CS263: Runtime Systems
Crust (C++ vs. Rust)

Rakshith Gopala Krishna
PERM: 9870890

Subramaniyam Shankar
PERM:9880527

Prof: Chandra Krintz

June 9, 2020

1 Problem Statement

In-depth analysis and comparison of different features in Rust and C++, to find which language is more preferable over the other for a particular systems programming use case.

1.1 Motivation

Rust is one of the modern systems programming language that markets itself as "A language empowering everyone to build reliable and efficient software" [26]. Stack Overflow's 2020 [29] developer survey puts Rust as the most loved technology. Rust's popularity can also be attributed to it's rich type system, memory management with the help of it's ownership model for variables. It also tries to provide better tools for managing big projects via Crates, which is its package system similar to npm for node.js. From a systems programming language standpoint, Rust aims to also avoid undefined behavior which is more common in traditional languages like C/C++. All these claims by Rust makes it a very interesting language from a systems programming perspective. Our goal will be to answer the following questions:

- What features of Rust made it so popular as a systems programming language?
- Are these features (or their variants) already being provided by C/C++?
- Investigate deeply, the claims made by Rust regarding memory safety and concurrency.

2 Ownership and Move Semantics

2.1 Rust

Ownership is a unique feature to Rust that helps it a lot in managing memory properly. So, each value in Rust has a variable called its owner and whenever the owner(variable) goes out of scope the resources associated with the value are also dropped. Also, a value(resource) can have only one owner at a time. This can be seen from code snippet 1 in the appendix.

Since there can be only one owner at a time, a move occurs implicitly when one tries to assign a variable to another. To enable copy instead of move, the type should implement `Clone` trait or call the `clone()` function. The code to display this behavior is listed in code snippet 2 in the appendix.

Rust is very strict when it comes to move semantics as it does not allow one to use a variable that has already been moved. Move semantics helps in performance over copying as it will be seen in the subsection 2.3.

2.2 C++

C++ introduced the concept of ownership with its smart pointers, in particular, `unique_ptr`. We discuss smart pointers in detail in section 3. A `unique_ptr` owns the resource it is pointing to and there can only be a single unique pointer to that particular resource at a time. C++ enforces this at compile time by disabling the copy constructor which will create new copies of the `unique_ptr`. The only way to create a new `unique_ptr` is to either instantiate them from raw pointers or by invoking the move constructor which essentially transfers the ownership of a resource from one smart pointer to the other.

Move semantics in C++ allow objects to be moved (pointer re-assignment) instead of copying them over (this is what used to happen traditionally before C++11). This proves to be very useful when dealing with objects with a large memory footprint. Move constructor is usually called when we pass an R-value reference which is created when the `std::move` function is called. R-value reference (essentially a reference to an R-value [1]) is a new entity which was added in C++11. This can be observed in code snippet 3 in the appendix.

2.3 Comparison

As mentioned in the previous subsections, the presence of move semantics have been particularly helpful performance wise. Instead of wasting extra time and resources in creating a temporary variable by copy, using move saves time. To test out the comparison between move and copy, a vector(which is a collection of variables that store the values on the heap (unlike local array)), containing 10,000,000 integer values were moved and copied in both Rust and C++. The code for this can be found in copy_vec.rs [4], move_vec.rs [21], vector_copy.cc [30] and vector_move.cc [31] files. It can be seen that move indeed takes less time as compared to copy.

[1] R-value is an identifier whose memory location cannot be determined. One such rvalue is a literal.

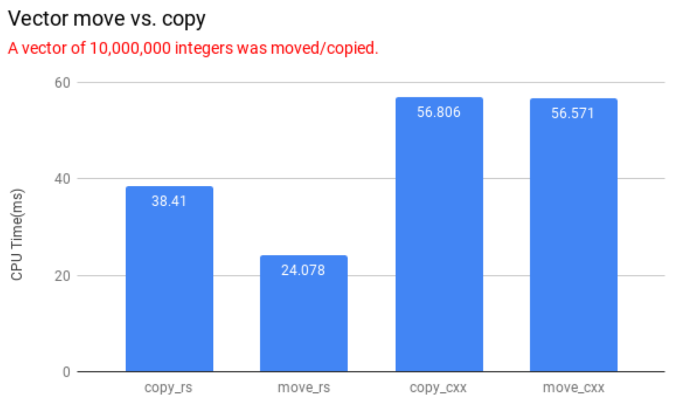


Figure 1: Ownership and move semantics comparison in Rust and C++

3 References and Smart pointers

3.1 Rust

When one wants to just borrow a value without taking ownership of the variable (i.e. try not to consume the other variable via move as described previously), it is done through references which is a pointer to the actual value. This is particularly useful in defining functions which do not take ownership of the variable. Refer code snippet 4 in the appendix for an example. References in Rust have the following **invariants**

