

Unbounded Lifetimes

- Unsafe code often produces a reference or lifetime out of nowhere.
    - These lifetimes are unbounded.
- Commonly, these are produced from dereferencing a raw pointer.
- Unbounded lifetimes expand to become as large as they need to be.
- This may produce lifetimes more powerful than `'static`.
- Generally, you can think of these as `'static`.
    - Almost no references are actually `'static`, though.

Implementing `Vec`

- Let's take a deep dive into how `std::vec::Vec` is actually implemented.
- This code will only compile on nightly, since it uses unstable features.
- Warning: this is going to get pretty advanced.

[1]All content taken from [The Rustonomicon]

Vec Layout

- A Vec has three parts:
  - a pointer to the allocated data
  - the size of the allocation
  - the number of elements in the vector
- Naively, this translates into this struct:

```rust
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

- Simple, right?

`Vec` Layout

- Sadly, this won't quite work.
- Some lifetime variance problems:
  - An `&Vec<&'static str>` can't be used in place of an `&Vec<&'a str>`.
- Some drop checker problems:
  - `*mut T` conveys no ownership.
  - The drop checker assumes we own no values of type `T`.

`Vec` Layout

- Using a `Unique` in place of an `*mut T` solves these problems!
  - `Unique` conveys ownership.
  - It also lets us be `Send` & `Sync` if `T` is, and auto-derefs to `*mut T`.
  - It also is guaranteed to never be null, allowing null pointer optimizations.

```rust
#![feature(unique)]
use std::ptr::{Unique, self};

pub struct Vec<T> {
    ptr: Unique<T>,
    cap: usize,
    len: usize,
}
```

`Vec` Allocation

- Now that we use a `Unique` to store our data, what does an empty `Vec` look like?
    - It can't have a null pointer...
    - ...but we don't want to allocate any data!
- Turns out, we can just fill the `Vec` with garbage! 🚮
- `cap` is 0, so we don't need to worry about accidentally accessing garbage.
- `std` exposes a value as `alloc::heap::EMPTY` to represent this value.

Vec Allocation

```
#![feature(alloc, heap_api)]
use alloc::heap::EMPTY;

impl<T> Vec<T> {
    fn new() -> Self {
        unsafe {
            assert!(mem::size_of::<T>() != 0);
            Vec { ptr: Unique::new(heap::EMPTY as *mut _),
                  len: 0,
                  cap: 0 }
        }
    }
}
```

`Vec` Allocation

- Now that we can allocate no space, we need to figure out how to allocate space.
- The `heap` module is our friend here.
  - This lets us talk to Rust's allocator (jemalloc by default).
- We also need to figure out how to handle an out-of-memory condition.
  - `panic!` is no good, since it can cause allocations.
  - `std` usually executes an illegal instruction to crash the program.
- We'll define an out-of-memory error like so:

```
fn oom() { ::std::process::exit(-9999); }
```

`Vec` Allocation

- Our logic for growing the `Vec` is roughly this:

```
if cap == 0 {
    allocate()
    cap = 1
} else {
    reallocate()
    cap *= 2
}
```

- Unfortunately, this is not so easy in practice...

`Vec` Allocation

- Because of some serious LLVM shenanigans, a `Vec` can only contain `isize::MAX` elements.
    - This *also* means we only care about byte-sized allocations, since e.g. `isize::MAX u16s` will only just fit in memory.
- Zero-size types are also tricky to allocate due to LLVM.
- tl;dr Read the ch. 10 in the Rustonomicon, this info is totally out of scope.

`Vec` Allocation

- Time to actually allocate data.
- The `heap` module exposes an `allocate` function that puts data into the heap.
- It also exposes a `reallocate` function that takes an existing allocation and resizes it.
- Our pseudocode from a few slides ago can now be implemented, with a few extra parts.

Vec Allocation

```
fn grow(&mut self) {
    unsafe {
        let align = mem::align_of::<T>();
        let elem_size = mem::size_of::<T>();
        let (new_cap, ptr) = if self.cap == 0 {
            let ptr = heap::allocate(elem_size, align);
            (1, ptr)
        } else {
            let new_cap = self.cap * 2;
            let old_num_bytes = self.cap * elem_size;
            assert!(old_num_bytes <=
                               (::std::isize::MAX as usize) / 2);
            let new_num_bytes = old_num_bytes * 2;
            let ptr = heap::reallocate(*self.ptr as *mut _,
                                old_num_bytes, new_num_bytes, align);
            (new_cap, ptr)
        }
    };

    if ptr.is_null() { oom(); }
    self.ptr = Unique::new(ptr as *mut _);
    self.cap = new_cap;
}
```

`Vec` Push & Pop

- Actual functionality! Woo!
- `push` only needs to check if the `Vec` is full to grow, write to the next available index, and increment the length.
  - Be careful! This method shouldn't read from the memory it writes to, since this is uninitialized.
  - Even calling `v[idx] = x` will try to drop the old value of `v[idx]`.
- To avoid reading uninitialized memory, use `ptr::write`.

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }
    unsafe {
        ptr::write(self.ptr.offset(self.len as isize), elem);
    }

    self.len += 1;
}
```

