



unsafe Rust

Not Quite C

Jack Duvall & Cooper Pierce

Carnegie Mellon University



1 `unsafe` Features

2 Type Sizing

3 FFI

- With C
 - `bindgen`
- With C++
 - `cxx`

What is `unsafe`?

Rust, without *some* compiler checks

What is `unsafe`?

Rust, without *some* compiler checks

- We can't check lifetime validity for raw pointers

What is `unsafe`?

Rust, without *some* compiler checks

- We can't check lifetime validity for raw pointers
- We might use memory or type-unsafe compiler-exposed functions

What is `unsafe`?

Rust, without *some* compiler checks

- We can't check lifetime validity for raw pointers
- We might use memory or type-unsafe compiler-exposed functions
- We might have uninitialised memory

What is `unsafe`?

Rust, without *some* compiler checks

- We can't check lifetime validity for raw pointers
- We might use memory or type-unsafe compiler-exposed functions
- We might have uninitialised memory
- We might have extra, uncheckable requirements to guaranteed soundness

Recall: Safety Guarantees

Using `unsafe` correctly **won't** excuse us from any of the rules in “regular” Rust:

Recall: Safety Guarantees

Using `unsafe` correctly **won't** excuse us from any of the rules in “regular” Rust:

- Values have one owner

Recall: Safety Guarantees

Using `unsafe` correctly **won't** excuse us from any of the rules in “regular” Rust:

- Values have one owner
- References are valid (i.e., don't dangle, correct lifetime, etc...)

Recall: Safety Guarantees

Using `unsafe` correctly **won't** excuse us from any of the rules in “regular” Rust:

- Values have one owner
- References are valid (i.e., don't dangle, correct lifetime, etc...)
- Mutable (exclusive) references are exclusive

Recall: Safety Guarantees

Using `unsafe` correctly **won't** excuse us from any of the rules in “regular” Rust:

- Values have one owner
- References are valid (i.e., don't dangle, correct lifetime, etc...)
- Mutable (exclusive) references are exclusive
- Values of a type have a memory layout consistent with that type, and aren't in some invalid state

Recall: Safety Guarantees

Using `unsafe` correctly **won't** excuse us from any of the rules in “regular” Rust:

- Values have one owner
- References are valid (i.e., don't dangle, correct lifetime, etc...)
- Mutable (exclusive) references are exclusive
- Values of a type have a memory layout consistent with that type, and aren't in some invalid state

`unsafe` may also have additional requirements to uphold if we're calling an `unsafe` function or implementing an `unsafe` trait, so that other code relying on some behaviour down the line isn't unsound.

Soundness

Because we're using the word `unsafe` to mean code that has access to some extra abilities (more on these in a second), it will be useful to have another term which means our code in `unsafe` blocks is correct.

We'll say code is *sound* if it cannot cause undefined behaviour, and *unsound* otherwise—regardless of whether or not the failure point is in the `unsafe` section or not.

Undefined Behaviour in Rust

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.
- Causing a data race

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.
- Causing a data race
- Executing code compiled with target features that the current thread of execution does not support

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.
- Causing a data race
- Executing code compiled with target features that the current thread of execution does not support
- Producing invalid values (either alone or as a field of a compound type such as enum/struct/array/tuple): see <https://doc.rust-lang.org/reference/behavior-considered-undefined.html>

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.
- Causing a data race
- Executing code compiled with target features that the current thread of execution does not support
- Producing invalid values (either alone or as a field of a compound type such as enum/struct/array/tuple): see <https://doc.rust-lang.org/reference/behavior-considered-undefined.html>
- Violating the <https://doc.rust-lang.org/reference/inline-assembly.html#rules-for-inline-assembly> for inline assembly.

Undefined Behaviour in Rust

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.
- Causing a data race
- Executing code compiled with target features that the current thread of execution does not support
- Producing invalid values (either alone or as a field of a compound type such as enum/struct/array/tuple): see <https://doc.rust-lang.org/reference/behavior-considered-undefined.html>
- Violating the <https://doc.rust-lang.org/reference/inline-assembly.html#rules-for-inline-assembly> for inline assembly.

This list isn't exhaustive and some items are still up for debate, but all in all a lot less than C.

unsafe Powers

So what abilities does `unsafe` grant us?

In `unsafe` code we can:

unsafe Powers

So what abilities does `unsafe` grant us?

In `unsafe` code we can:

- Dereference a raw pointer

unsafe Powers

So what abilities does `unsafe` grant us?

In `unsafe` code we can:

- Dereference a raw pointer
- Call an `unsafe` function

unsafe Powers

So what abilities does `unsafe` grant us?

