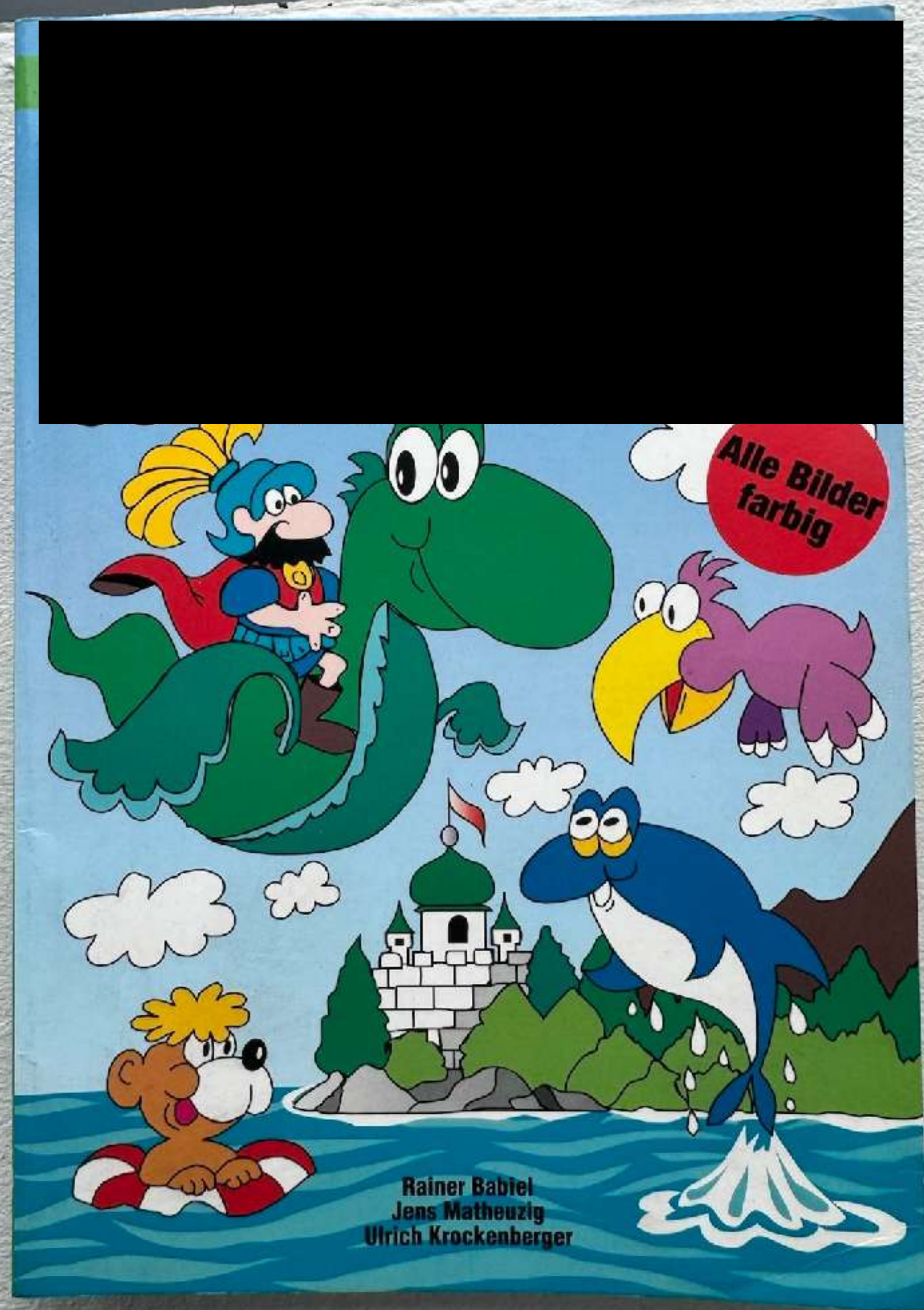


Unsafe for Work



Stefan Baumgartner - oida.dev - deadparrot@mastodon.social - @deadparrot.bsky.social



Hero with a cape

On top a friendly dragon

Rescues a princess

On his long journey

He battles fierce animals

And rides a dolphin

What's the deal with unsafe?

Should we be worried about proliferation of unsafe in Rust code?