Vec Push & Pop

- With `pop`, the data being removed is initialized, so we can read it.
- Unfortunately, just moving the value out doesn't work.
  - This would leave memory uninitialized.
- `ptr::read` does the trick here, copying the data out.

```
fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.ptr.offset(self.len as isize)))
        }
    }
}
```

`Vec` Deallocation

- Now we want to be able to deallocate a `Vec`.
- This actually requires a manual implementation of `Drop`.
- The whole `Vec` can be deallocated by popping it down to zero, then calling `heap::deallocate`.
- Obviously, we shouldn't try to deallocate anything if the `Vec` has no memory allocated.

Vec Deallocation

```
impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            while let Some(_) = self.pop {}
            let align = mem::align_of::<T>();
            let elem_size = me::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes,
                                                      align);
            }
        }
    }
}
```

`Vec` `Deref`

- How do you implement`Deref` & `DerefMut` for `Vec`?
- For `Vec<T>`, these methods return an `&[T]`/`&mut [T]`, respectively.
- Basically, this boils down to calling `::std::slice::from_raw_parts(*self.ptr, self.len)`.

`Vec` Insert & Remove

- Inserting into a `Vec` needs to shift all elements over to the right from the given index.
- Fortunately, we have an easy way to do this using `ptr::copy`, which is like `memmove` from C.
- `ptr::copy` copies some chunk of memory from here to there, and the source and destination may overlap.
- Inserting at index `i` shifts `[i .. len]` to `[i+1 .. len+1]`, using the old `len`.

Vec Insert & Remove

```
pub fn insert(&mut self, index: usize, elem: T) {
    assert!(index <= self.len);
    if self.cap == self.len { self.grow(); }

    unsafe {
        if index < self.len {
            ptr::copy(self.ptr.offset(index as isize),
                      self.ptr.offset(index as isize + 1),
                      self.len - index);
        }
        ptr::write(self.ptr.offset(index as isize), elem);
        self.len += 1;
    }
}
```

`Vec` Insert & Remove

- Removing elements by index just behaves in the opposite direction.
- Shift all elements from `[i+1 .. len+1]` to `[i .. len]` using the *new* `len`.

Vec Insert & Remove

```
pub fn remove(&mut self, index: usize) -> T {
    assert!(index < self.len);
    unsafe {
        self.len -= 1;
        let t = ptr::read(self.ptr.offset(index as usize));
        ptr::copy(self.ptr.offset(index as isize + 1),
                  self.ptr.offset(index as isize),
                  self.len - index);
        t
    }
}
```

Rust FFI

- FFI: Foreign Function Interface.
- FFI means calling one language from another.
- For us, this means:
  - Rust calling C
  - C calling Rust
  - Other languages (e.g. Python/Ruby) calling Rust
- Why C? C is the de facto language used for FFI.
  - Often referred to as the programming *lingua franca*.

Calling C from Rust

- Why?
  - Sometimes you need to call another library - e.g. OpenSSL - that would be too costly to reimplement in Rust.
  - Sometimes you need to interface with a particular language; many native programming languages can make C bindings.
- Calling foreign functions from Rust is `unsafe`!
  - Because, of course, C is unsafe.

Calling C from Rust

- Compile C to static libraries (`.a`/`.lib`).
  - `cc -c -o foo.o foo.c`
  - `ar rcs libfoo.a foo.o`
- Or to dynamic libraries (`.so`/`.dylib`/`.dll`).
  - `cc -c -fPIC -o foo.o foo.c`
  - `cc -shared -fPIC -o libfoo.so foo.o`

Calling C from Rust

- In C:

```c
int32_t foo() { return 10; }
```

- In Rust:

```rust
#[link(name = "foo", kind = "static")]  // links libfoo.a.
extern {
    fn foo() -> i32;
}

#[link(name = "foo")]  // links libfoo.so.
extern {               // By default, this is also
    fn foo() -> i32;   // statically linked.
}
```

- Calling foreign functions is unsafe:

```rust
fn main() {
    println!("foo: {}", unsafe { foo() });
}
```

Calling C from Rust

- In C APIs, it's common to use incomplete struct definitions to define types whose implementation shouldn't be exposed to users.
  - These can't be instantiated by the user.

```
struct OpaqueThing;
```

- To represent a type like this in Rust (for FFI), we want to make a type that we can't instantiate! So we do this:

```
enum OpaqueThing { }
```

- Now, we can't have one of these, but we *can* have a pointer to one: `*mut OpaqueThing`.
  - Pointers to opaque types are very common in C interfaces.

