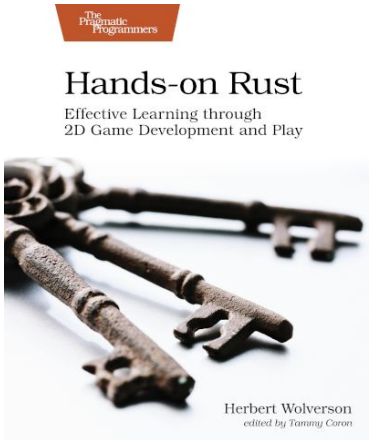
Rust Safety

Safe at Any Speed

herbert.wolverson@ardanlabs.com

About Herbert Wolverson

- Ardan Labs Rust Trainer & Consultant
- Author of *Hands-on Rust* and *Rust Brain Teasers*
- Author of the *Rust Roguelike Tutorial*
- Lead developer, *LibreQoS*, *bracket-lib*.
- Contributor to many open source projects.



What We’re Going to Cover

- Memory Safety
  - Bounds Checking & Buffer Overruns
  - Use After Free
  - Use After Move
  - Type Coercion
- Avoiding Bugs & Vulnerabilities with Clear Intent
- Data Races
- Some General Advice
- Q&A

Memory Safety

What does Memory Safety Mean?

A memory-unsafe program can be tricked into accessing data outside of its intended area.

You might give an attacker a chance to read your private data.

You might even let the attacked run code of their choice in your process.

Memory safety issues in C and C++ were a driving force behind the development of safe languages such as Java, the .NET family, Go and Rust.

Let’s Exploit Some C!

```c
#include <stdio.h>

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 10; i++) {
        printf("Loc: %d : %d\n", i, numbers[i]);
    }
    return 0;
}
```

Compile with “-Wall” - all warnings.

No warnings! We’re good, right?

The output:

```
Loc: 0 : 1
Loc: 1 : 2
Loc: 2 : 3
Loc: 3 : 4
Loc: 4 : 5
Loc: 5 : 0
Loc: 6 : 1795486976
Loc: 7 : 1051140916
Loc: 8 : 1
Loc: 9 : 0
```

Oh dear. You’ve read beyond your array. The program didn’t crash. You potentially just leaked important information!

If your program *wrote* to that memory with user provided input, you may have just allowed an attacker to take over your server.

How about some C++?

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    for (int i = 0; i < 10; i++) {
        std::cout << "Loc: " << i << " : " << numbers[i] << "\n";
    }
    return 0;
}
```

Basically the same, but using a C++ vector.

No warnings with -Wall

The output:

```
Loc: 0 : 1
Loc: 1 : 2
Loc: 2 : 3
Loc: 3 : 4
Loc: 4 : 5
Loc: 5 : 0
Loc: 6 : 1041
Loc: 7 : 0
Loc: 8 : 979595084
Loc: 9 : 975190304
```

Oh dear. It's the same problem!
Even though vectors store the length of the vector, the default doesn't *check* that you're performing a valid access.

(Note: MSVC in Debug mode does check!)

A Little C++ Good News

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    for (int i = 0; i < 10; i++) {
        std::cout << "Loc: " << i << " : " << numbers.at(i) << "\n";
    }
    return 0;
}
```

Using “at(index)” instead of [index].

No warnings

The output:

```
Loc: 0 : 1
Loc: 1 : 2
Loc: 2 : 3
Loc: 3 : 4
Loc: 4 : 5

terminate called after throwing an
instance of 'std::out_of_range'

  what():  vector::_M_range_check: __n
(which is 5) >= this->size() (which is 5)
```

Much better. Rather than handing out your corporate secrets, the program crashed.

This is what you want. It’s better to stop than to do something potentially catastrophic.