I've noticed that despite the Rust guarantees of safety, there is an awful lot of `unsafe` blocks, functions and calls all over Rust code. Here is the definition of `Rc`, for example:

```
impl<T> Rc<T> {
    #[cfg(not(no_global_oom_handling))]
    #[stable(feature = "rust1", since = "1.0.0")]
    pub fn new(value: T) -> Rc<T> {
        unsafe {
            Self::from_inner(
                Box::leak(Box::new(RcBox { strong: Cell::new(1), weak: Cell::new(0), value: value }))
            )
        }
    }
}
```

Don't know about you, but I'm not comfortable knowing that anytime I use `Rc`, there is something unsafely leaking (`Box::leak`) in memory.

Code for `RefCell` [also contains lots of unsafe](#).

As does the mainstay data structure of Rust, [Vec](#).

But maybe this unsafety proliferates only on the core language level, and doesn't spread to library code? Nope, a quick search on some of the most well-known Rust libraries shows lots of `unsafe`:

3680 in [azul](#)

147 in [rayon](#)

2 functions and 1 pattern match in [ripgrep](#)

25 in [rust-crypto](#)

Is rust `unsafe` just moving the problem?

(Warning: this post is written by a self taught n00b whose day job is web development) I was listening [to this podcast](#), and about the 35 minute mark, the interviewee says that Rust hasn't really solved the problem for memory management, they have just shifted it.

I have a few questions about this comment. First, is memory management in game development that much of a different problem than normal programming where the borrow checker doesn't work for 95% of the cases? And isn't this the point of Rust's `unsafe` keyword? Most of your programming can be handled with the borrow checker, and you can isolate the few places it doesn't work to make sure those areas are contained away from the majority of your code base? This is still a major advantage over C++ right (as far as memory management and reducing memory leaks)? Second, I know there are some game engines for rust (or is it just godot?), but they aren't yet in the place to replace Unity or the Unreal Engine. Do you think rust will replace C++ for the majority of Game development?

Unsafe Rust

Somewhat new to Rust, and obviously what stands out is the compile-time guarantees, especially concerning memory safety. I'm curious about the tradeoff though and the availability of "unsafe Rust," which seems to turn Rust into something closer to a modernized C++. How much do people use "unsafe Rust"? How well does Rust work without these features for distributed/multiproc systems?

- Unsafe code
 - Last resort
 - Bypasses borrow checker completely



https://www.reddit.com/r/rust/comments/1esj2v8/unsafe_rust/

https://www.reddit.com/r/rust/comments/1ox5s5k/is_rust_unsafe_just_moving_the_problem/

https://www.reddit.com/r/rust/comments/xs5wut/should_we_be_worried_about_proliferation_of/



unsafe deactivates
the borrow checker

No.

```
fn split<T>(slice: &[T], point: usize) -> (&[T], &[T]) {  
    (&slice[..point], &slice[point..])  
}
```

```
fn split<T>(slice: &[T], point: usize) -> (&[T], &[T]) {  
    (&slice[..point], &slice[point..])  
}
```

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {  
    (&mut slice[..point], &mut slice[point..])  
}
```



```
fn split<T>(slice: &[T], point: usize) -> (&[T], &[T]) {
    (&slice[..point], &slice[point..])
}
```

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {
    (&mut slice[..point], &mut slice[point..])
}
```

```
5 | fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {
  |                                     - let's call the lifetime of this reference `'1`
6 |     (&mut slice[..point], &mut slice[point..])
  |     -----^^^^^-----
  |     |         |               |
  |     |         |               second mutable borrow occurs here
  |     |         first mutable borrow occurs here
  |     returning this value requires that `*slice` is borrowed for `'1`
```

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {  
    unsafe { (&mut slice[..point], &mut slice[point..]) }  
}
```

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {
    unsafe { (&mut slice[..point], &mut slice[point..]) }
}

```

```

5 | fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {
  |                                     - let's call the lifetime of this reference `'1`
6 |     unsafe { (&mut slice[..point], &mut slice[point..]) }
  |
  |         -----^^^^-----
  |         |         |         |
  |         |         |         second mutable borrow occurs here
  |         |         |         first mutable borrow occurs here
  |         |         |         returning this value requires that `*slice` is borrowed for `'1`
  |

```


unsafe deactivates
the borrow checker

WRONG

A language superset

Unsafe Rust Allows

- Dereferencing raw pointers
- Calling unsafe functions and methods, including extern
- Implementing unsafe traits
- Mutate static variables
- Access unions

