Rust-Style Borrow Checking in C

Introduction

Rust is a relatively young programming language that has gained significant popularity in recent years. Arguably its most compelling feature is its borrow checker, which promises memory safety at compile time. The appeal is obvious when compared to the two most common alternatives. The first is manual memory management with no safety guarantees, the system used in C and C++. The second is automatic memory management through garbage collection, the system used in Python and Java. While the first sacrifices safety for runtime efficiency, the second sacrifices runtime efficiency for safety. The Rust borrow checker proposes a third option: strict compile-time checks that ensure memory safety without loss of runtime efficiency.

The borrow checker is divided into three main topics: ownership, borrowing, and lifetimes. Ownership rules ensure every value in the program has exactly one owner variable, and the value is freed when its owner goes out of scope [1]. Borrowing rules govern references, which are pointers to values that do not take ownership of the original value. At any time, there can be any number of immutable references to a value, or exactly one mutable reference to that value [2]. Lifetime rules ensure that references do not outlive the values they point to [3].

By enforcing these rules, Rust eliminates many common memory errors. One such error is use-after-free (and by extension double-free), which is prevented due to the guarantee that there is no way to access a value once its owner frees it by going out of scope [4]. Another class of prevented errors is data races. Since the existence of a mutable reference prohibits the existence of any other references, it is not possible for unsynchronized threads to cause a data race. However, the borrow checker cannot prevent race conditions in general [5]. Finally, the borrow checker prevents many, but not all, memory leaks. Owned values on the stack and the heap are automatically freed when their owner goes out of scope, but some language features such as Reference Counting types enable the user to create blocks of memory that are never freed [6].

The goal of this project is to assess what can be gained by applying these rules to C programs. By implementing a limited borrow checking program and testing it on Linux kernel functions, I show that some aspects of Rust's borrow checker can be easily adapted to C programs. This limited implementation is the basis for my later conclusions on the difficulty and feasibility of borrow checking in C.

Implementation

For the remainder of this report, I will refer to my limited borrow checker as CBorrow, the source code for which can be found at [7].

CBorrow is a Rust program that takes as input a path to a C source file and outputs error messages identifying where the input file breaks the borrow checking rules. My goal was to recreate the output of the Rust borrow checker as closely as possible. To accomplish this, I guided the development process with over twenty toy programs, each representing a situation the Rust compiler responds to in a unique way. After identifying a Rust program that showed new behavior, I transliterated each one into C and modified CBorrow to match the Rust compiler's output. The full list of input files is included in the *inputs/development* directory of the linked repository, and each file includes comments describing the desired error message (or lack thereof).

Parsing

The first step CBorrow takes is parsing the given C source file into an abstract syntax tree (AST). This functionality is handled by a third-party Rust library for parsing C files, called Lang-C [8, 9]. The AST traversal uses the visitor design pattern, where CBorrow can define functions that overwrite the standard traversal to include additional functionality. The written documentation for Lang-C is sparse, but [8] includes a list of every visit function and node type used in the AST. Another feature of the library is printing the entire AST, which was immeasurably helpful during the development process. One such printed AST is included in *ast.txt*. I have encountered two difficulties with this parser. The first is that every included library in the C input file will be stepped-into and read as if it were a part of the input file itself. This is not a bug in the parser, but it did greatly increase the difficulty of testing CBorrow on Linux kernel files. The second is that it is unable to parse some C files, notably stdlib.h.

Design

The smallest data types in CBorrow are defined in *variable.rs*. The most important is *VarType*, which stores whether a variable in the C program is an owner, a copyable value, a mutable reference, or a constant reference. Owners store whether they currently have ownership, and references store a set of variables they might be pointing to. Also included is *Id*, which stores a variable's name and scope for differentiating shadowed variable names. Finally, the *Variable* struct stores a variable's *Id* and *VarType*, along with a set of mutable references that might be pointing to it, and a set of immutable references that might be pointing to it. It may seem odd to store mutable references as a set when the borrow checker asserts that only one exists at a time. In short, a set is needed to model all possible reference states arising after conditional control flow. For more information, see the scopes portion of the rules section.