How about Go?

```go
package main

import "fmt"

func main() {
    array := [5]int{0, 1, 2, 3, 4}
    fmt.Println(array[5])
}
```

Fails to compile! The compiler detects that 5 is out of bounds, and refuses to compile the program.

```go
package main

import "fmt"

func main() {
    array := [5]int{0, 1, 2, 3, 4}
    for i := 0; i < 10; i++ {
        fmt.Println(array[i])
    }
}
```

Compiles and prints 0, 1, 2, 3, 4 - and then panics with an *Index Out of Range* error.

Perfect! Go won't be leaking your company secrets this way.

Rust

```rust
fn main() {
    let array = [0, 1, 2, 3, 4];
    println!("{}", array[5]);
}
```

Fails to compile. The compiler finds the out-of-bounds statically and won't compile the program.

```rust
fn main() {
    let array = vec![0, 1, 2, 3, 4];
    for i in 0..10 {
        println!("{}", array[i]);
    }
}
```

Prints 0 through 4 and panics with *index out of bounds*.

Perfect! Rust won't be exposing your corporate secrets, either.

Opting Out of Safety with Rust

There is a performance cost associated with bounds-checking every buffer access.

`if index < array.len()` is not free!

IF:

- Profiling can prove that this is your bottleneck.
- YOU can prove that your array indices are *always* going to be safe.

You can opt out of bounds-checking with `get_unchecked`.

```rust
fn main() {
    let array = vec![0, 1, 2, 3, 4];
    unsafe {
        println!("{}", array.get_unchecked(5));
    }
}
```

Prints “0”.

When you tell Rust to be **unsafe**, it can be just as unsafe as C.

You can *make this safe* - but *unverifiable* - if you know the index range and check it once, and then perform thousands of array accesses.

Even better… use an iterator that does this for you safely.

Use After Free & Move

What is Use After Free/Move?

Systems languages that let you manage your own memory need a way to *allocate* and *deallocate* memory.

You avoid memory leaks by allocating, and then deallocating when you're done with it. This is largely solved with RAII and/or reference counting.

Allocating, Deallocating and then accidentally using the deallocated variable is a "use after free".

On a good day, your program crashes harmlessly.

On a bad day, it can lead to remote code execution. A search for "use after free" CVEs found that they are *everywhere*.

C++

```c
#include <stdio.h>
#include <stdlib.h>

struct MyClass {
    int data;
};

int main() {
    MyClass * a = new MyClass();
    a->data = 5;
    delete(a);
    printf("%d", a->data);
    return 0;
}
```

No warnings with -Wall

Printed “16160336655” on my machine.
Segmentation Fault in the online C playground.

You create an object and delete it. Then you try to use it.

This is a really common problem in C (with malloc/free) and C++.

Modern C++

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <memory>

struct MyClass {
    int data;
};

int main() {
    auto a = std::make_unique<MyClass>();
    a->data = 5;
    a.release();
    printf("%d", a->data);
    return 0;
}
```

Let's use a unique_ptr for a Modern C++ experience!

Explicitly releasing it *does not* prevent you from trying to use it.

The program segmentation faults and crashes.

This is uncommon, definitely harder to do by accident than a regular `new`/`delete`.

We're going to skip Go, because it is garbage collected and doesn't really let you delete anything. The Go Runtime will decide when you aren't using something and get rid of it at some point in the future.

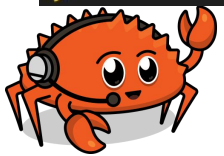
Rust Lifetimes and Ownership/Borrowing

```rust
struct MyClass {
    data: i32,
}

fn main() {
    let a = MyClass { data : 5 };
    std::mem::drop(a);
    println!("{}", a.data);
}
```

```rust
struct MyClass {
    data: i32,
}

fn main() {
    let a = Box::new(MyClass { data : 5 });
    std::mem::drop(a);
    println!("{}", a.data);
}
```



Neither program will compile.

The first just makes a structure on the stack, deletes it with “drop” and tries to access it.

The second makes it on the heap with a smart pointer (Box and unique_ptr are really similar), deletes it and tries to use it.

Rust tracks the lifetime and ownership of every variable - and won’t let you use after free.

There’s still no garbage collection, deletion remains deterministic.

C++ Use After Move

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <memory>

struct MyClass {
    int data;
};

void do_something(std::unique_ptr<MyClass> a) {
    printf("%d", a->data);
}

int main() {
    auto a = std::make_unique<MyClass>();
    a->data = 5;
    printf("%d", a->data);
    do_something(std::move(a));
    printf("%d", a->data);
    return 0;
}
```

Moving a variable hands it ownership to a function, and turns the previous variable into an “*xvalue*”: a “moved from” state that is valid, but *using* it is undefined behavior.