Pointer Types

Borrow checked

Unchecked

T Owned Type

$\&T$ Reference

$\&mut T$ Mutable Reference

$*const T$ Raw Pointer

$*mut T$ Mutable Raw Pointer

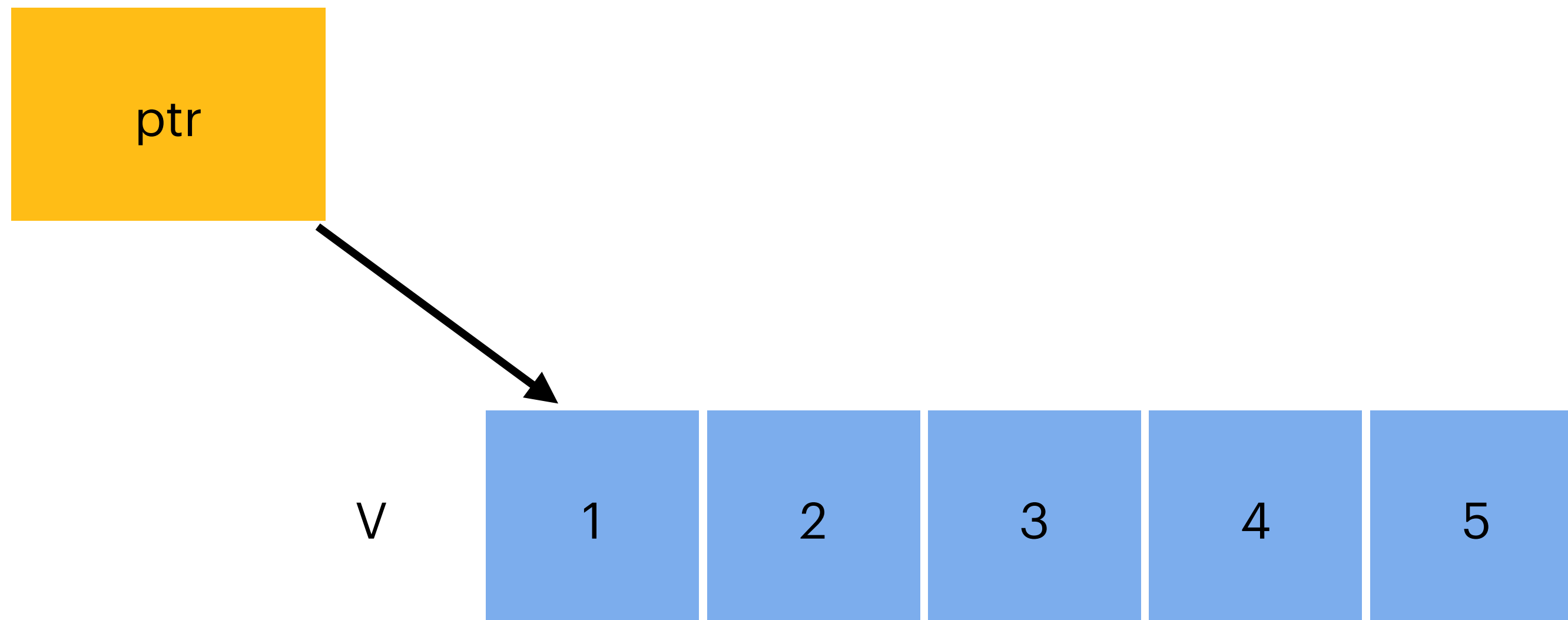
- The borrow checker is still enabled with the correct types: owned values, shared and mutable references — “borrows”
- There are more pointer types: Raw and mutable raw pointers! They are not being checked by the borrow checker.

```
let mut v = [1, 2, 3, 4, 5];  
let ptr = &raw mut v[0];
```

```
let ptr: *mut i32, 2, 3, 4, 5];  
let ptr = &raw mut v[0];
```




```
let mut v = [1, 2, 3, 4, 5];  
let ptr = &raw const v[0];
```



```
let mut v = [1, 2, 3, 4, 5];  
let ptr = &raw const v[0];  
println!("{:?}", ptr);
```

