**Rust is memory safe, but at what cost?**

[[](https://medium.com/@apivan.tuntakurn?source=post_page-----3f309110b2a7--------------------------------)](https://medium.com/@apivan.tuntakurn?source=post_page-----3f309110b2a7--------------------------------)

[Apivano](https://medium.com/@apivan.tuntakurn?source=post_page-----3f309110b2a7--------------------------------)

·

Follow

8 min read

·

Dec 7

116

9

**Introduction**

It’s important to focus on practical applications rather than unattainable ideals, understanding that the key difference between designing and practice lies in the details encountered during implementation.

For instance, in programming languages, ideally const should be default. But in practice, local variables, almost all the time are not const.

I believed that C++ is overcomplicated, and really need a total replacement. But I believed Rust is not a good solution. I personally looking forward to Cppfront or Carbon than Rust

This document looks at the memory management as it is a fundamental difference between C++ and Rust.

**Do we really need to store a constant reference?**

Whenever you pass a function that uses a constant reference, it usually for reading the input to perform some operation. It might return a new transformed output of it, or perhaps it will return a (reference) view of it.

// RUST ✅ SAFE AND COMPILE  
  
  
fn longest\_word(x: &String, y: &String) -> String {  
 if x.len() > y.len() {  
 x.clone()  
 } else {  
 y.clone()  
 }  
}  
  
fn main() {  
 let magic1 = String::from("abracadabra!");  
 let magic2 = String::from("shazam!");  
  
 let result = longest\_word(&magic1, &magic2);  
 println!("The longest magic word is {}", result);  
}

//C++ ✅ SAFE AND COMPILE  
#include <iostream>  
#include <string>  
   
std::string longest\_word(const std::string& x, const std::string& y) {  
 if (x.length() > y.length()) {  
 return x;  
 } else {  
 return y;  
 }  
}  
  
int main() {  
 std::string magic1 = "abracadabra!";  
 std::string magic2 = "shazam!";  
  
 std::string result = longest\_word(magic1, magic2);  
 std::cout << "The longest magic word is " << result << std::endl;  
  
 return 0;  
}

But who would actually store the constant reference in such a way that it goes out of scope?

// C++ ☣️ UNSAFE AND COMPILE ☣️  
struct BadStruct  
{  
 BadStruct(int& somethingwrong):m\_thiswillbebad(somethingwrong)  
 {  
 }  
 const int& m\_thiswillbebad; // how common is this?  
};  
   
int main()  
{  
 int\* x = new int(10);  
 auto w = BadStruct(\*x); // still ok   
   
 std::cout << w.m\_thiswillbebad; // still ok  
   
 delete x;  
 std::cout << w.m\_thiswillbebad; // ☣️ bad use-after-free ☣️  
   
}

**Linear Code: Do We Really Need Borrow Checking?**

From our previous discussion, here is what we usually do when working with linear code that creates a view of transformed data.

// C++ ✅ SAFE AND COMPILE  
#include <iostream>  
#include <ranges>  
#include <vector>  
  
int main()  
{  
 auto const ints = { 0, 1, 2, 3, 4, 5 };  
 auto even = [](int i) { return 0 == i % 2; };  
 auto square = [](int i) { return i \* i; };  
  
 // Create reference view  
 auto ref = ints | std::views::filter(even) | std::views::transform(square);  
  
 // Use it  
 for (int i : ref)  
 std::cout << i << ' ';  
  
 // Or clone it  
 auto cloned\_vector = std::vector<int>(ref.begin(), ref.end());  
  
}

// RUST ✅ SAFE AND COMPILE  
fn main() {  
 let ints = [0, 1, 2, 3, 4, 5];  
 let even = |i: i32| i % 2 == 0;  
 let square = |i: i32| i \* i;  
   
 // Creating an iterator that filters and maps the values  
 let ref\_view = ints.iter().filter(|&&i| even(i)).map(|&i| square(i));  
   
 // Using the iterator directly  
 for i in ref\_view {  
 print!("{} ", i);  
 }  
 println!();  
   
 let ref\_view\_again = ints.iter().filter(|&&i| even(i)).map(|&i| square(i));  
 let cloned\_vector: Vec<\_> = ref\_view\_again.collect();  
   
 // Optionally, print the cloned vector  
 for i in cloned\_vector {  
 print!("{} ", i);  
 }  
 println!();   
}

**The single mutable reference rule presents a real challenge.**

Just be aware that the most Rust examples are **constant references**. In real life, references are **not always constant**. We use mutable references because we want to refer to same variable. So that it can be modified from various places, ensuring it receives the same updates, don’t we? If it not the case, then why not just use value type?