This program crashes with a segmentation fault. Who knows what is in that memory area now?

(Note: a wise C++ programmer would use a shared_ptr and reference counting if ownership is shared, or never touch a again if ownership of a unique_ptr is moved!)

Rust: Use After Move

```rust
struct MyClass {
    data: i32,
}

fn do_something(a: Box<MyClass>) {
    println!("{}", a.data);
}

fn main() {
    let a = Box::new(MyClass{ data: 5 });
    println!("{}", a.data);
    do_something(a);
    println!("{}", a.data);
}
```

An undecorated parameter in a Rust function *moves* the value into the function. “Move by default”

This will not compile. Rust has tracked ownership, seen that `a` has been moved - and will refuse to let you use it by mistake.

Rust gives you the power to manage your own memory, but takes away many of the dangers!

Overflow and Underflow

What is Overflow/Underflow?

Working with unsigned, 8-bit integers:

255 + 1 = 0! (You may have expected 256)

0 - 1 = 255! (You may have expected -1)

If you are parsing/calculating numbers that are used as an index to a buffer, you can trigger the issues we've already discussed.

Alternatively, *bad things can happen*:

Between 1985 and 1987, arithmetic overflow in the Therac-25 radiation therapy machines caused the death of at least six people from radiation overdoses.

August 2016: a casino machine at Resorts World casino printed a prize ticket of $42,949,672.76 as a result of an overflow bug.

It's not all bad: *Lamborghini American Challenge* on the Super Nintendo can underflow your cash, giving you $65,535,000 more than it would have had after going negative.

(Source: Wikipedia)

Overflows in C, C++ and Go

```c
#include <stdio.h>
#include <stdlib.h>
#include <cstdint>

int main() {
    uint8_t n = 255;
    printf("%d, ", n+1);
    n = 0;
    printf("%d\n", n-1);
    return 0;
}
```

Prints:
0
255

```go
package main

import "fmt"

func main() {
    var n uint8 = 255
    fmt.Println(n + 1)
    n = 0
    fmt.Println(n - 1)
}
```

Prints:
0, 255

Overflows in Rust

```rust
fn main() {
    let n : u8 = 255;
    println!("{}", n+1 );
    let n: u8 = 0;
    println!("{}", n-1);
}
```

Does not compile! The compiler figures out that n+1 and n-1 can be constants, notices that they are invalid - and won't let you compile the program.

```rust
fn main() {
    let mut n: u8 = 0;
    for _ in 0 .. 257 {
        println!("{n}");
        n += 1;
    }
}
```

So we make it a little harder for the compiler. This compiles.

In debug mode it prints 0 to 255 and then panics with *attempt to add with overflow*.

In release mode, it prints 0 to 255, and then 0 again.

ALWAYS check your code in DEBUG mode!

What if I WANT to Overflow?

Some algorithms require overflow or underflow to function. This isn't a bad thing. But wouldn't it be nice to be able to declare *what we want* - without needing to write:

```
result = a + b;
if(a > 0 && b > 0 && result < 0)
```

And equivalents everywhere?

(There are packages for Go, C and C++ to help with this)

```rust
fn main() {
    // Checked Arithmetic
    let n: u8 = 255;
    if let Some(n) = n.checked_add(1) {
        println!("{n}")
    } else {
        println!("Adding would result in overflow.")
    }

    println!("Wrapping: {}", n.wrapping_add(1));
    println!("Saturating: {}", n.saturating_add(1));
}
```

Prints:

```
Adding would result in overflow.
Wrapping: 0
Saturating: 255
```

In other words: you've opted in to clearly stating the behavior you expect. It's obvious to you, it's obvious to the next person to read your code.