0x16f0262a4



It's just C/C++ all over again...


```
let mut v = [1, 2, 3, 4, 5];  
let ptr = &raw mut v[0];
```

```
*ptr = 9;  
println!("{:?}", *ptr);
```

```
let mut v = [1, 2, 3, 4, 5];  
let ptr = &raw mut v[0];  
  
*ptr = 9;  
println!("{}", *ptr);
```

```
error[E0133]: dereference of raw pointer is unsafe and  
requires unsafe function or block  
--> src/main.rs:33:5  
   |  
33 |     *ptr = 9;  
   |     ^^^^ dereference of raw pointer  
   |  
= note: raw pointers may be null, dangling or unaligned;  
they can violate aliasing rules and cause data races: all  
of these are undefined behavior
```

```
let mut v = [1, 2, 3, 4, 5];  
let ptr = &raw mut v[0];
```

```
unsafe { *ptr = 9 };  
println!("{}", unsafe { *ptr });
```

```
fn dangling_pointer() {  
    let ptr: *const i32;  
  
    {  
        let value = 42;  
        ptr = &raw const value;  
    }  
  
    unsafe {  
        println!("Dereferencing ptr: {}", *ptr);  
    }  
}
```

```
fn dangling_pointer() {  
    let ptr: *const i32;  
  
    {  
        let value = 42;  
        ptr = &raw const value;  
    }  
  
    unsafe {  
        println!("Dereferencing ptr: {}", *ptr);  
    }  
}
```

DEBUG
> 42

UNDEFINED BEHAVIOUR

RELEASE
> 24565

Unsafe Blocks

- unsafe blocks allow dereferencing raw pointers.
- They also allow calling of unsafe functions.
- For you as a developer this means: Make sure you do the right thing and try to meet all safety conditions.

```
fn danging_pointer() {  
    let ptr: *const i32;  
  
    {  
        let value = 42;  
        ptr = &raw const value;  
    }  
  
    🐉 { // Thar be dragons!  
        println!("Dereferencing ptr: {}", *ptr);  
    }  
}
```

Unsafe blocks and functions

Contracts

unsafe fn

- Unsafe functions highlight that this code might cause undefined behaviour. This behaviour needs to be checked!

unsafe {}

- Unsafe blocks highlight that you as a developer checked all preconditions to run unsafe code.

**unsafe moves safety checks from the
compiler to the developer**

Explicit.

It's just C/C++ all over again...

WRONG

Safe abstractions

Safe abstractions for unsafe code

- Unsafe can't be avoided, really.
- But it can be avoided to work with unsafe entirely.
- Unsafe code calls for safe abstractions.

A look at Bevy

- Bevy as a game engine has a lot of self-made data structures that are optimised for performance
- We take one example how Bevy uses unsafe to make safety guarantees.

```
pub struct ThinArrayPtr<T> {  
    data: NonNull<T>,  
    #[cfg(debug_assertions)]  
    capacity: usize,  
}
```

Similar to `Vec<T>`, but with the capacity and length cut out for performance reasons.

NonNull<T>

*mut T but non-zero and covariant.

This is often the correct thing to use when building data structures using raw pointers,

```

pub struct ThinArrayPtr<T> {
    data: NonNull<T>,
    #[cfg(debug_assertions)]
    capacity: usize,
}

impl<T> ThinArrayPtr<T> {
    fn empty() -> Self {
        Self {
            data: NonNull::dangling(),
        }
    }
}

```

Creates a new NonNull that is dangling, but well-aligned.

This is useful for initializing types which lazily allocate, like Vec::new does.

```

pub const fn dangling() -> Self {
    // SAFETY: mem::align_of() returns a non-zero
    // usize which is then casted
    // to a *mut T. Therefore, `ptr` is not null
    // and the conditions for
    // calling new_unchecked() are respected.
    unsafe {
        let ptr = crate::ptr::dangling_mut::<T>();
        NonNull::new_unchecked(ptr)
    }
}

```

```

pub struct ThinArrayPtr<T> {
    data: NonNull<T>,
    #[cfg(debug_assertions)]
    capacity: usize,
}

impl<T> ThinArrayPtr<T> {
    fn empty() -> Self {
        Self {
            data: NonNull::dangling(),
        }
    }

    pub fn with_capacity(capacity: usize) -> Self {
        let mut arr = Self::empty();
        if capacity > 0 {
            // SAFETY:
            // - The `current_capacity` is 0 because it was just created
            unsafe { arr.alloc(NonZeroUsize::new_unchecked(capacity)) };
        }
        arr
    }
}

```

Creates a non-zero without checking whether the value is non-zero. This results in undefined behaviour if the value is zero.