In `unsafe` code we can:

- Dereference a raw pointer
- Call an `unsafe` function
- Implement an `unsafe` trait

unsafe Powers

So what abilities does `unsafe` grant us?

In `unsafe` code we can:

- Dereference a raw pointer
- Call an `unsafe` function
- Implement an `unsafe` trait
- Access fields in a `union`

unsafe Powers

So what abilities does `unsafe` grant us?

In `unsafe` code we can:

- Dereference a raw pointer
- Call an `unsafe` function
- Implement an `unsafe` trait
- Access fields in a `union`
- Reading from or writing to a `static mut` variable

unsafe Powers

So what abilities does `unsafe` grant us?

In `unsafe` code we can:

- Dereference a raw pointer
- Call an `unsafe` function
- Implement an `unsafe` trait
- Access fields in a `union`
- Reading from or writing to a `static mut` variable

What might be some use cases for these?

Rust Has Pointers?

There are two pointer types in Rust:

- `*mut T`
- `*const T`

Raw pointers have less guarantees than other types:

Rust Has Pointers?

There are two pointer types in Rust:

- `*mut T`
- `*const T`

Raw pointers have less guarantees than other types:

- aren't checked by the borrow checker (cf. references)

Rust Has Pointers?

There are two pointer types in Rust:

- `*mut T`
- `*const T`

Raw pointers have less guarantees than other types:

- aren't checked by the borrow checker (cf. references)
- they aren't guaranteed to point to valid memory (cf. references)

Rust Has Pointers?

There are two pointer types in Rust:

- `*mut T`
- `*const T`

Raw pointers have less guarantees than other types:

- aren't checked by the borrow checker (cf. references)
- they aren't guaranteed to point to valid memory (cf. references)
- they aren't guaranteed to be aligned (cf. references)

Rust Has Pointers?

There are two pointer types in Rust:

- `*mut T`
- `*const T`

Raw pointers have less guarantees than other types:

- aren't checked by the borrow checker (cf. references)
- they aren't guaranteed to point to valid memory (cf. references)
- they aren't guaranteed to be aligned (cf. references)
- don't handle cleaning up the underlying resource (cf. owned values)

Using Raw Pointers

We we use a raw pointer we have to guarantee it is:

Using Raw Pointers

We use a raw pointer we have to guarantee it is:

- non-null

Using Raw Pointers

We we use a raw pointer we have to guarantee it is:

- non-null
- aligned

Using Raw Pointers

We use a raw pointer we have to guarantee it is:

- non-null
- aligned
- the read would be entirely contained within one allocation

Using Raw Pointers

We use a raw pointer we have to guarantee it is:

- non-null
- aligned
- the read would be entirely contained within one allocation
- and some other rules: see <https://doc.rust-lang.org/std/ptr/index.html>

Using Raw Pointers

We use a raw pointer we have to guarantee it is:

- non-null
- aligned
- the read would be entirely contained within one allocation
- and some other rules: see <https://doc.rust-lang.org/std/ptr/index.html>

```
let mut x = 42;
let x_ptr = &mut x as *mut i32;

unsafe {
    *x_ptr += 27;
}

assert_eq!(x, 69);
```

Another Example

```
let address = 0x012345usize;  
let r = address as *const i32;  
  
// Oh boy, now we can read arbitrary memory  
unsafe {  
    println!("{}", *r);  
}
```

Another Example

```
let address = 0x012345usize;
let r = address as *const i32;

// Oh boy, now we can read arbitrary memory
unsafe {
    println!("{}", *r);
}
```

What does Miri have to say about this?

<https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=e288775bda449a2edcaece3cc1e24211>

Rust Has unions?

Like C, Rust has `union` types, mostly for FFI.

```
#[repr(C)]
enum ValKind { Int, Pointer }

#[repr(C)]
union ValContents {
    i: i32,
    p: *const std::ffi::c_void,
}

#[repr(C)]
struct Value {
    kind: ValKind,
    payload: ValContents,
}
```

Using a `union`

```
fn is_zero(Value { kind, payload } : Value) -> bool {  
    unsafe {  
        match kind {  
            ValKind::Int      => payload.i == 0,  
            ValKind::Pointer => payload.p.is_null(),  
        }  
    }  
}
```

Rust Has `static` Variables?

Slightly different from `const` variables, which we haven't talked about much:

- actually corresponds to a location in the program
- can take a reference to it
- must be `Send` (shareable across threads)
- can read from any non-mutable `static` variable without `unsafe` iff it is `Sync`

Rust Has `static` Variables?