Type Coercion

Converting between Types is Tricky

```c
#include <stdio.h>
#include <stdlib.h>
#include <cstdint>

int main() {
    uint16_t n = 260;
    uint8_t o = n;
    printf("%d = %d", n, o);
}
```

This C program prints:

260 = 4

N is a 16-bit unsigned integer. O is an 8-bit unsigned integer. There isn't room for the value 260, and C simply takes the last 8 bits and copies them over.

Note: this does generate a compiler warning in this simple form.

It's quite likely that this isn't what you intended.

Type Rejection

Go:

```go
package main
import "fmt"

func main() {
    var n uint16 = 260
    var o uint8 = n
    fmt.Println(n, " = ", o)
}
```

Rust:

```rust
fn main() {
    let n: u16 = 260;
    let o: u8 = n;
    println!("{n} = {o}");
}
```

Both Go and Rust do the right thing.

Assigning a 16-bit integer to an 8-bit integer is disallowed, with varying error messages.

Neither language will silently lose some of your data in this way.

The Downside of Type Rejection

```go
package main
import "fmt"

func main() {
    var n uint8 = 250
    var o uint16 = n
    fmt.Println(n, " = ", o)
}
```

```rust
fn main() {
    let n: u8 = 250;
    let o: u16 = n;
    println!("{n} = {o}");
}
```

Neither of these will compile.

We're doing something safe: we're moving from an 8-bit integer to a 16-bit integer. It's *absolutely guaranteed* that the first variable will fit into the second variable without data loss.

But - strict typing on both compiler's behalf won't let you do it.

So, Let’s Force The Issue!

```go
package main

import "fmt"

func main() {
    var n uint16 = 260
    var o uint8 = uint8(n)
    fmt.Println(n, " = ", o)
}
```

```rust
fn main() {
    let n: u16 = 260;
    let o: u8 = n as u8;
    println!("{n} = {o}");
}
```

Both languages let us ask the compiler to hold our beer, and convert anyway.

In Go, type(n) forces the conversion. In Rust, appending “n as type” does the same.

And… we’re back to **260 = 4**.

Going the other way - assigning a byte to a 16-bit integer works perfectly.

Wouldn’t it be nice to have a way to check that the conversion was safe?

Rust and into()

```rust
fn main() {
    let small : u8 = 255;
    let big: u16 = small.into();
    println!("{big}");
}
```

Rust implements the *Into* trait for safe conversions. This example compiles and works: you are *widening* the integer type, so it's always safe.

```rust
fn main() {
    let big : u16 = 512;
    let small: u8 = big.into();
    println!("{small}");
}
```

This doesn't compile. Rust **does not** implement *Into* for potentially lossy conversions.

Safeguard #1: Use `.into()` for conversions (or "from"). If it doesn't compile, the compiler will even suggest the next option.

Rust and try_into()

Sometimes, it's quite safe to narrow a type. A 32-bit integer containing "12" can quite safely be narrowed down to 16 or 8 bits.

You may need to know this at runtime!

You *could* liberally sprinkle your code with:

```
if n > u8::MAX as u32 { // Oops
```

OR, you could use `try_into()`

```rust
fn main() {
    let numbers: Vec<u16> = (250 .. 260).collect();
    let converted: Vec<u8> = numbers
        .into_iter()
        .flat_map(|n| n.try_into())
        .collect();
    println!("{converted:?}");
}
```

`try_into()` returns a `Result`, with `Ok(n)` if the conversion is safe - and an error if it isn't.

In this case, "flat_map" simply keeps the good results (and prints 250 through 255, the valid conversions). You can use the error however you need.

It also serves to acknowledge that you know there might be an issue here, and are taking it into consideration.

Clearly Indicating Intent

Oops, that function changed my variables!