Safety

The value must not be zero.

```

pub const unsafe fn new_unchecked(n: T) -> Self {
    //...
}

```



```
pub struct ThinArrayPtr<T> {  
    data: NonNull<T>,  
    #[cfg(debug_assertions)]  
    capacity: usize,  
}
```

```
impl<T> ThinArrayPtr<T> {  
    fn empty() -> Self {  
        Self {  
            data: NonNull::dangling(),  
        }  
    }  
}
```

```
pub fn with_capacity(capacity: usize) -> Self {  
    let mut arr = Self::empty();  
    if capacity > 0 {  
        // SAFETY:  
        // - The `current_capacity` is 0 because it was just created  
        unsafe { arr.alloc(NonZeroUsize::new_unchecked(capacity)) };  
    }  
    arr  
}
```



Safe abstractions you have used.

- Vec<T>
- Rc<T>, Arc<T>
- Mutex<T>
- RwLock<T>
- ...

Unsafe for work

**You're fine.
Really.**

Unsafe Rust Allows

- Dereferencing raw pointers
- Calling unsafe functions and methods, including extern
- Implementing unsafe traits
- Mutate static variables
- Access unions

When do you need it.

- Hardware Access
- FFI
- Performance Optimisations

Language	files	blank	comment	code
Rust	194	7607	5596	51858
D	999	2508	0	13541
JSON	1103	0	0	10664
TypeScript	56	664	18834	9772
Markdown	53	1361	1	3211
JavaScript	38	321	1093	1852
Groovy	2	113	109	891
TOML	30	96	9	854
PlantUML	10	137	2	623
YAML	20	19	24	521
Dockerfile	9	58	55	222
Protocol Buffers	5	84	344	214
Bourne Shell	19	62	33	159
HTML	1	5	0	90
LLVM IR	6	26	22	82
Python	1	18	19	65
Text	2	13	0	52
INI	1	3	0	15
SUM:	2549	13095	26141	94686

→ runtime (main) ✓

1 occurrence of actively using unsafe

Language	files	blank	comment	code
Rust	194	7607	5596	51858
D	999	2508	0	13541
JSON	1103	0	0	10664
TypeScript	56	664	18834	9772
Markdown	53	1361	1	3211
JavaScript	38	321	1093	1852
Groovy	2	113	109	891
HTML	1	5	0	90
LLVM IR	6	26	22	82
Python	1	18	19	65
Text	2	13	0	52
INI	1	3	0	15
SUM:	2549	13095	26141	94686

→ runtime (main) ✓

```
// SAFETY: Assumes that V8 passes a valid string pointer
let details_c_str = unsafe { CStr::from_ptr(details.detail) };
details_c_str.to_string_lossy()
```

1 occurrence of actively using unsafe

// SAFETY



MatsRivel · 3mo ago ·

I think people think you just write all low level rust code in an unsafe-block.

I am working on a microcontroller project in Rust for work, and I am not yet using unsafe for anything, even when accessing memory directly (through the `esp_idf_svc` crate)



251



Reply



Award



Share



FuckFN_Fabi · 3mo ago ·

You are using a library that does the unsafe part for you... But it is great that many crates provide a "safe" unsafe implementation



179



Reply



Award



Share



Sapiogram · 3mo ago ·

But it is great that many crates provide a "safe" unsafe implementation

This is a great point, and (imo) one of Rust's primary reasons for existing: Allowing library authors to write safe abstractions on top of unsafe (In the Rust sense) primitives. Of course, this requires a high level of trust in library authors, but it's better than the alternative of every line of code being potentially unsafe.

esp-idf-svc - crates.io: Rust

crates.io/crates/esp-idf-svc/0.49.1

crates.io

Type 'S' or '/' to search

Browse All Crates

Log in with GitHub

esp-idf-svc v0.49.1

Implementation of the embedded-svc traits for ESP-IDF (Espressif's IoT Development Framework)

#embedded #esp32 #esp-idf #idf #svc

Readme

101 Versions

Dependencies

Dependents

Safe Rust wrappers for the services in the ESP IDF SDK

CI failing

crates.io v0.49.1

docs esp-rs

join matrix

2177 users

wokwi

Click to Simulate

Highlights

- Supports almost all ESP IDF services: timers, event loop, Wifi, Ethernet, HTTP client & server, MQTT, WS, NVS, OTA, etc.
- Implements the traits of `embedded-svc`
- Blocking and `async` mode for each service (`async` support where feasible)
- Re-exports `esp-idf-hal` and `esp-idf-sys` as `esp_idf_svc::hal` and `esp_idf_svc::sys`. You only need to depend on `esp_idf_svc` to get everything you need