1. There can be **only one mutable reference** (the kind of reference with which one can mutate data) to a value at the same time

   or

   Any number of immutable references to a value at the same time

2. References should always point to a **valid memory** (i.e. cannot have dangling references)

These invariant are enforced at both compile time and runtime.

While references borrow a value, **Smart Pointers** on the other hand, not only are references but also take ownership of the value they are pointing to. They also store the values on the heap rather than on the stack.

`Box<T>` is a smart pointer that allows one to store a value in the heap and take ownership of the value. It can be used as both immutable (cannot change the value) or mutable (can change the value). The exact code snippet displaying this behavior is listed in code snippet 5.

`Rc<T>` is another smart pointer used to create an immutable pointer whose ownership will be shared across multiple owners. `Rc` stands for reference counted, which means that it will only drop the resources of the pointer only when its reference count becomes zero (i.e. only when the last owner goes out of scope). This pointer is not thread safe.

`RefCell<T>` is another smart pointer which can have only one owner just like `Box<T>`. The main difference between them is that, `RefCell<T>` provides interior mutability. So, if it is wrapped around an immutable pointer like `Rc<T>`, it can still be mutated. So `Rc<T>` and `RefCell<T>` are usually used together, with one wrapping the other depending upon the requirement. Using their combination might feel like it is ignoring the invariants of Rust (as one can mutate the value using any one of the pointers and multiple such pointers exist at the same time). This kind of construct still does not violate the invariants of Rust. Even though multiple such pointers exist in the same scope, but at runtime only can be actually used at a time to modify the data, preserving the invariants. It provides programmers more flexibility and is useful in creating structures like graphs, where a child node can have multiple parents and each parent can mutate the values inside the child node.

One problem with the combination of `Rc<T>` and `RefCell<T>` is that they can be used to create reference cycles, which leads to a memory leak as demonstrated in the rc_mem_leak.rs [24] (the reference count will be more than one because of the children nodes pointing back). To overcome this, a variant of `Rc<T>` is used which is called the `Weak<T>` pointer. A resource will be cleared even if it's weak reference count is non zero, as soon as it's normal reference count (also called strong count) becomes zero. So, in an example of a graph all the parent pointers (in the children nodes) will be modeled with the help of `Weak<T>`.

3.2 C++

As the name suggests, smart pointers automatically take care of memory of the objects they represent. The programmer is freed from the responsibility of manually de-allocating heap-allocated resources. Note that using raw pointers is not bad, but it makes it significantly harder to determine who is the owner of the resource and when it should be freed when there are multiple raw pointers to the same resource. The idea is to encapsulate the raw pointers inside stack allocated smart pointers. As soon as the smart pointers go out of scope, the resource they represent is freed from the heap. This way the resources are managed automatically. `unique_ptr<T>` discussed previously is a smart pointer.

`shared_ptr<T>` are reference counting pointers. They consist of two segments internally. First segment manages the resource it points to, and the second segment manages the number of references to the said resource. Accesses to the second segment of the shared pointer is thread-safe. As the name suggests, the ownership of the resource is shared between many pointers. The reference count is increased or decreased according to the number of shared pointers managing the resource. As soon as the last pointer goes out of scope, the reference count becomes zero which triggers the de-allocation of the resource. The code to illustrate this behavior is listed in code snippet 6.

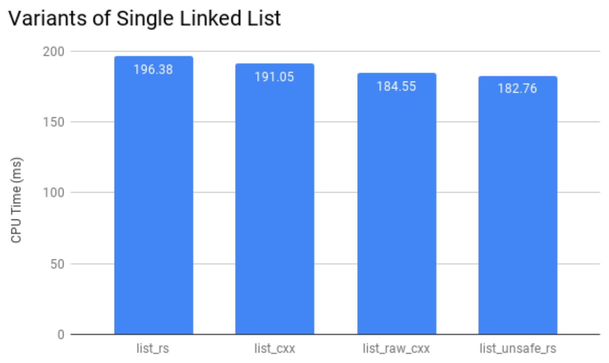
One thing to note here is that since this is a reference counting smart pointer, circular references with shared pointer means that the reference count will never go to zero and therefore the resource is never freed, leading to a memory leak. This is demonstrated in shared_ptr_mem_leak.cc [27].

