Effizientes Programmieren in C, C++ und Rust

Unsafe Rust

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024

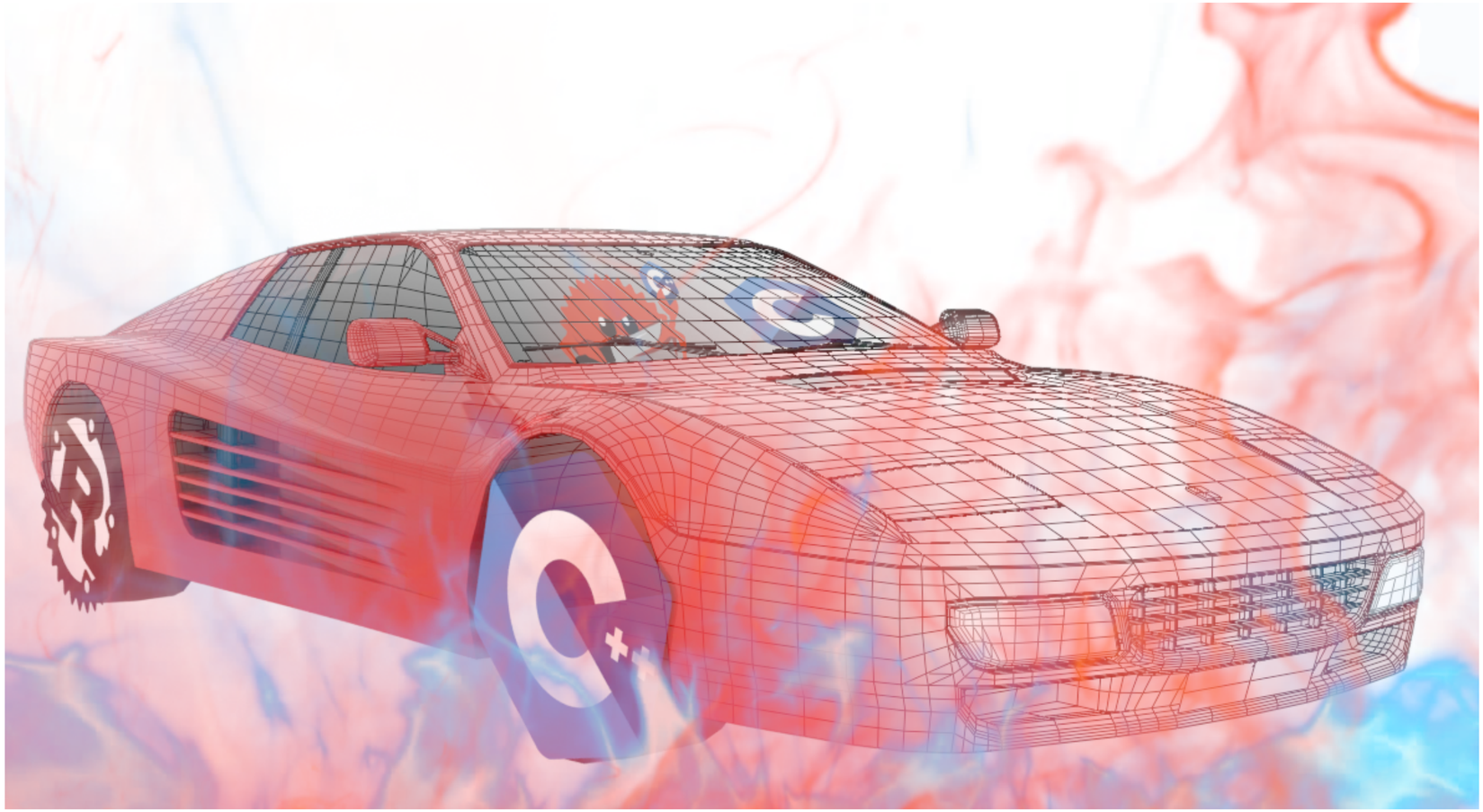


Table of Contents

An Unsafe Foundation?

- All Rust code relies on *unsafe* Rust under the hood

Common Misconception:
Rust can't be safe if its foundation is unsafe!

style="display: block; width: 90%;
float: right"/>

Actually, this is how all of programming works...

- Assembly is a safer abstraction upon the processor internals
- Operating systems abstract unsafe hardware
- Java is built upon the unsafe low-level code that runs the JVM

An Unsafe Foundation?