Metadata

📅 4 months ago

🔖 v1.77.0

📄 MIT OR Apache-2.0

📦 188 KiB

Install

Run the following Cargo command in your project directory:

```
cargo add esp-idf-svc@=0.49.1
```

Or add the following line to your Cargo.toml:

```
esp-idf-svc = "=0.49.1"
```

Documentation

```
fn dangling_pointer() {  
    let ptr: *const i32;  
  
    {  
        let value = 42;  
        ptr = &raw const value;  
    }  
  
    unsafe {  
        println!("Dereferencing ptr: {}", *ptr);  
    }  
}
```

```
$ cargo +nightly miri run
```

```
fn dangling_pointer() {
    let ptr: *const i32;

    {
        let value = 42;
        ptr = &raw const value;
    }

    unsafe {
        println!("Dereferencing ptr: {}", *ptr);
    }
}
```

\$ cargo +nightly mi

```
error: Undefined Behavior: out-of-bounds pointer use: alloc1905 has been freed, so this
pointer is dangling
--> src/main.rs:15:43
15 |         println!("Dereferencing ptr: {}", *ptr);
    |         ^^^^^ out-of-bounds pointer use: alloc1905
                has been freed, so this pointer is
                dangling
= help: this indicates a bug in the program: it performed an invalid operation, and
caused Undefined Behavior
```

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {  
    let ptr = slice.as_mut_ptr();  
    let len = slice.len();  
  
    unsafe {  
        (  
            from_raw_parts_mut(ptr, point),  
            from_raw_parts_mut(ptr.add(point), len - point),  
        )  
    }  
}
```

```
$ cargo +nightly miri run
```

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {  
    let ptr = slice.as_mut_ptr();  
    let len = slice.len();  
  
    unsafe {  
        (  
            from_raw_parts_mut(ptr, point),  
            from_raw_parts_mut(ptr.add(point), len - point),  
        )  
    }  
}
```

```
$ cargo +nightly miri run
```



Havoc.

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {  
    let ptr = slice.as_mut_ptr();  
    let len = slice.len();  
  
    unsafe {  
        (  
            from_raw_parts_mut(ptr, point),  
            from_raw_parts_mut(ptr.add(point - 1), len - point + 1),  
        )  
    }  
}
```

```
$ cargo +nightly miri run
```

```

fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {
    let ptr = slice.as_mut_ptr();
    let len = slice.len();

    unsafe {
        (
            from_raw_parts_mut(ptr, point),
            from_raw_parts_mut(ptr.add(point - 1), len - point + 1),
        )
    }
}

```