Tell the Next Programmer - or Future You - What You Meant

We've touched on this a lot, so I wanted to make it explicit:

A **lot** of bugs happen at the boundaries between functions, or when you rely on a behavior (wrapping, type conversion) and the next programmer - or future you - come to the code some time later and don't see your intent.

You can help with this:

- You can use comments, but there's no promise anyone will read them. Self-documenting code is safer.
- Clearly flag wrapping, saturating, checked arithmetic by using the appropriate functions.
- Clearly flag type conversions with into() for always-safe and try_into() for fallible conversions. Don't use "as"!
- And this section - let the language help you by being pedantic.

Spot The Difference!

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>

void big_scary_fn(std::string fs) {
    fs += "B";
    std::cout << fs << ", ";
}

int main() {
    std::string s = "A";
    std::cout << s << ", ";
    big_scary_fn(s);
    std::cout << s << "\n";
}
```

It prints: **A, AB, A**

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>

void big_scary_fn(std::string &fs) {
    fs += "B";
    std::cout << fs << ", ";
}

int main() {
    std::string s = "A";
    std::cout << s << ", ";
    big_scary_fn(s);
    std::cout << s << "\n";
}
```

It prints: **A, AB, AB**

Did you notice the &? You passed a *reference* to your variable and let the writer of big_scary_fn do whatever they want to your variable! There's no change on the caller side - just the function signature.

Mitigating this in C++

```
void big_scary_fn(const std::string &fs) {
    fs += "B";
    std::cout << fs << ", ";
}
```

Pretend that big_scary_fn is an enormous function written by an outsourced company you've never met.

Hopefully your pucker factor went up!

You can mitigate this a bit in C++ by marking the reference as "const".

This function will no longer compile, because the function signature *promises* that you won't mutate the reference.

It's still up to you to trust that the next version won't quietly remove that keyword (or do evil things with `const_cast`) - but you can at least get a bit of safety by having the code indicate intent on the callee/function signature side.

Go Reduces the Problem

```go
package main
import "fmt"

func big_scary_fn(s string) {
    s += "B"
    fmt.Print(s, ", ")
}

func main() {
    var s string = "A"
    fmt.Print(s, ", ")
    big_scary_fn(s)
    fmt.Println(s)
}
```

Prints: A, AB, A

```go
package main
import "fmt"

func big_scary_fn(s *string) {
    *s += "B"
    fmt.Print(*s, ", ")
}

func main() {
    var s string = "A"
    fmt.Print(s, ", ")
    big_scary_fn(&s)
    fmt.Println(s)
}
```

Prints: A, AB, AB

But it's *much* nicer than the C++ equivalent. You explicitly had to make the string a pointer, and add the "&" to pass it by reference.

So you know that the big scary function *can* change your variables.

Rust Takes this Problem *Seriously*

```
fn big_scary_fn(s: &mut String) {
    *s += "B";
    print!("{s}, ");
}

fn main() {
    let mut s = String::from("A");
    print!("{s}, ");
    big_scary_fn(&mut s);
    println!("{s}");
}
```

To make this spot-the-difference work in Rust, you really have to be explicit about your intent:

- If you don't declare `s` as mutable (with "`mut`"), you can't change it all - in your own function or the scary one.
- Borrowing a reference with `&` doesn't give you read-write abilities, either. The function signature has to specify `&mut` - or it can't change the reference.
- When you borrow a variable to send to a function, *you* also have to specify `&mut` to grant mutability. If someone changes the "`big_scary_fn`" signature - your code won't compile.

Data Races

A Simple C++ Data Race

```cpp
#include <thread>
#include <iostream>

int main() {
    int counter = 0;
    std::thread t1([&counter]() {
        for (int i = 0; i < 1000000; ++i) {
            ++counter;
        }
    });
    std::thread t2([&counter]() {
        for (int i = 0; i < 1000000; ++i) {
            ++counter;
        }
    });
    t1.join();
    t2.join();

    std::cout << counter << std::endl;

    return 0;
}
```

This C++ program spawns two threads. Each thread increments “counter” 1,000,000 times. The program then uses join() to wait for the threads to complete and prints out the result.