Slightly different from `const` variables, which we haven't talked about much:

- actually corresponds to a location in the program
- can take a reference to it
- must be `Send` (shareable across threads)
- can read from any non-mutable `static` variable without `unsafe` iff it is `Sync`

```
static VAR1: &'static str = "Hello";
static mut VAR2: &'static str = " World";

fn main() {
    println!("{}", VAR1);
    unsafe {
        println!("{}", VAR2);
    }
}
```

1 `unsafe` Features

2 Type Sizing

3 FFI

- With C
 - `bindgen`
- With C++
 - `cxx`

Zero Sized Types

There are a number of types in Rust which take up zero bytes!

- `()`
- `enums` with one variant
- Unit-like `structs`
- `structs` with entirely zero-sized fields

Zero Sized Types

There are a number of types in Rust which take up zero bytes!

- `()`
- `enums` with one variant
- Unit-like `structs`
- `structs` with entirely zero-sized fields

This is useful in some cases: something like a `Set<T>` can be implemented as a `Map<T, ()>` and because the compiler knows the values are zero-sized, it can avoid loads and stores to memory.

Likewise, something like `Vec<()>` can avoid allocating.

Sometimes, we do have to be careful about accounting for ZSTs in `unsafe` code, because it means the size of a type might not give us a valid offset or alignment.

Dynamically Sized Types

We've already seen a couple of dynamically sized types:

Dynamically Sized Types

We've already seen a couple of dynamically sized types:

- `dyn Trait`

Dynamically Sized Types

We've already seen a couple of dynamically sized types:

- `dyn Trait`
- `[T], str`

Dynamically Sized Types

We've already seen a couple of dynamically sized types:

- `dyn Trait`
- `[T]`, `str`

These can't be stored directly on the stack, because we **don't know their size at compile time**.

We say these types “don't implement `Sized`”, which can be denoted as `T: ?Sized`.

Dynamically Sized Types

`struct`s are allowed to have a DST as their last field, and if so, will themselves be a DST:

```
struct AllocBlock {  
    header: u64,  
    data: [u8],  
}
```

Layouts

Unlike C, Rust does not guarantee a specific data layout for your types, e.g.:

```
struct Foo {  
    i: i32,  
    f: f64,  
    j: i32,  
}
```

might only take up 16 bytes, instead of 24! That said, all `Foos` will have the same layout (for a given compiler version, subject to some other caveats).

Layouts

Unlike C, Rust does not guarantee a specific data layout for your types, e.g.:

```
struct Foo {  
    i: i32,  
    f: f64,  
    j: i32,  
}
```

might only take up 16 bytes, instead of 24! That said, all `Foos` will have the same layout (for a given compiler version, subject to some other caveats).

... but what if I want a guaranteed layout because I'm doing something which relies on it?

Specifying a Layout

There are a number of attributes we can use to ensure a specific layout:

¹really, one non-zero sized field

Specifying a Layout

There are a number of attributes we can use to ensure a specific layout:

- Rust—the default

¹really, one non-zero sized field

Specifying a Layout

There are a number of attributes we can use to ensure a specific layout:

- Rust—the default
- C—do what C does; not valid for some types

¹really, one non-zero sized field

Specifying a Layout

There are a number of attributes we can use to ensure a specific layout:

- Rust—the default
- C—do what C does; not valid for some types
- transparent—for one field structs¹; do the same as that field.

¹really, one non-zero sized field

Specifying a Layout

There are a number of attributes we can use to ensure a specific layout:

- Rust—the default
- C—do what C does; not valid for some types
- transparent—for one field structs¹; do the same as that field.
- packed—no padding; byte aligned

¹really, one non-zero sized field

Specifying a Layout

There are a number of attributes we can use to ensure a specific layout:

- Rust—the default
- C—do what C does; not valid for some types
- transparent—for one field structs¹; do the same as that field.
- packed—no padding; byte aligned
- a couple more we won't discuss, see [here](#)

¹really, one non-zero sized field

Specifying a Layout

There are a number of attributes we can use to ensure a specific layout:

- Rust—the default
- C—do what C does; not valid for some types
- transparent—for one field structs¹; do the same as that field.
- packed—no padding; byte aligned
- a couple more we won't discuss, see [here](#)

To use one of these, we use an attribute:

```
#[repr(C)]  
struct UnboundedArray<T> {  
    len: usize,  
    capacity: usize,  
    contents: *mut T,  
}
```

¹really, one non-zero sized field

1 `unsafe` Features

2 Type Sizing

3 FFI

- With C
 - `bindgen`
- With C++
 - `cxx`

With C

So You Want To Call C From Rust, Huh?