//RUST ❌ SAFE but NOT Compile   
//Just because the compiler doesn't know something is safe,  
//doesn't mean it isn't safe.  
fn main() {  
 let mut data = 10;  
   
 let ref1 = &mut data; // First mutable reference to `data`  
 let ref2 = &mut data; // This line will cause a compile-time error  
   
 \*ref1 += 1; // We're using the first mutable reference to modify `data`  
   
 println!("data: {}", data);  
}

//C++ ✅ SAFE AND COMPILE  
int main() {  
 int data = 10;  
   
 int& ref1 = data; // First mutable reference to `data`  
 int& ref2 = data; // Second mutable reference to `data`  
   
 ref1 += 1; // Modifying `data` through the first reference  
   
 std::cout << "data: " << data << std::endl;  
   
 return 0;  
}

The borrowing semantics in Rust are not inherently flawed. However, the coding benefit of borrowing largely depends on the where it is return; Ideally, resource borrowing should occur only during read/write operations of a entity and should be usable immediately thereafter.

It’s true that in the Rust code mentioned earlier, adding a scope can make the borrowing more thin. However, this becomes challenging when dealing with non-linear scopes.

**Non-linear Code: Addressing the Event Loop Challenge.**

Here’s an example of fundamental code operating within an event loop. The count variable is captured by reference in both schedulers. Thanks to the strand mechanism, both callbacks can read and write the same variable without encountering concurrency issues.

//C++ ✅ SAFE AND COMPILE  
#include <iostream>  
#include <boost/asio.hpp>  
   
using namespace boost::asio;  
using namespace std;  
   
int main() {  
 io\_context io;  
   
 int count = 0;  
   
 steady\_timer t1(io, chrono::seconds(1));  
 steady\_timer t2(io, chrono::seconds(2));  
   
 // Using the same strand for both timers  
 io\_context::strand strand(io);  
   
 // Set the timer to fire after 1 second, using the strand  
 t1.async\_wait(strand.wrap([&count](const system::error\_code& /\*e\*/) {  
 cout << "Timer 1: " << ++count << endl;  
 }));  
   
 // Set the timer to fire after 2 seconds, using the strand  
 t2.async\_wait(strand.wrap([&count](const system::error\_code& /\*e\*/) {  
 cout << "Timer 2: " << ++count << endl;  
 }));  
   
 io.run();  
   
 return 0;  
}

This is not possible in Safe Rust.

You must either utilize RC (Reference Counter, similar to shared\_ptr), RefCell (also a reference counter) or refactor the communication channel between asynchronous functions to ensure the compiler recognizes its safety.

//RUST ✅ SAFE AND COMPILE, but at what cost?  
use tokio::time::{sleep, Duration};  
use tokio::sync::mpsc;  
   
#[tokio::main]  
async fn main() {  
 let (tx, mut rx) = mpsc::channel(10);  
   
 // Timer 1: Set to fire after 1 second  
 let tx1 = tx.clone();  
 tokio::spawn(async move {  
 sleep(Duration::from\_secs(1)).await;  
 tx1.send("Timer 1 fired").await.unwrap(); // 💰 Here we go,   
 // 💰 transforming it into a common  
 // 💰 , dynamically-typed 'lost-type'  
 // 💰 string message. });  
   
 // Timer 2: Set to fire after 2 seconds  
 let tx2 = tx.clone();  
 tokio::spawn(async move {  
 sleep(Duration::from\_secs(2)).await;  
 tx2.send("Timer 2 fired").await.unwrap();  
 });  
   
 // 💰 Counter task  
 tokio::spawn(async move {  
 let mut count = 0;   
 while let Some(message) = rx.recv().await {  
 count += 1;  
 println!("{}: Counter is now {}", message, count);  
 }  
 }).await.unwrap();  
}

The code below represents a **typical day’s coding in C++, C#, JavaScript, Python, and almost every other programming language**. However, this poses a significant challenge for Rust

// C++ ✅ SAFE AND COMPILE  
 this->m\_RaiiDtorDisconnection += this->onclick().connnect([this](auto&& at) {  
  
 this->m\_textLabel.text = "clicked" + at.x + at.y;  
 this->m\_textLabel.state.unfocus();  
 });  
  
  
 this->m\_RaiiDtorDisconnection += this->onkeydown().connnect([this](auto&& k) {  
  
 if (key == key.Enter)  
 {  
 this->m\_textLabel.clear()  
 }  
 else if (key == key.U)  
 {  
 this->m\_textLabel.style.addUnderline();  
 }  
  
 });