Weak pointers can be used to mitigate this issue. `std::weak_ptr<T>` is a smart pointer that holds a non-owning reference to a resource which is managed by `std::shared_ptr<T>`. A weak pointer only models a temporary ownership and the decision to free the original resource is never influenced by the scope of the weak pointer. This is why weak pointers can be used to break reference cycles.

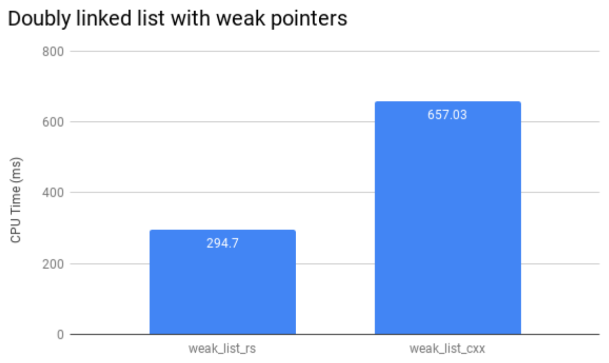
3.3 Comparison

We used the smart pointers mentioned above and also used raw pointers to create a linked list containing 5,000,000 nodes. We traverse to the last node and modify and print the its value. linked_list.rs [15] implements the logic of linked list using `Box<T>`. list_unique.cc [18] uses `std::unique_ptr<T>` to implement the same. linked_list_unsafe.rs [16] uses raw pointers and unsafe rust to create the linked list and list.cc [17] also uses raw pointers to achieve the same. From Fig 2a it can be seen that raw pointer implementation of linked list in Rust is the fastest, but the differences are very small. All of these implementations do not contain any memory leaks.

weak_list.rs [33] uses a combination of `Weak<T>`, `Rc<T>` and `RefCell<T>` to create a doubly linked list without any memory errors. weak_list.cc [32] uses `std::weak_ptr<T>` and `std::shared_ptr<T>` to achieve the same result. Both the implementations create a doubly linked list of 5,000,000 nodes and update and print the last node. The Rust implementation of doubly linked list is significantly faster than the C++ counterpart (Fig 2b). When inspected in Valgrind, the C++ implementation was doing twice the amount of allocs and free as compared to the Rust version, which we believe might be the reason for making C++ implementation slower.



(a) Singly linked list implemented with both raw pointers and smart pointers in Rust and C++



(b) Doubly linked list implemented with weak pointers in both Rust and C++

Figure 2: Various linked list implementations using smart and raw pointers in Rust and C++

4 Zero cost abstractions (Iterators and closures)

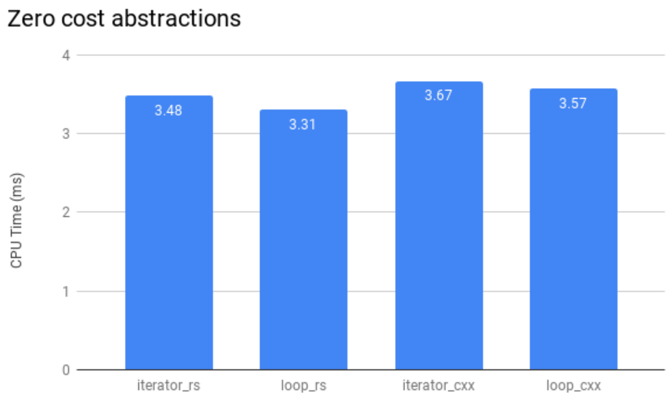
Rust tries to follow the concept of zero cost abstractions very dearly. Rust has **closures**, which are nothing but lambda functions which can be used to declare small functions in place instead of writing a whole new function and calling it. A closure has the benefit of capturing the variables enclosing the closure, but Rust requires to move those variables to the closures, before it can use them. This is particularly useful when creating threads.

Iterators also exist in Rust. They are used to loop or iterate over a collection of data, performing same kind of action on each element.