Conceptually, not too bad, just a few simple steps:

- Declare what C functions are available
- Link against the C library
- Call the function, using `unsafe`

Review: Calling Conventions

a.k.a. the Application Binary Interface (ABI)

Review: Calling Conventions

a.k.a. the Application Binary Interface (ABI)

a.k.a. how to talk to other people ('s C code)

Review: Calling Conventions

a.k.a. the Application Binary Interface (ABI)

a.k.a. how to talk to other people ('s C code)

- How are arguments passed?
- What registers are clobbered?
- How do you get the return value?

Rust Supported Calling Conventions (via LLVM)

- “Rust”—Rust’s own calling convention
- “C”—(default) calling convention used by your C compiler
- “system”—calling convention used by your OS, usually same as “C” except on Win32 where it’s “stdcall”
- “cdecl”—x86_32 calling convention
- “stdcall”—Win32 x86_32 ABI
- “win64”—x86_64 Windows ABI
- “sysv64”—x86_64 non-Windows
- “aapcs”—ARM
- “fastcall”
- “vectorcall”

See <https://doc.rust-lang.org/reference/items/external-blocks.html> for details.

External Linkage With `extern`

```
// Function we can link against
extern "C" {
    fn my_other_c_function(x: i32, y: i32) -> i32;
}

// Function that we export and can be linked to
#[no_mangle]
extern "C" fn my_rust_function(x: i32, y: i32) -> i32 { ... }
```


Review¹: Linkage

How can we link code?

¹okay, probably not

Review¹: Linkage

How can we link code?

- Dynamic Linkage: “hey OS i want this library please load it when u launch me”

¹okay, probably not

Review¹: Linkage

How can we link code?

- Dynamic Linkage: “hey OS i want this library please load it when u launch me”
 - Pros: Smaller binary size, flexible to upgrade library

¹okay, probably not

Review¹: Linkage

How can we link code?

- Dynamic Linkage: “hey OS i want this library please load it when u launch me”
 - Pros: Smaller binary size, flexible to upgrade library
 - Cons: Code can't handle upgrades in a significant number of cases

¹okay, probably not

Review¹: Linkage

How can we link code?

- Dynamic Linkage: “hey OS i want this library please load it when u launch me”
 - Pros: Smaller binary size, flexible to upgrade library
 - Cons: Code can't handle upgrades in a significant number of cases
- Static Linkage: “hey Compiler i want this library please put it next to my code”

¹okay, probably not

Review¹: Linkage

How can we link code?

- Dynamic Linkage: “hey OS i want this library please load it when u launch me”
 - Pros: Smaller binary size, flexible to upgrade library
 - Cons: Code can't handle upgrades in a significant number of cases
- Static Linkage: “hey Compiler i want this library please put it next to my code”
 - Pros: You always get the library version you want

¹okay, probably not

Review¹: Linkage

How can we link code?

- Dynamic Linkage: “hey OS i want this library please load it when u launch me”
 - Pros: Smaller binary size, flexible to upgrade library
 - Cons: Code can't handle upgrades in a significant number of cases
- Static Linkage: “hey Compiler i want this library please put it next to my code”
 - Pros: You always get the library version you want
 - Cons: Upgrading requires re-compilation

¹okay, probably not

Specifying Linkage for `extern` C Functions

```
#[link(name = "foo")] // kind = "dylib"
extern {
    fn cool_foo() -> *const u8;
}

#[link(name = "bar", kind = "static")]
extern {
    fn cool_bar() -> *const u8;
}
```


Things To Watch Out For

A couple of potential linking pitfalls:

- Your compiler can find the library you're linking against
 - For dynamic libraries, OS needs to find too!
- Your definitions in Rust exactly match the definitions in C

bindgen

Idea: Computer, Write Rust FFI For Me

Steps:

- Tell `bindgen` to make bindings at compile time
- Use `include!` macro to textually include generated bindings
- Link against C library
- Call the functions using `unsafe`

How Do We Do Stuff At Compile Time?

`build.rs` scripts!