The AST-independent logic is written in *borrow_checker.rs*, specifically as methods of the *BorrowChecker* struct. The most important field in that struct, *scopes*, stores the entire state of the analysis in a vector. Each element of the vector represents one scope level in the input C file, with the first being the global scope and the last being the current scope. Each scope is stored as a map from variable name strings to corresponding *Variable* instances. During the AST traversal, entering a new scope pushes it to the back of the vector, and leaving a scope pops it from the back of the vector. The next two important fields are *structs* and *functions*, which store the *VarType*s of every struct field and function parameter declared in the input C file. These types are needed to correctly handle struct field uses and function calls. For example, a function call with argument *&x* invalidates all of variable *x*'s references if it takes a mutable reference, but only *x*'s mutable references if it takes an immutable reference. All the other *BorrowChecker* fields are either for the user to control CBorrow's output, or convenience variables used to track values across different nodes in the AST. The most complicated group of these handles compiling struct member names into a single string, since each piece of the name is a separate node in the AST.

An example function in borrow_checker.rs is as follows:

```
408     // Adds the variable's name to the proper scope mapping.
409     pub fn declare_variable(
410         &mut self,
411         declarator: &Declarator,
412         specifiers: &Vec<Node<DeclarationSpecifier>>,
413         function_parameter: bool,
414     ) {
415         let DeclaratorKind::Identifier(identifier: &Node<Identifier>) = &declarator.kind.node else {
416             return;
417         };
418 
419         let name: String = identifier.node.name.clone();
420         let var_type: VarType = self.get_var_type(declarator, specifiers);
421         let scope: usize = self.scopes.len() - 1;
422         self.scopes.last_mut().unwrap().insert(
423             k: name.clone(),
424             v: Variable::new(name.clone(), scope, var_type.clone()),
425         );
426         self.declare_unknown_global(&name, var_type, function_parameter);
427     }
```

This function is called when a variable declaration is visited in the AST. It determines the variable's name and *VarType* from its declarator and declaration specifiers, which are provided by the AST. It then creates a new *Variable* instance for this variable and adds it to the current scope. The final line calls another function that checks if the new variable is a reference that requires a dummy variable to be created as a stand-in for some unknown value. For example, if this variable is a parameter to the function the borrow-checker is running on and is a reference pointing to some value beyond the scope of the analysis, a dummy global variable is used in its place.

The functions that overwrite AST visit behavior are written in *ast_traversal.rs*. There are thirteen in total, with each one handling a different AST node and thus a different element of the C syntax. Most of the complicated logic has been abstracted to *borrow_checker.rs*, leaving only the logic that interacts heavily with the AST. An example function is given below.

```rust
    // Variable declarations.
    fn visit_declaration(&mut self, declaration: &'ast Declaration, _: &'ast span::Span) {
        for declarator: &Node<InitDeclarator> in &declaration.declarators {
            self.declare_variable(
                declarator: &declarator.node.declarator.node,
                &declaration.specifiers,
                function_parameter: false,
            );
            self.visit_init_declarator(init_declarator: &declarator.node, &declarator.span);
        }
    }
```

This function is run any time the AST traversal encounters a variable declaration. It first calls the *declare_variables()* function shown on the previous page, then calls another function to visit the next portion of the declaration, which might include an initialization for the variable.

The final file of the Rust source code is *main.rs*, which simply creates an instance of the *BorrowChecker* struct and calls the proper methods to begin the AST traversal. There are a few arguments the user can adjust in this file, the most important being the path to the input C file. Additionally, the user can specify which functions of the input file they want to check and what kinds of extra output they want the checker to provide.

```
25:     [{p1, p2}]
25:     [{p1, random, p2}]
26:     [{p1, random, p2}]
26:     [{p1, test:1, random, p2}]
27:     [{p1, test:1, random, p2}]
27:     [{p1, test:1, random, z, p2}]
28:     [{p1, test:1, random, z, p2}]
28:     [{p1, test:1, test.testOwner.mutRef, random, z, p2, test.testOwner:1}]
29:     [{p1, test:1, test.testOwner.mutRef, random, z, p2, test.testOwner:1}]
29:     [{random, z, p1, test:1, test.testOwner.mutRef, test.testOwner:0, testKill:1, p2}]
30:     [{random, z, p1, test:1, test.testOwner.mutRef, test.testOwner:0, testKill:1, p2}]
ERROR: Use of moved value 'test.testOwner' used on line 30.
30:     [{random, z, testError:1, p1, test:1, test.testOwner.mutRef, test.testOwner:0, testKill:1, p2}]
```