- Safe abstractions over unsafe foundations are everywhere...
- ...but in Rust, this divide is an integral part of the language
  - *Most popular languages are almost completely on one (or the other) side of the divide**

Use cases for unsafe

- Implementation of efficient data structures
- Specific performance improvements for hot code (e.g. removing a bounds check)
- Foreign function calls (FFI): calling a C library
- Direct interaction with the OS or hardware

⇒ Some things are impossible or inefficient using only safe Rust

** let's not talk about Go here*

The Concept of Unsafe

- *Guide to unsafe Rust:* *The Rustonomicon*
- Goal of unsafe: enable all the things that are possible in C
- Unsafe *does not* disable the borrow checker
- Violating the borrowing rules via unsafe causes undefined behavior

⇒ Unsafe is the subset of Rust where UB exists

`unsafe` marks code where the programmer must uphold invariants that can't be checked by the compiler

```
let ptr = &my_data as *mut T;
unsafe {
    // do things with raw pointers
}
```

The Concept of Unsafe

- Unsafe allows five things that are otherwise forbidden:
  - Dereference raw pointers
  - Call `unsafe` functions (including FFI)
  - Implement `unsafe` traits
  - Obtain exclusive references to global variables *(a very bad idea anyways)*
  - Access a `union` *(extremely niche)*

Summary

- We get access to raw pointers and already existing unsafe APIs
  - *Note: one of these APIs is inline assembly (not discussed further here)*

Magic Unsafe APIs in `std`

Magic 🪄: shorthand for "special cased by the compiler, can't be implemented as a library"

`UnsafeCell<T>`

- Allows mutation through shared references
- Basis for custom interior mutability

`MaybeUninit<T>*`

- Allows the contained value to be uninitialized

  ⇒ Enables uninitialized memory

`ManuallyDrop<T>`

- Disables automatic dropping of contained value

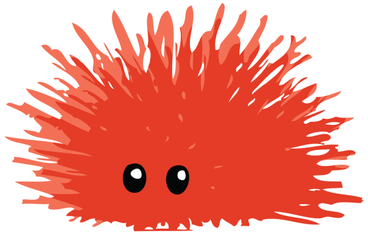
`Pin<&mut T>`

- Allows to construct unmovable values

  ⇒ Stable address, self references

** this one actually isn't truly magic*

Raw Pointers

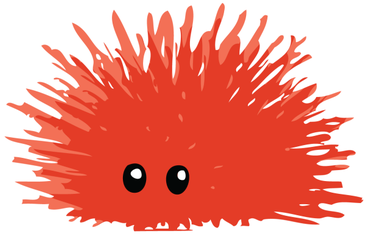


Unlike references, raw pointers...

- Are nullable
- Have no lifetime as a type (not borrow-checked) ⟵ *compile-time concern!*
- Are allowed to **alias** ⟵ *run-time concern! (does not cause UB)*
- Can point to uninitialized memory
- Support pointer arithmetic
- Allow memory access without a move or drop

UB warning: Aliasing is restricted again when pointers and references interact!

Raw Pointers



Creating a new raw pointer

- There are three fundamental ways to create a raw pointer
- Other raw pointers are just copies

1. Cast a reference to a pointer
2. Use `addr_of!` / `addr_of_mut!`
   - *Does **not** create a temporary reference*
3. Get a raw pointer from `malloc` (or another external function)

```
let ptr = &pointee as *const i32;
```

```
let ptr = addr_of!(pointee);
```

```
let tmp = Box::<i32>::new(0);
let ptr = Box::into_raw(tmp);
```

- 1. should generally be avoided (easy to cause UB)
- Note: creating raw pointers is safe, using them is unsafe

Raw Pointers

Example: Quicksort

(simplified: splitter should not be included in recursion)

```
fn partition_raw(data: &mut [u32], splitter: u32) -> usize {
    let Range {start: mut first_large, end } = data.as_mut_ptr_range();
    let mut first_unseen: *mut u32 = first_large;

    while first_unseen < end { unsafe {
        if first_unseen.read() <= splitter {
            ptr::swap(first_large, first_unseen);
            first_large = first_large.add(1);
        }
        first_unseen = first_unseen.add(1);
    }}
    unsafe { first_large.offset_from(data.as_mut_ptr()) as usize }
}

fn quicksort(data: &mut [u32]) {
    if data.len() <= 1 { return; }
    let splitter = /* determine splitter randomly */;
    let mid = partition_raw(data, splitter);
    let (left, right) = data.split_at_mut(mid);
    quicksort(left); quicksort(right);
}
```