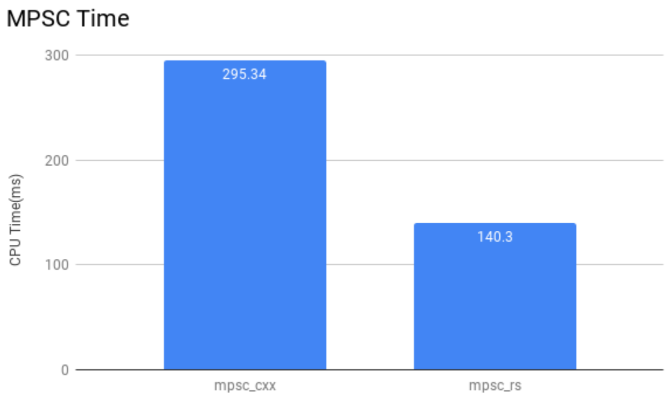
C++ also has these zero cost abstractions. Lambda expressions are anonymous inline functions. The parameters to the function can be captured either by value or by reference. Iterators display the exact behavior as in Rust.

4.1 Comparison

zero_cost_iterators.rs [35] implements a small operation using iterators and closures and zero_cost_loop.rs [37] mimics the same logic using traditional loops and if conditions. zero_cost_iterators.cc [34] implements the same logic in terms of iterators and closures and zero_cost_loop.cc [36] does the same with traditional loops. From figure 3a it can be seen that the time difference between an implementation of the same logic using iterators and closures and one without them, is actually very close in both Rust and C++. Hence, it can be concluded that both of these languages implement the zero-cost abstractions pretty well.



(a) Zero cost abstractions in Rust and C++



(b) Concurrency in Rust and C++

Figure 3: Zero cost abstractions and concurrency in Rust and C++

5 Concurrency

Many a times it is required to share data among threads in a concurrent program. There are actually two ways in which data can be shared across threads, either via **Passing Messages** or by **Sharing same memory location** across threads.

5.1 Sharing same memory location

In Rust threads are spawned easily by the `thread::spawn` keyword, which takes in a function or a closure. All environment variables required by that thread, have to be moved to that thread, transferring the ownership of those variable to that thread, making them useless in the parent thread. Applying restrictions like this at compile time removes a lot of potential errors that can occur. So to share the same memory location across multiple threads, a different kind of **Smart Pointer** is used called the `Arc<T>`. It is different from `Rc<T>` in the way that the increment of the reference count in `Arc<T>` happens atomically. But `Arc<T>` creates only an immutable pointer. Just like `RefCell<T>`, its thread safe counterpart is `Mutex<T>`. `Mutex<T>` uses locks for synchronized access of the memory location avoiding data races. Hence, a `Mutex<T>` wrapped inside an `Arc<T>` is usually used to share the same memory location across different threads. But, using their combination naively can lead to deadlocks. Rust does not do anything special to prevent deadlocks.

In C++, shared pointers can be used to share the same resource between multiple threads. As mentioned earlier, the shared pointer has two segments. The control block keeps track of the reference counts of the shared resource. This segment is thread-safe and we do not need to synchronize accesses to this part when we create new shared pointers to the resource in separate threads. However, accesses to the data segment which actually refers to the actual memory location of the resource is not thread-safe. It is the programmer's responsibility to synchronize accesses to the data segment of a shared pointer to prevent data race conditions and undefined behavior.

5.2 Passing messages

The second way to share data between threads is by **Passing Messages** across threads. Buffers are used to share data. Data on it is populated by the **Producers** (which can be one or many) and consumed by a single **Consumer**.

Rust provides an in built library called `mpsc`, which is short for "multiple producer single consumer". There are two kinds of channels, one is **asynchronous** and does not block on `send` as it internally has an "infinite buffer"[25] and the other (that we actually tested) is the **synchronous** channel type. The synchronous type has a buffer of limited length. In this case `send` will not block until the buffer is full and `recv` blocks until there is some message in the buffer. As soon one sends a variable to another thread via the buffer, the ownership of that variable is transferred to the other thread making that variable invalid in the sender thread. Hence, implying this kind of restriction helps avoid many errors.

C++ does not provide an MPSC library like Rust does, so we implemented a small MPSC program to mimic the Rust library. We implemented three classes, Buffer, Producer and Consumer. The Buffer class provides functions to add and remove elements from a double ended queue data structure. The Producer class provides a function to produce random numbers and invokes `Buffer.insert()` function to insert the produced numbers into the buffer. Consumer class provides a function to call the `Buffer.remove()` function to get the numbers from the front of the queue data structure. We used `std::mutex` to synchronize accesses to the queue, and `std::condition_variable` to signal other threads as and when the queue becomes full or empty.