This is the partial output of checking *ownership1.c* with ownership set prints enabled. Every variable identifier used in the program is recorded in one of the maps in *scopes*, and the keys of each map are printed here. Owner types have a 1 or 0 signifying current ownership. The numbers on the left represent line numbers in the program, with the first print being the state before that line and the second being the state after that line.

```
7:      [{{},{}'->x}]
7:      [{{c1},{}'->x; c1->{x}}]
8:      [{{c1},{}'->x; c1->{x}}]
8:      [{{c1},{}'->x; c2->{}; c1->{x}}]
9:      [{{c1},{}'->x; c2->{}; c1->{x}}]
9:      [{{c2, c1},{}'->x; c2->{x}; c1->{x}}]
10:     [{{c2, c1},{}'->x; c2->{x}; c1->{x}}]
10:     [{m'->{x}; {},{m}'->x; c2->{x}; c1->{x}}]
11:     [{m'->{x}; {},{m}'->x; c2->{x}; c1->{x}}]
ERROR: using 'c1', an invalid constant reference to 'x', on line 11
11:     [{m'->{x}; {},{m}'->x; c2->{x}; c1->{x}}]
```

This is the partial output of *borrow0.c* with reference prints enabled. Reference variables are printed with the form *ref->{variables}*, while other types are printed with the form *{c},{m}'->variable*. In either case, an apostrophe signifies a mutable reference.

Rules

I will now give an overview of the rules CBorrow follows during analysis.

Global Declarations

Four possible global declarations are handled: structs, function prototypes, global variables, and function definitions. Struct declarations have their field types analyzed and converted into *VarType*s for later lookup when an instance uses one of its fields for the first time. Function declarations similarly have their parameter types analyzed and converted into *VarTypes* for lookup when the function is called. Global variables are treated the same as local variables but are placed in the global scope map. Finally, function definitions are analyzed only if the function name is included in the user-defined list of functions to analyze. If that is the case, the function's scope is added to the *scopes* vector, and its parameters are added to the scope. Because this is an intraprocedural analysis, the details of the arguments that will be passed to fill these parameters are unknown. In their place, when needed, CBorrow creates some unknown global variable. This is necessary when parameters are pointer types. Instead of pointing to nothing, they point to a dummy global variable whose name matches the pointer but with '?' affixed to the front.

Scopes

A change in scope is identified by an opening or closing curly bracket. Upon entering a new scope, a new map is added to the *scopes* vector. The state of the *scopes* vector is then saved in a backup and the code inside the new scope is analyzed. After the analysis, the resulting *scopes* vector is merged with the backup. During the merge, conflicts of ownership are resolved by assuming the variable has lost ownership, and differences in sets of references are resolved by taking the union of the two possibilities. In other words, the control flow is that of a strict "may" analysis, where losing ownership

or creating references in any scope results in as many error messages as possible. This merging process works well for loops and if-statements, whose bodies may not run, but it does not support simple scopes made of just curly brackets with no conditional control flow. After the merge, the now-completed scope is popped off the *scopes* vector. Any references to variables in the removed scope will print an error if they are used.

Conditionals

The above scope mechanism works as intended for loop bodies, but for if-else blocks, one additional step is required. The *else* portion must be analyzed as if the *if* block was not run, and vice versa. Because this is a forward analysis, the *if* block behaves this way by default. For the *else* block, one additional backup of the initial *scopes* vector is stored before analyzing the *if* block and re-loaded before analyzing the *else* block.

Variable Declarations

The declared variable is added to the current scope. Its *VarType* is determined by analyzing its type specifiers. A dummy variable may also be created if needed, as described in *Global Declarations*. All structs are assumed to be owner types, pointers marked with *const* before the type are immutable, all other pointers are mutable, and any type that does not meet those criteria is assumed to be a copy type. The decision to count only *const* pointers as immutable stems from a desire to make CBorrow as much like the Rust compiler as possible. Where Rust uses an explicit mutability flag *mut*, it is reasonable for CBorrow to use C's explicit immutability flag instead of trying to infer pointer types based on use.