- Placed at root of package next to `Cargo.toml`
- Run before Rust code compiled, can do arbitrary configuration since it's a binary itself
- Special output used to control behavior of Cargo

Small build.rs Example

```
fn main() {  
    // Tell Cargo that if the given file changes, to rerun this  
    // build script.  
    println!("cargo:rerun-if-changed=src/hello.c");  
  
    // Use the `cc` crate to build a C file and statically link it.  
    cc::Build::new()  
        .file("src/hello.c")  
        .compile("hello");  
}
```

bindgen build.rs Example

```
fn main() {  
    println!("cargo:rustc-link-lib=bz2");  
    println!("cargo:rerun-if-changed=wrapper.h");  
    let bindings = bindgen::Builder::default()  
        .header("wrapper.h")  
        .parse_callbacks(Box::new(bindgen::CargoCallbacks))  
        .generate()  
        .expect("Unable to generate bindings");  
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());  
    bindings  
        .write_to_file(out_path.join("bindings.rs"))  
        .expect("Couldn't write bindings!");  
}
```

build.rs Handles Linkage For Us!

```
println!("cargo:rustc-link-lib=bz2");
```

does dynamic linking, looking for libbz2.so, and

```
println!("cargo:rustc-link-lib=static=bz2");
```

does static linking, looking for libbz2.a

See <https://doc.rust-lang.org/cargo/reference/build-scripts.html> for all options

Including Generated Bindings

This step needs to be done because Cargo only looks at the source tree for files to compile, and `build.rs` scripts should not be modifying that directly:

```
// Contents of src/lib/ffi.rs  
  
#![allow(non_upper_case_globals)]  
#![allow(non_camel_case_types)]  
#![allow(non_snake_case)]  
include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```


Example C Header To Parse

```
typedef struct CoolStruct {  
    int x;  
    int y;  
} CoolStruct;  
  
void cool_function(int i, char c, CoolStruct* cs);
```

Example bindgen Generated Bindings

```
#[repr(C)]
pub struct CoolStruct {
    pub x: ::std::os::raw::c_int,
    pub y: ::std::os::raw::c_int,
}

extern "C" {
    pub fn cool_function(i: ::std::os::raw::c_int,
                        c: ::std::os::raw::c_char,
                        cs: *mut CoolStruct);
}
```

With C++

One Option: “C++ is just C”

One Option: “C++ is just C”

- Make a C interface to your C++ library

One Option: “C++ is just C”

- Make a C interface to your C++ library
- Use the same techniques as before to use that interface in Rust

One Option: “C++ is just C”

- Make a C interface to your C++ library
- Use the same techniques as before to use that interface in Rust
- ???

One Option: “C++ is just C”

- Make a C interface to your C++ library
- Use the same techniques as before to use that interface in Rust
- ???
- Profit?

The Issue: C++ Is Not Just C

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
 - `std::string` → `char *`

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
 - `std::string` → `char *`
 - `std::vector<int>` → `int *`

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
 - `std::string` → `char *`
 - `std::vector<int>` → `int *`
- We lose safety guarantees if we just use pointers

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
 - `std::string` → `char *`
 - `std::vector<int>` → `int *`
- We lose safety guarantees if we just use pointers
- Hey, wait a minute, doesn't Rust solve those same problems?

CXX

Main Features

- Shared Structs/Enums
- Opaque Types (on either side)
- Functions (on either side)
 - Not type-generic ones though!

Canonical Example

```
#[cxx::bridge]
mod ffi {

    extern "Rust" {
        // Rust stuff
    }

    unsafe extern "C++" {
        // C++ stuff
    }

}
```

Rust Stuff: All The Stuff You Love!

```
type MultiBuf;
```

```
fn next_chunk(buf: &mut MultiBuf) -> &[u8];
```

- Can also use `String`, `&str`, `Vec<T>`, `&[T]`, `Box<T>!`
- Converted to `rust::String`, `rust::Str`, `rust::Slice<T>`, `rust::Box<T>`, `rust::Vec<T>` in C++ code
- These are C++-native types, with the utilities you expect, much easier to work with than raw pointers

C++ Stuff: All The Stuff You Can Tolerate!

- `std::unique_ptr<T>`, `std::shared_ptr<T>`, `std::string`, `std::vector<T>`
- Converted to `UniquePtr<T>`, `SharedPtr<T>`, `CxxString`, `CxxVector` in Rust code
- `Result<T>` from Rust will be `rust::Error` in C++ and a C++ function throwing an exception will be `Result<T, cxx::Exception>` in Rust

C++ Stuff: Code Example

```
include!("example/include/blobstore.h");

type BlobstoreClient;

fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;

fn put(self: &BlobstoreClient, buf: &mut MultiBuf) -> Result<u64>;
```

Not Quite Complete

There are a couple missing features on the todo list:

- Async Rust with Futures \leftrightarrow C++ Coroutines: used to be not present but we have this now! It's great!
- C++ function pointers \rightarrow Rust