5.3 Comparison

To test the memory sharing model for concurrency, 10 threads are created which update a particular memory location atomically and the main thread prints the value inside of that memory location after waiting for all the threads to finish. arc_thread.rs [1] implements the above mentioned logic using `Arc<T>` and `Mutex<T>`. shared_ptr_thread.cc [28] implements the same logic using `std::shared_ptr<T>`.

To test the message passing model, 10 threads are created, each of which create 100 random numbers and the main thread which is the consumer keeps on consuming the data. mpsc.rs [23] implements the above mentioned logic in Rust and mpsc.cxx [22] in C++. From Fig 3b it can be seen that Rust implementation is faster than C++.

For both C++ and Rust none of our codes leak memory. All the strict rules and conditions which Rust imposes at compile time, make it a better choice than C++ for writing concurrent programs correctly.

6 Monomorphisation, Generics and Dynamic Dispatch

Rust also supports using Generics. One can write functions, structs and enums containing generic types. With help of one more feature called `Trait` (which are very similar to interfaces from Object Oriented Programming paradigm), one can achieve a lot of flexibility and functionality using generics. generic.rs [12] creates function that accepts any variable type which implements the `Debug` trait (which is required by the `println` function to be able to print them on the screen). That function is called with three variable types, that inherently implement the `Debug` trait, `i32` (int), `f64` (float) and `String`. Rust performs monomorphisation on the function and actually defines the function three times in the assembly output of the code, one for each of the type, which can be seen in the lines 4960 to 5074 in generic.s [11] (look for "print" keyword). Hence, Rust makes sure that there is no overhead in using Generics this way. For monomorphisation, Rust infers the type statically and also increases the size of the executable with every function call of different type. To avoid this, dynamic.rs [7] implements the same function, but this time it will be called dynamically and the type will be resolved only at runtime, thereby not making any extra copies of the same function. It is verified by looking at the assembly dynamic.s [8] from line 4890 to 4927.

We can observe the same effects of monomorphisation, generics and dynamic dispatch in C++ as well. The aforementioned logic for testing monomorphisation is implemented in generics.cc [13] and verified by the lines 163 to 277 in generics.s [14]. dyn_dispatch.cc [5] implements the dynamic dispatch behaviour and verified by the assembly dyn_dispatch.s [6] from line 29 to 54, which has only one definition of the function.

7 Index Unrolling

We also explored index unrolling of loops. This is more of a compiler feature rather than a language feature. It was seen that, the C++ compiler being more mature, gave programmers more control over which loop can be unrolled and which cannot be unrolled. loop.cc [19] implements a simple loop and the unrolling is verified by the corresponding assembly from lines 161 to 402 in loop.objdump [20]. Rusts compiler which internally uses LLVM, unrolls loops if they are smaller than a certain threshold but the programmer does not have much control over which loop will be unrolled or which won't. for_loop_unrolling.rs [9] implements a simple loop and is verified by the corresponding assembly from lines 82 to 226 in for_loop_unrolling.s [10]. Such optimisations and fine grained controls are better in C++ as compared to Rust.

Appendices

A Code snippets

```
fn main {
    let s = String::from("hello"); // s is valid from this point forward
    // do stuff with s
}   // this scope is now over, and s is no longer valid, call drop
```

Code Snippet 1: Ownership in Rust

```
fn main {
    let s1 = String::from("hello");
    let s2 = s1; // s1 is moved to s2
    println!("s1: {}, s2: {}", s1, s2); // gives an error as s1 moves to s2
}
```

Code Snippet 2: Move semantics in Rust

```
int main(){
    std::unique_ptr<int> u;
    int *p = new int(5);
    u=std::unique_ptr<int>(p);
    std::unique_ptr<int> u2;
    u2 = std::unique_ptr<int>(std::move(u)); // cannot invoke the copy constructor here
    std::cout<<u.get()<<std::endl;           // prints 0
    std::cout<<u2.get()<<" "<<*u2<<std::endl;
}
```