Assignments (LHS = RHS)

If LHS is an owner type, its ownership is renewed. If RHS is an owner type, its ownership is revoked. If either LHS or RHS is an owner or a mutable reference, all references to them are invalidated, as moving a borrowed value and then using the reference is illegal. Note that in CBorrow, an invalid reference occurs when a reference stores a variable that it is pointing to, but the variable does not store the reference as one of its pointers. If LHS is a pointer, RHS is added to LHS's set of pointed-to variables, and LHS is added to RHS's set of references. The mutability or immutability of the added reference is decided by LHS's *VarType*, which is always defined before this step. If RHS is a dereferenced reference, CBorrow checks that its referenced variable is either a copy type or an immutable reference, both of which are copiable. If both LHS and RHS are reference types, they must be of the same type. An immutable reference assignment simply adds another immutable reference. A mutable reference assignment adds LHS as a mutable reference and invalidates RHS.

Function Calls

The effects on each argument's state are computed separately. For each argument, if it is an owner type, it loses ownership, and all its references are made invalid. If it is the address of a variable using *&*, the stored signature of the called function is checked to

see whether that argument takes a mutable reference or an immutable reference. If the argument represents the creation of a mutable reference, all references to the variable are invalidated. If it represents the creation of an immutable reference, only the mutable references to the variable are invalidated. The same dereferencing rules as in *Assignments* are applied if the argument is a dereferenced reference. Passing a copy type, mutable reference, or immutable reference variable directly has no effect on the variable's state.

Identifiers

Each identifier visited is used as a chance to check for common error conditions and print error messages. Identifiers representing owner types are checked to see if they have ownership over their value. Identifiers representing reference types are checked to see if they are still considered valid by the variables they may point to. Each struct member identifier seen is also subject to these checks, for example, if the input C file includes *x.y.z* and *x.y* is an owner type without ownership of its value, an error is printed. There are exceptions to the identifier checks, such as the LHS of an assignment, which does not require ownership to be valid, since it is being assigned.

Member Expressions

Handling member expressions is one of the most logically complicated portions of CBorrow due to the way each name of the expression is separated into its own node in the AST. Another difficulty is that fields of a struct are not declared explicitly. For example, if a struct is passed into the analyzed function as a parameter, it might use one of its fields for the first time as the RHS of an assignment. This necessitates CBorrow's ability to create variables outside of declarations, as well as store all struct field types to discover the *VarType* of the new struct member. Another approach would be to declare all the struct's members as soon as the struct itself is declared. I elected to handle it dynamically to reduce the size of the *scopes* vector.

Dereferencing (operator)*

The reference being dereferenced is tested for validity, then stores the first variable in the set of variables it possibly references, for use by the calling function. Taking just the first of the possibly referenced variables is an assumption that could cause CBorrow to deviate from the Rust compiler's output, though this is a product of CBorrow's design and not a limitation of the C syntax.

Evaluation

To evaluate the behavior of CBorrow compared to the Rust borrow checker, I copied functions from the Linux kernel and transliterated them into Rust. I then ran CBorrow on the original C functions and the Rust compiler on the transliterated functions to compare the error messages produced. This process was more labor intensive than anticipated, as the parsing library was incapable of handling all the linked header files necessary to parse the C functions in their original files. To get around this issue, I copied every necessary definition into the testing files, barring some unused struct members, which were omitted to avoid an ever-expanding web of dependencies. Nonetheless, the time required to create and transliterate examples means I have only tested two functions. The first function, nicknamed *kernel0*, has several revisions representing attempts to resolve the identified errors.

Kernel0*:*

This test is run on *perf_event_max_stack_handler()* from the file callchain.c. All kernel function files and the changes made to them can be found in */inputs/kernel0*. Note that the line numbers may differ, and CBorrow may flag a connected issue, such as the use of an invalid reference instead of the line where the reference was invalidated.

The initial function code (after altering some syntax to fit the limitations of CBorrow):

```
59  // function to check, from callchain.c
60  int perf_event_max_stack_handler(struct ctl_table *table, int write, void *buffer, size_t *lenp, loff_t *ppos) {
61      int *value = (*table).data;
62      int new_value = *value, ret;
63      struct ctl_table new_table = *table;    // ERROR: Cannot move non-Copy type '?table' from behind a reference.
64
65      new_table.data = &new_value;
66      ret = proc_dointvec_minmax(&new_table, write, buffer, lenp, ppos);
67      if (ret || !write) {
68          return ret;
69      }
70
71      mutex_lock(&callchain_mutex);
72      if (atomic_read(&nr_callchain_events)) {
73          ret = -EBUSY;
74      }
75      else {
76          *value = new_value;
77      }
78
79      mutex_unlock(&callchain_mutex);
80
81      return ret;
82  }
```