Same issue with coroutine code

// C++ ✅ SAFE AND COMPILE  
Task<> deleteUsers(const std::string& userId)  
{  
 auto userDatabase = app().getDatabaseClient();  
 co\_await userDatabase->executeAsync("DELETE FROM users WHERE user\_id = $1", userId);  
 auto transectionDatabase = app().getTransectionClient();  
 co\_await transectionDatabase->executeAsync("DELETE FROM transactions WHERE user\_id = $1", userId);  
}  
   
Task<> identifyLowScoreUsers()  
{  
 auto database = app().getDatabaseClient();  
 auto userRecords = co\_await database->executeAsync("SELECT user\_id FROM users "  
 "WHERE score < 5;");  
 for(const auto& user : userRecords)  
 co\_await deleteUsers(user["user\_id"].as<std::string>());  
}

**Compile-time borrow checker is practically a move-only value semantics with some syntactic sugar**

Since the reference can only be borrowed one entity at a time. It’s close to value semantics than reference semantics. Rust borrow checker simply just a syntactic sugar for returning the value to where it came from. The benefit is this is that it is totally avoiding use-after-move. Below is the C++ equal valent code of borrow reference.

// C++ ✅ SAFE AND COMPILE  
auto SomeFunc = [](Xobj&& x) -> Xobj   
{  
 return x;  
}  
   
auto x = Xobj{};  
x = SomeFunc(std::move(x));

// RUST ✅ SAFE AND COMPILE  
fn SomeFunc(x: &mut Xobj) {  
 x  
}  
  
fn main() {  
 let mut x= CreateXobj();  
 SomeFunc(&mut x);  
}

**Rust making memory safe by prohibit using unmanage references just like every other language**

Like every other garbage collection language (such as C#, JavaScript, Python, etc.), Rust simply prohibit the use of unmanage references, allowing only reference counting, which are the least efficient form of garbage collection.

**It’s not the borrow checker that makes Rust memory safe; rather, it’s the prohibition of using unmanaged references that ensures memory safety.**

**Monolithic borrow checker do more harm than good**

The code below is legitimate and does not deal with manual memory coding by human. The LinkedList (or std::list in C++) doesn’t cause reallocation. However, the borrow checker falsely flags an error when pushing items to a linked list while iterating over it. An opt-in borrow checker might be useful in some cases, but enforcing this by default is not a good idea.

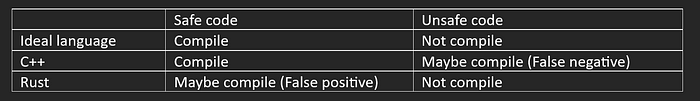
//RUST ❌ SAFE but NOT Compile  
use std::collections::LinkedList;  
   
fn main() {  
 let mut list = LinkedList::new();  
 list.push\_back(1);  
 list.push\_back(2);  
 list.push\_back(3);  
   
 for &item in &list {  
 if item % 2 == 0 {   
 list.push\_back(item + 10);   
 }  
 }  
   
}

**But are those C++ examples potentially unsafe?**

Yes, but code can only become unsafe if someone modifies it and rebuilds it.

So, the ideal programming language should:

* Compile when the code is safe
* Not compile when the code is unsafe



C++ Compiler errors, warnings, and linters help reduce the compilation of unsafe code. However, I’m aware that aggressive linting, similar to Rust’s approach, might prevent the compilation of safe code.

It’s a difficult choice between preferring a permissive or a regulative approach.

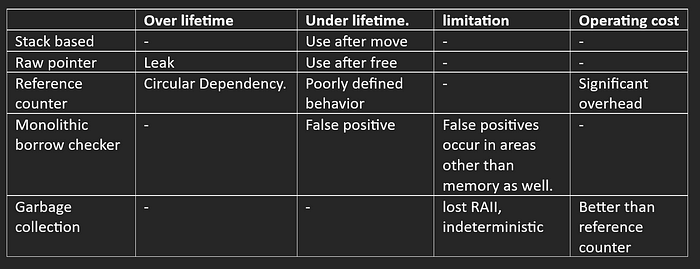
**We need permissiveness to get thing done and regulation to get thing safe.**

**Reference counting is a bad solution.**

Reference counters have a performance cost and can introducing bug if the weak references was used just to counter circular dependency.

if(!weak\_x.lock()){  
// What should be done in this situation?  
  
// Let not do anything?  
// Is this consider an error?  
// Should I terminate the application?  
  
// Poorly defined behavior isn't much better than undefined behavior  
}

Comparison of each memory manage approach



Rust monolithic unsafe combined with monolithic borrow checker is the problem.

Rust unsafe keyword doesn’t tell what is really need to be careful about it. If many unsafe keywords have to be used for the already-safe-code, then unsafe will lost its meaning.

If the rule is to minimize the used of unsafe keywords as much as possible and ok to trade with poorer performance and convoluted code, maybe garbage collection language is a better choice.