The Safety Contract

- Any API that is not safe **must** be marked with `unsafe`
- Unsafe code can and should be encapsulated behind a safe API

  $\Rightarrow$ Safe and efficient code reuse

Asymmetric Trust

- Safe code is based on absolute trust that unsafe is correctly encapsulated
- Unsafe code **must not** trust the calling (safe) code at all

Safe API: A function is safe if **no possible input** causes Undefined Behavior.

Example: Generic Sorting

- Goal: efficient generic sorting algorithm (requires `unsafe`) with safe API
- Comparison operation is user-provided via the `Ord` trait

The Safety Contract

Example: Generic Sorting

- Goal: efficient generic sorting algorithm with safe API
- Comparison operation is user-provided via the `Ord` trait

Safe API: A function is safe if **no possible input** causes Undefined Behavior.

The Worst Case

- A malevolent caller might implement comparisons as follows:
  1. Throw two coins
  2. Return the random result, or panic as fourth option

⇒ Must not cause UB for any possible sequence of non-sensical comparisons

Problems

- Panic safety: unexpected panic could cause double free

  ⇒ Use `ManuallyDrop`
- Out of bounds access through inconsistent comparison results

Undefined Behavior in Rust

- The basic principle of UB in Rust is exactly the same as in C and C++
  - *The compiler may do optimizations under the assumption that UB **never happens***
- Only some UB is the same in both Rust and C++

Similar UB in Rust and C++

- Use-after-free
- Double free
- Data races
- (Unchecked) out of bounds access
  aka buffer overflow
- Using pointer arithmetic to move a pointer outside its original allocation
  (no access required!)
- Mutating data declared immutable
- Invalid (reinterpret) casts
- Foreign function call with incompatible ABI

Undefined Behavior in Rust

UB in C++, but not in Rust

- Signed integer overflow
- Aliasing of pointers with different type (except `char*` and `void*`)
- Accessing data through a pointer with wrong type* (though the data is valid for both types)
- Comparing pointers from different allocations
  (though computing the exact difference is still UB in Rust)

** i.e., no object of this type was previously constructed at the accessed location*

UB in Rust, but not in C++

- Violating the borrowing rules
  (i.e., aliasing a mutable reference)
- Mutation through a shared reference without interior mutability *(ignoring `const` e.g. via `const_cast` is just fine in C++)*
- The sole **existence** of a reference to uninitialized memory is perhaps UB*

** this is still in discussion, though current opinion favors "not UB"*

Unsafe Case Study: Removing Bounds Checking

```rust
fn transform_without_bounds_check(data: &mut [MyData]) {
    let index_mapping: Vec<usize> = // ...
    for &mapped_index in &index_mapping {
        debug_assert!(mapped_index < data.len());
        // SAFETY: part XY of algorithm ensures that all entries
        // in `index_mapping` are correct indizes into `data`
        let val = unsafe { data.get_unchecked_mut(mapped_index) };
        // ...
    }
}
```

Remarks

- Best practice: add safety comment for usages of **`unsafe`**
  - *Preconditions for soundness of the code*
  - *Why (you think) the preconditions hold*
  - *Add debug assertions for the preconditions if possible*
- Compiler can sometimes remove bounds checks, but not always
- Often, bounds checks can be avoided via iterators instead

Unsafe Case Study: Constructing data on the heap (V1)

```rust
fn array_on_heap_naive() -> Box<[u32; 10000]> {
    let mut array = [0; 10000];
    for (index, val) in array.iter_mut().enumerate() {
        *val = compute_from_index(index);
    }
    Box::new(array)
}
```

Benchmark: 2.31 μs *godbolt.org/z/567qqzbWv*

Problems?