Round 0: Initial code. Rust and CBorrow both give the same error:
Rust:

```
error[E0505]: cannot move out of `*table` because it is borrowed
  --> src\main.rs:73:36
   |
62 |     table: &mut ctl_table,
   |     ----- binding `table` declared here
...
70 |     let value: &mut i32 = (*table).data;
   |                            ------------- borrow of `*table.data` occurs here
...
73 |     let mut new_table: ctl_table = *table;
   |                                    ^^^^^^ move out of `*table` occurs here
...
85 |         *value = new_value;
   |         ------------------ borrow later used here
```

CBorrow:

```
63      struct ctl_table new_table = *table;
ERROR: Cannot move non-Copy type '?table' from behind a reference on line 63.
```

Round 1: After changing the mutable reference *table* to a concrete struct parameter, Rust and CBorrow still give matching errors.
Rust:

```
error[E0505]: cannot move out of `table` because it is borrowed
  --> src\main.rs:73:36
   |
62 |     table: ctl_table,
   |     ----- binding `table` declared here
...
70 |     let value: &mut i32 = table.data;
   |                           ---------- borrow of `*table.data` occurs here
...
73 |     let mut new_table: ctl_table = table;
   |                                    ^^^^^ move out of `table` occurs here
...
85 |         *value = new_value;
   |         ------------------ borrow later used here
```

CBorrow:

```
76          *value = new_value;
ERROR: using 'value', an invalid mutable reference to '?table.data', on line 76
```

Round 2: After moving the declaration of pointer “value” after the declaration of *new_table*. Rust and CBorrow still give matching errors.
Rust:

```
error[E0382]: borrow of moved value: `table`
  --> src\main.rs:80:27
   |
62 |     table: ctl_table,
   |     ----- move occurs because `table` has type `ctl_table<'_>`, which does not implement the `Copy` trait
...
72 |     let mut new_table: ctl_table = table;
   |                                    ----- value moved here
...
80 |     let value: &mut i32 = table.data;
   |                           ^^^^^^^^^^ value borrowed here after move
```

CBorrow:

```
70      int *value = table.data;
ERROR: Use of moved value 'table' on line 70.
```

Round 3: Replacing the *new_table* initialization as follows:

```
64      struct ctl_table new_table;
65      new_table.data = &new_value;
66      new_table.maxlen = (*table).maxlen;
67      new_table.extra1 = (*table).extra1;
68      new_table.extra2 = (*table).extra2;
```

This variation gave no errors in both the Rust borrow checker and CBorrow.
This is because *new_value* has been separated from ‘table’ and the other fields, which are borrowed from *table*, are not used by *table* until after *new_table* is done being used. Note that *new_table* cannot move ownership out of any of the fields it has borrowed from ‘table’ because they are behind mutable references.

Kernel1*:*
For the second test, I copied *badblocks_set* from *badblocks.c*. After collecting all the dependencies, CBorrow reported no errors. After translation, including removing all pointer indexing, the Rust compiler also reported no errors.

While I wish I was able to test on more Linux kernel functions, I firmly believe most differences between CBorrow’s output and the Rust compiler’s output at this stage would be due to easily fixable C syntax oversights, assuming an input within CBorrow’s project scope.

Limitations

CBorrow is by no means a completed borrow checker, as should be clear by the relative simplicity of the rules specified above. That said, a full borrow checker for C programs is and should be beyond the scope of this project. In this section, I list some limitations of CBorrow. This is not a complete list, as even now I am sure there are aspects of the borrow checker and the C syntax that could cause unanticipated results.

Single-Function

CBorrow is a single-function tool. Even analyzing each function of a program sequentially using CBorrow would not give the same results as analyzing an entire program at once with the Rust compiler. The most obvious proof of this fact is CBorrow's inability to handle the values of pointers passed as parameters to functions, instead using dummy global variables. Return values from function calls are also not supported and would require similar approximations in the current design, as the true return value cannot be known.