Code Snippet 3: Ownership with unique pointers in C++

```
fn take_ownership (s: String) -> i32 {
    s.len()
}
fn borrow_value (s: &String) -> i32 {
    s.len()
}
fn main () {
    let s1 = String::from ("Hello");
    println!("Length: {} ", take_ownership(s1)); //s1 cannot be used after this line
    let s2 = String::from ("Hello");
    println!("Length: {}", borrow_value(s2)); //only a reference is passed
    println!("String: {}", s2); //s2 still can be used
}
```

Code Snippet 4: References in Rust

```
fn main {
    let a = Box::new(5); //5 value will be stored on the heap
    println!("{}", *a);
} //5 and its associated resources will be cleared automatically
```

Code Snippet 5: Box pointer in Rust

```
int main(){
    std::shared_ptr<int> u;
    int *p = new int(5);
    u=std::shared_ptr<int>(p);
    std::shared_ptr<int> u2(u); // copy constructor invoked, increases the reference count
}
```

Code Snippet 6: Shared pointer in C++

B Testing framework and GitHub Link

The GitHub Repo Link of this project can be found at: https://github.com/grakshith/crust/

We wrote a small script benchmark.py[2], which uses config.json[3] to identify which executables to run and for how many iterations. Once the benchmark is run, all the results are stored in a file called `result.json`, which contains the mean execution time, median of those times and their standard deviation times for each executable.

The benchmark creates a `subprocess` for each executable and uses `os.wait4` to wait for the child to finish and get the corresponding stats. The time used for measurement is the sum of `rusage.ru_utime` and `rusage.ru_stime`, which is unaffected by thread sleep times and measures the actual CPU Time taken by the executable. The graphs used in this report compare the mean execution times of the executables.

All the benchmarks used in this report were run on a machine with Intel Xeon E-2136 CPU and 16 GB of main memory.

References

[1] *arc_thread.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/arc_thread.rs.

[2] *benchmark.py*. URL: https://github.com/grakshith/crust/blob/master/src/benchmark.py.

[3] *config.json*. URL: https://github.com/grakshith/crust/blob/master/src/config.json.

[4] *copy_vec.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/copy_vec.rs.

[5] *dyn_dispatch.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/dyn_dispatch.cc.

[6] *dyn_dispatch.s*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/asm/dyn_dispatch.s.

[7] *dynamic.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/dynamic.rs.

[8] *dynamic.s*. URL: https://github.com/grakshith/crust/blob/master/src/rs/asm/dynamic.s.

[9] *for_loop_unrolling.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/for_loop_unrolling.rs.

[10] *for_loop_unrolling.s*. URL: https://github.com/grakshith/crust/blob/master/src/rs/asm/for_loop_unrolling.s.

[11] *generic.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/asm/generic.s.

[12] *generic.s*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/generic.rs.

[13] *generics.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/generics.cc.

[14] *generics.s*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/asm/generics.s.

[15] *linked_list.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/linked_list.rs.

[16] *linked_list_unsafe.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/linked_list_unsafe.rs.

[17] *list.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/list.cc.

[18] *list_unique.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/list_unique.cc.

[19] *loop.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/loop.cc.

[20] *loop.objdump*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/asm/loop.objdump.

[21] *move_vec.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/move_vec.rs.

[22] *mpsc.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/mpsc.cc.

[23] *mpsc.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/mpsc.rs.

[24] *rc_mem_leak.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/rc_mem_leak.rs.

[25] *Rust MPSC channel*. URL: https://doc.rust-lang.org/std/sync/mpsc/fn.channel.html.

[26] *Rust website quote*. URL: https://www.rust-lang.org.

[27] *shared_ptr_mem_leak.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/shared_ptr_mem_leak.cc.

[28] *shared_ptr_thread.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/shared_ptr_thread.cc.

[29] *Stack Overflow 2020 Developer Survey*. URL: https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted.

[30] *vector_copy.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/vector_copy.cc.

[31] *vector_move.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/vector_move.cc.

[32] *weak_list.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/weak_list.cc.

[33] *weak_list.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/weak_list.rs.

[34] *zero_cost_iterators.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/zero_cost_iterators.cc.

[35] *zero_cost_iterators.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/zero_cost_iterators.rs.

[36] *zero_cost_loop.cc*. URL: https://github.com/grakshith/crust/blob/master/src/cxx/zero_cost_loop.cc.

[37] *zero_cost_loop.rs*. URL: https://github.com/grakshith/crust/blob/master/src/rs/src/zero_cost_loop.rs.