It compiles and runs without warnings.

It prints: `1103767`
Or maybe: `1239576`
Or maybe: `1133526`

This is a *data race*. You are adding to counter in multiple threads. Adding isn’t a single-stage operation: you read the value, add one to it, and store the value. It’s quite likely that sometimes the threads will trip over one another.

The result: your careful calculation is now an over-complicated random number generator.

Let’s Go, Data Race

```go
package main

import (
    "fmt"
    "sync"
)

const LOOP_COUNTER = 10000
const N_THREADS = 100

var counter = 0
var wg sync.WaitGroup

func adder() {
    for i := 0; i < LOOP_COUNTER; i++ {
        counter += 1
    }
    wg.Done()
}

func main() {
    for i := 0; i < N_THREADS; i++ {
        wg.Add(1)
        go adder()
    }
    wg.Wait()
    fmt.Println(counter)
}
```

Go has the same problem. On the Go Playground,
it doesn’t give any warnings - if you enable data race detection it will find it.

Use your warnings!

(And try not to comment on the wait group syntax versus joining..)

Safe Rust, No Data Races

```rust
use std::thread;

const LOOP_COUNTER: usize = 10000;
const N_THREADS: usize = 100;

/// Safety: Hold my beer
fn main() {
    static mut COUNTER: usize = 0;

    thread::scope(|scope| {
        for _ in 0.. N_THREADS {
            scope.spawn(|| {
                for _ in 0.. LOOP_COUNTER {
                    unsafe {
                        COUNTER += 1;
                    }
                }
            });
        }
    });

    unsafe {
        println!("{COUNTER}");
    }
}
```

Just in case you want to have the same error in Rust, you can. You have to turn off all of the safety features with the `unsafe` keyword to do it. Every access to the global mutable `COUNTER` is unsafe, because a globally mutable variable isn't exactly a great idea in a multi-threaded environment.

Without adding "`unsafe`" everywhere, Rust *will not compile your data race*.

Safety with Atomics

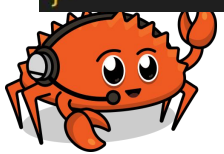
```rust
use std::{thread, sync::atomic::{AtomicUsize, Ordering}};

const LOOP_COUNTER: usize = 10000;
const N_THREADS: usize = 100;

fn main() {
    let counter = AtomicUsize::new(0);

    thread::scope(|scope| {
        for _ in 0.. N_THREADS {
            scope.spawn(|| {
                for _ in 0.. LOOP_COUNTER {
                    counter.fetch_add(1, Ordering::Relaxed);
                }
            });
        }
    });

    println!("{}", counter.load(Ordering::Relaxed));
}
```



All 3 languages will let you write a safe version using atomics.

An “atomic” variable uses CPU intrinsics to ensure that the read/update/write cycle is “atomic” - performed in one, uninterrupted action.

Note: I personally think C++ has the nicest syntax for this, you can just declare an atomic<int> and increment it.

Rust is the only language of the three that will *force* you to use an atomic without invoking unsafe code.

Just use a Mutex!

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int main() {
    std::mutex mutex;
    int counter = 0;
    std::thread t1([&counter, &mutex]() {
        for (int i = 0; i < 1000000; ++i) {
            std::lock_guard<std::mutex> guard(mutex);
            ++counter;
        }
    });
    std::thread t2([&counter, &mutex]() {
        for (int i = 0; i < 1000000; ++i) {
            std::lock_guard<std::mutex> guard(mutex);
            ++counter;
        }
    });
    t1.join();
    t2.join();

    std::cout << counter << std::endl;

    return 0;
}
```

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

type SafeCounter struct {
    mu sync.Mutex
    v  map[string]int
}

func (c *SafeCounter) Inc(key string) {
    c.mu.Lock()
    c.v[key]++
    c.mu.Unlock()
}

func (c *SafeCounter) Value(key string) int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.v[key]
}

func main() {
    c := SafeCounter{v: make(map[string]int)}
    for i := 0; i < 1000; i++ {
        go c.Inc("somekey")
    }

    time.Sleep(time.Second)
    fmt.Println(c.Value("somekey"))
}
```

Go and C++ both includes Mutexes. Lock the mutex, every other access has to wait.