```

error: Undefined Behavior: trying to retag from <4464> for Unique permission at
alloc1963[0x4], but that tag does not exist in the borrow stack for this location
--> src/main.rs:17:9
17 | /
18 | |     std::slice::from_raw_parts_mut(ptr, point),
19 | |     std::slice::from_raw_parts_mut(ptr.add(point - 1), len - point + 1),
20 | | )
   | | ^
   | | |
   | | | trying to retag from <4464> for Unique permission at alloc1963[0x4], but that...

```

\$ cargo +nightly m

Miri

- Miri helps identifying undefined behaviour
- Not everything can be caught by Miri, but it's a great start!
- When writing unsafe, use Miri for additional checks.

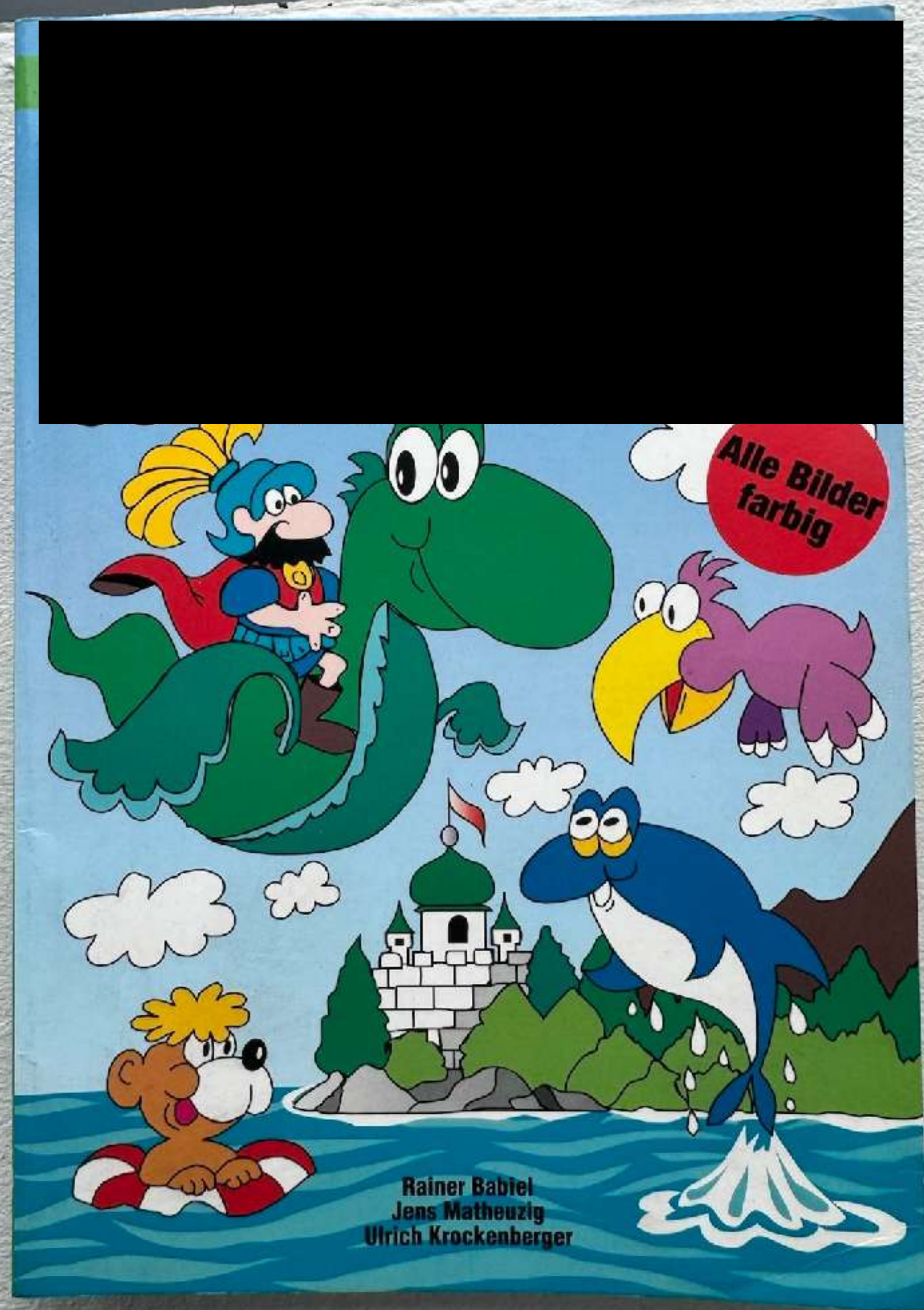
Unsafe is a Safety Feature

```
fn split<T>(slice: &[T], point: usize) -> (&[T], &[T]) {  
    (&slice[..point], &slice[point..])  
}
```

vs.

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut [T]) {  
    let ptr = &raw mut slice[0];  
  
    unsafe {  
        (  
            std::slice::from_raw_parts_mut(ptr, point),  
            std::slice::from_raw_parts_mut(ptr.add(point), slice.len() - point),  
        )  
    }  
}
```

Unsafe is about
highlighting and preventing
undefined behaviour



Hero with a cape
On top a friendly dragon
Rescues a princess

On his long journey
He battles fierce animals
And rides a dolphin

Markt & Technik

The title screen for Super Mario World. At the top, the words "SUPER" and "NINTENDO" are written in a blue, blocky font. Below them, the words "SUPER MARIO WORLD" are written in a large, bold, pink font with a black outline. The background is a light blue sky with a few small, stylized clouds. At the bottom, there are some green hills and a red mushroom.

SUPER NINTENDO SUPER MARIO WORLD

**Alle Bilder
farbig**

Rainer Babel
Jens Matheuzig
Ulrich Krockenberger



fin.