- Step 1: zero-initialization
- Step 2: initialization with actual values
- Step 3: copy the array from stack to heap

$\Rightarrow$ Steps 1 and 3 are unnecessary, but the (current) compiler won't remove them*

$\Rightarrow$ Unnecessary writes, rather wasteful for cache, blowup of stack size

** though it executes each step extremely efficient*

Unsafe Case Study: Constructing data on the heap (V2)

```rust
fn array_on_heap_from_fn() -> Box<[u32; 10000]> {
    // this std function looks like magic...
    let array = array::from_fn(compute_from_index);
    Box::new(array)
}
```

Benchmark: 1.85 μs

Remarks

- Compiler deduces (backwards!) size of array from return type

  $\Rightarrow$ Type inference is sometimes almost magic

- Removes zero-initialization
  (though older compiler versions introduce a second copy instead...)
- Copy from stack to heap (which is worse) is still there

  $\Rightarrow$ Better, but still not efficient

Unsafe Case Study: Constructing data on the heap (V3)

```rust
fn array_on_heap_unsafe() -> Box<[u32; 10000]> {
    let mut result = Box::new([MaybeUninit::<u32>::uninit(); 10000]);
    for (index, val) in result.iter_mut().enumerate() {
        // SAFETY: pointer from `as_mut_ptr` is aligned and valid
        unsafe { *val.as_mut_ptr() = compute_from_index(index); }
    }
    // SAFETY: - everything is initialized
    // - MaybeUninit has same alignment and size as underlying type
    unsafe { mem::transmute(result) }
}
```

Benchmark: 1.27 μs

Problems

- "Maybe" UB, since the write might semantically create a reference to uninitialized memory
  (same discussion as already mentioned)
- Guaranteed UB if used with a `Drop` type instead of `u32`, since uninitialized value is dropped

  $\Rightarrow$ Fix: `ptr::write` does not drop old value *(and thus can't create a reference)*

Unsafe Case Study: Constructing data on the heap (V4)

```rust
fn array_on_heap_unsafe() -> Box<[u32; 10000]> {
    let mut result = Box::new([MaybeUninit::<u32>::uninit(); 10000]);
    for (index, val) in result.iter_mut().enumerate() {
        // SAFETY: - pointer from `as_mut_ptr` is aligned and valid
        // - `write` involves no references or dropping
        unsafe { val.as_mut_ptr().write(compute_from_index(index)); }
    }
    // SAFETY: - everything is initialized
    // - MaybeUninit has same alignment and size as underlying type
    unsafe { mem::transmute(result) }
}
```

Benchmark: 1.27 μs

Success!

- This code is both safe and efficient
- Almost factor 2 improvement to naive version

Unsafe Case Study: Eliminating Branches

```rust
fn normal_div(input: i32, divisor: i32) -> i32 {
    input / divisor
}
```

Fast division?

- Szenario: we need to divide a lot
- Divisions are slow anyways
- But is the generated assembly at all optimal?

- Oh no, the generated assembly has a branch with a panic
  (actually there are even two)
- Reason: division by zero

⇒ `unreachable_unchecked` allows to remove branches

Unsafe Case Study: Eliminating Branches

```rust
// SAFETY: caller must ensure divisor != 0
unsafe fn unchecked_div(input: i32, divisor: i32) -> i32 {
    if divisor == 0 {
        // SAFETY: see precondition of function
        unsafe { unreachable_unchecked() }
    }
    input / divisor
}
```

Fast division?

- `unreachable_unchecked` is defined as "causes immediate UB"
- Guaranteed UB allows the compiler to remove all branches leading there
- Use with caution!
- This eliminates one branch
- But wait, there is still a panic? Why?
- Answer: Overflow!

Unsafe Case Study: Eliminating Branches

```rust
// SAFETY: caller must ensure divisor != 0 and no overflow happens
unsafe fn unchecked_div(input: i32, divisor: i32) -> i32 {
    if divisor == 0 || (divisor == -1 && input == i32::MIN) {
        // SAFETY: see precondition of function
        unsafe { unreachable_unchecked() }
    }
    input / divisor
}
```

Fast division

- `i32::MIN / -1 == i32::MAX + 1`
- Signed integer division can overflow in exactly one case
- Luckily, we can remove it, too

⇒ Equivalent to normal division in C++

```
mov eax, edi
cdq                 # sign extension
idiv esi            # signed division
ret
```

Unsafe Case Study: Doubly Linked Lists

- References can only reference external data
  (external to the struct holding the reference)
- We can't build data structures from references
- Box supports only single ownership

  ⇒ We need raw pointers*

  ⇒ Just implement it like we would in C

Problem:

Moving a node invalidates pointers...
How to construct the list?