Calling Rust from C

- Why?
  - Writing particularly dubiously-safe code in Rust.
  - Writing any part of a C program in a nicer language.

Rust from C: `#[repr(C)]`, `extern "C"`

- Rust has its own rules about memory layout and calling convention, which are different from C's.
- In order to call from C (**or any other language!**), we have to use C rules. (C doesn't have generics, enums with fields, etc.)

```rust
#[repr(C)]
pub enum Color { Red = 1, Blue, Green = 5, Yellow }

#[repr(C)]
pub struct Bikeshed { height: f64, area: f64 }

extern "C" pub fn paint(bs: Bikeshed, c: Color) { /* ... */ }
```

- Use opaque structs to hide stuff that C can't use:

```rust
struct X<T>(T); // C doesn't have generics.

#[repr(C)]
pub struct Xi32(X<i32>); // This struct hides the type parameter.
```

Rust from C: Static Linking

- Compile Rust to static libraries (`.a`) and link at build time.
  - Compile to `target/release/libfoo.a`.
- `Cargo.toml`:

```
[lib]
crate-type = ["staticlib"]
```

- $ `cc -Ltarget/release -lfoo -o main main.c`

```
#include <stdint.h>

int32_t foo(); // Function prototype for Rust function

int main() {
    printf("foo() -> %d\n", foo());
}
```

Rust from C: Dynamic Linking

- Compile Rust to dynamic libraries (`.so`) and load at runtime.
  - Compile to `target/release/libfoo.so`.
- `Cargo.toml`:

```
[lib]
crate-type = ["dylib"]
```

- In C:

```
#include <dlfcn.h>

int main() {
    void *handle = dlopen("target/release/libfoo.so", RTLD_LAZY);
    int32_t (*foo)() = dlsym(handle, "foo"); // get function ptr
    // plus error checking

    printf("foo() -> %d\n", foo());

    dlclose(handle);
}
```

Calling Rust from (e.g.) Python

- Why?
    - Many languages (Python, Ruby, etc.) are extremely slow compared with C and Rust.
    - Traditionally, high-performance functions are written in C or C++, and called from scripting languages.
    - Rust is an ideal alternative high-performance language.
    - Great for data processing, scientific computing, multithreaded code, etc.

Rust from Python

- Compile Rust to a dynamic library (`.so`/`.dylib`/`.dll`).
- In Python:

```
import ctypes
libfoo = ctypes.CDLL("target/release/libfoo.so")
print("foo() -> {}".format(libfoo.foo()))
```

Rust from Ruby

- Compile Rust to a dynamic library (`.so`/`.dylib`/`.dll`).
- In Ruby:

```ruby
require 'fiddle'
require 'fiddle/import'

module Foo
  extend Fiddle::Importer
  dlload 'target/release/libfoo.so'
  extern 'int foo()'
end

puts Foo.foo()
```

rust-bindgen

- Generates Rust bindings for a C header.
    - Install via `cargo install bindgen`

```
typedef char my_bool;
typedef struct st_mysql_field { char *name; /* */ } MYSQL_FIELD;
my_bool STDCALL mysql_thread_init(void);
```

- $ `bindgen -l mysql -match mysql.h -o mysql.rs /usr/include/mysql/mysql.h`

```
pub type my_bool = ::std::os::raw::c_char;

#[repr(C)] #[derive(Copy)] pub struct Struct_st_mysql_field {
    pub name: *mut ::std::os::raw::c_char, /* */ }
impl ::std::clone::Clone for Struct_st_mysql_field { /* */ }
impl ::std::default::Default for Struct_st_mysql_field { /* */ }
pub type MYSQL_FIELD = Struct_st_mysql_field;

#[link(name = "mysql")]
extern "C" { pub fn mysql_thread_init() -> my_bool; }
```

rusty-cheddar

- Generates C bindings (header) for Rust code.
  - Generation is done in the `build.rs` script.
- Only C-like enums (no fields).
- No diverging (`-> !`) functions.

[1] disclaimer: we haven't tried this.