C++ uses RAII to make it hard to forget to release your lock.

Go requires you to call “unlock”.

Notice one very important thing.

NEITHER language **makes** you use the Mutex, or reminds you that you forgot to do so. The mutex is a separate entity, and it’s up to you to remember to lock/unlock it.

Rust Mutex

```rust
use std::{thread, sync::Mutex};

const LOOP_COUNTER: usize = 10000;
const N_THREADS: usize = 100;

struct Counter(usize);

fn main() {
    let counter = Mutex::new(Counter(0));

    thread::scope(|scope| {
        for _ in 0.. N_THREADS {
            scope.spawn(|| {
                for _ in 0.. LOOP_COUNTER {
                    let mut lock = counter.lock().unwrap();
                    lock.0 += 1;
                }
            });
        }
    });


    println!("{}", counter.lock().unwrap().0);
}
```

Here's a Rust version with a Mutex.

Notice that the Mutex is *wrapped around* the protected variable, and you have to lock() the variable before you gain access to the contents.

Rust **won't** let you forget to lock your synchronization primitive. You **can't** access the contents without locking, and you **can't** compile your program without some sort of synchronization primitive!

Rust also uses the RAII pattern, similar to C++ - so your lock is released as soon as it goes out of scope.

Interior Mutability

Rust is very strict, but offers some escape hatches. Unsure when a variable will end its life (maybe it goes to multiple threads?)? You can use an Arc to thread-safely reference count.

Want Mutex-type protection on individual fields rather than the whole structure? Interior mutability is the answer:

```
struct InteriorMutable {
    safe_1: Mutex<String>,
    safe_2: RwLock<String>,
    safe_3: AtomicU32,
}

type SharedInteriorMutable = Arc<InteriorMutable>;
```

Now an instance can be shared between threads. Each field can be locked independently. And the borrow-checker will *still* remind you if you added another field and forgot to protect it.

This works because of the “Sync Autotrait”. Any structure for whom *all* of its fields are synchronize-safe automatically becomes “Sync”.

Sync structures still obey the “one borrow at a time” rule, but can enforce that check with a runtime guard like a Mutex, RwLock, or atomic.

You really can have your cake, and eat it.

Things I Didn’t Cover

Safe Rust doesn’t have null pointers.

Rust functions are usually designed to avoid problems. For example, “Command” separates the command and the parameters, reducing the “filename;ls” problem with programs that just shell out to cat.

Rust doesn’t have exceptions, but uses sum types for error handling. You *have* to at least acknowledge that an error was possible to use the result of a function.

Rust enforces strict aliasing rules, no two references point to the same area of memory at the same time.

Rust strong-typing makes it easy to avoid unit conversions, and to wrap unsafe code in safe facades.

You almost-always have the “unsafe” escape hatch if you *need* to interact with unsafe code (code written outside of Rust), hardware, etc.

Some General Advice

You can write perfect code in C, C++, Go and Rust.

Each language in turn offers more seat-belts for safety. EVERY language decision is an engineering compromise: use the one that fits what you need.

Go is much simpler to learn, and a great fit for a lot of programming.

Rust is harder to learn, safer for really heavy-duty concurrency, and remains close to the metal.

Every language also offers you the opportunity to shoot yourself in the foot. Rust and Go make it harder - but really bad code in any language is going to give someone a bad day.

Rust isn't perfect. Soundness holes exist (check out std::mem::forget and some of the safe transmute calls!) - but Rust does a good job of helping you not shoot yourself in the foot!

Let Us Help!

Developing complex systems with great performance and high-confidence safety is hard. Go and Rust make it easier - but “doing it right” requires a learning curve.

At Ardan Labs, we offer training, consulting and staffing services to help you make the best technology stack decisions, and tailor your solution to your customer’s problems.

QA