** or do something cyclic with reference counting, but that would be rather crazy, right?*

```rust
struct Node<T> {
    next: *mut Node<T>,
    prev: *mut Node<T>,
    element: T,
}

impl<T> Node<T> {
    pub fn new(element: T) -> Self {
        // ...
    }

    pub fn get(&self) -> &T {
        &self.element
    }

    pub fn get_mut(&mut self) -> &mut T {
        &mut self.element
    }

    // this function is actually a bad idea
    pub fn next(&mut self) -> &mut Node<T> {
        unsafe { &mut *self.next }
    }
}
```

Unsafe Case Study: Doubly Linked Lists

- Introduce some indirection so nodes are always on the heap
- Achieved via wrapper struct

Problem: undefined behavior!

- Note: we can detect UB with Miri
  (like a more powerful usan, but as an interpreter)
- Trying to actually use the list results in

error: Undefined Behavior: attempting a write access using <2414> at alloc1135[0x0], but that tag does not exist in the borrow stack for this location

```
struct Node<T> {
    next: *mut Node<T>,
    prev: *mut Node<T>,
    element: T,
}

struct LinkedList<T> {
    head: Box<Node<T>>,
}

impl<T> LinkedList<T> {
    pub fn new(element: T) -> Self {
        let mut head = Box::new(
            Node::new(element)
        );
        // initialize pointers
        head.next = addr_of_mut!(*head);
        head.prev = addr_of_mut!(*head);
        Self { head }
    }
}
```

Raw Pointers and Aliasing

- Raw pointers may alias with other raw pointers...
- ...but the invariants of references can also be broken with pointers

- Creating an exclusive reference invalidates pointers to the data
- Writes invalidate shared references
- Note: Box is roughly like an exclusive reference

```
let mut test = 0;
let ptr_1 = addr_of_mut!(test);
let ptr_2 = addr_of_mut!(test);
unsafe {
    *ptr_1 = 1;
    *ptr_2 = 2;
}
```

```
let mut test = 0;
let ptr_1 = &mut test as *mut _;
let ptr_2 = &mut test as *mut _;
//              ^^^^ invalidates ptr_1
unsafe {
    *ptr_1 = 1;  // <-- UB here!
    *ptr_2 = 2;
}
```

Raw Pointers and Aliasing

Stacked Inheritance

- References/pointers inherit the permissions of their "parent"
- The parent of a reference is not invalidated *(note: copies of pointers are considered the same pointer)*
- Writes of the parent invalidate children
- This is called the borrow stack*

** though there are two different formalization attempts*

```rust
let mut test = 0;
let ptr_1 = addr_of_mut!(test);
let mut_ref = unsafe { &mut *ptr_1 };
let ptr_2 = mut_ref as *mut _;
unsafe {
    *mut_ref = 42;
    *ptr_1 = 1;
}
```

```rust
let mut test = 0;
let ptr_1 = addr_of_mut!(test);
let mut_ref = unsafe { &mut *ptr_1 };
let ptr_2 = mut_ref as *mut _;
unsafe {
    *mut_ref = 42;
//  ^^^^^^^^^^^^^ invalidates ptr_2
    *ptr_2 = 2;
}
```

Back to Linked Lists

- Accesses through Box invalidate raw pointers
- Use another raw pointer instead

Rough Guideline

- Avoid mixing references/Box with raw pointers
- Internally, use only pointers derived from the allocation itself
- Convert pointers to references at public API boundary
- However: there are quite a few more details to build a proper linked list in Rust

```
struct Node<T> {
    next: *mut Node<T>,
    prev: *mut Node<T>,
    element: T,
}

struct LinkedList<T> {
    head: *mut Node<T>,
}

impl<T> LinkedList<T> {
    pub fn new(element: T) -> Self {
        // get the allocation pointer
        let mut head = Box::into_raw(
            Box::new(Node::new(element))
        );
        unsafe {
            (*head).next = head;
            (*head).prev = head;
        }
        Self { head }
    }
}
```

Summary

What we learned

- There is an asymmetric **safety contract** between safe and unsafe code
- Unsafe allows low-level control and some specific optimizations
- It is possible to remove virtually any overhead with unsafe
- Unsafe Rust is not like C

Unsafe "gotchas"

- Panic safety
- Drop safety
- Uninitialized memory
- Interactions between raw pointers and references