Lifetimes

Explicit lifetimes are not supported. In Rust, lifetime specifiers have two uses. The first is informing the borrow checker how long a reference returned from a function should be considered valid, relative to the parameters of the function. However, CBorrow does not consider the return values of function calls. If it did, lifetime annotations for function prototypes would become necessary, which would be the first required change to the C syntax I've encountered. However, assuming user-specified function prototype lifetimes, I believe the reference to the return value of the function could simply have its scope adjusted to enforce the specified lifetime requirements. The second use of lifetime specifiers is on struct definitions and their fields, which means the reference type fields must outlive their parent struct. While explicitly specifying the lifetime of each field is not possible without changing the C syntax, CBorrow does implicitly apply some of the same logic. Whenever a struct field is declared, it is assigned the same scope as its parent struct. If a struct field points to some variable with a shorter lifetime and is used after the variable goes out of scope, CBorrow prints an error. So, while CBorrow does not support explicit lifetimes, it does support implicit lifetimes within the checked function, and I believe the current model could be adapted to support explicit lifetime specifiers.

Copy vs Owned

CBorrow assumes that all struct types are owned and all primitive types are copyable. Of course, in Rust, it is possible to implement the Copy trait on a struct, so this assumption is stricter than it should be. Additionally, void pointers pose a challenge, as the type they are referencing is unknown statically. Assuming a void pointer references either a copyable type or an owned type is easy, but differentiating between the two cases is more challenging. CBorrow assumes such referenced variables are copyable and does not attempt to discern any further information.

C Syntax

CBorrow's grasp of C syntax is extremely limited. The only guaranteed supported components of the C syntax are those listed by name in the rules section. Additionally, CBorrow is subject to the following limitations:

- Stacked address operators &&x are not supported, though references to references are in general.
- The arrow operator x->y is not supported, (*x).y must be used instead.
- Array indexing is not supported.
- Called functions must be declared as function prototypes.
- Function prototypes must name all parameters.
- Ellipsis parameters such as in *printf()* simply assume arguments should be treated as mutable references or owner types (assumed strictness).
- Every block must be surrounded by curly brackets, even single-line if-statement and loop bodies.
- The parsing library crashes while parsing some inputs, such as stdlib.h.

Conclusion

Throughout the development of CBorrow, I have become convinced that much of the logic used in the Rust borrow checker can be applied to C programs to improve their memory safety. Though CBorrow is subject to the many limitations listed in the previous section, none of them have given me cause to believe that borrow checking as a whole is infeasible in C. On the contrary, at an intraprocedural level, a relatively simple forward analysis can replicate the common error messages of the Rust borrow checker. This differs greatly from my hypothesis at the start of this project, when I believed extensive changes to the C syntax would be necessary to get any tangible value from a borrow checker.

To achieve memory safety using a hypothetical C borrow checker more advanced than CBorrow, the most important change would be to consider the true arguments and return values of every function in the program. A checker that does not understand the connections between function calls and the data they return cannot guarantee that a program is memory safe. I believe an improved analysis is possible in C by analyzing the context of each function call, specifically the scopes of the arguments used in the call. To fully implement this change, it would be necessary to augment the C grammar with explicit lifetime specifiers. Automatic memory safety, would also require programmatically inserting *free()* statements since a completed borrow checker must enforce that owned values are freed when the owner goes out of scope.

Due to the difficulty I faced in parsing and analyzing the C syntax and the changes necessary to create a complete borrow checker, I believe the best approach for a full-scale implementation would be a new C compiler designed with borrow checking in mind. However, even the small necessary syntax changes would differentiate it as a language separate from C. I doubt such a language would be able to find many users when competing with standard C and Rust.

With the current implementation, the only purpose I can imagine for a tool as incomplete as CBorrow is perhaps in translating C programs to Rust. A translator could decide whether a specific function needs to be marked *unsafe* based on the output of a tool like CBorrow. However, this project has given me no reason to believe a complete, Rust-style borrow checker cannot be implemented for C programs. Assuming a few small changes to the C syntax and paradigm, such as explicit lifetime identifiers and programmatic *free()* statements, all of Rust's memory safety guarantees listed in the introduction could be enforced in a C program.

References

[1] https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
[2] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
[3] https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html
[4] https://insights.sei.cmu.edu/blog/rust-software-security-a-current-state-assessment/
[5] https://doc.rust-lang.org/nomicon/races.html
[6] https://doc.rust-lang.org/book/ch15-06-reference-cycles.html
[7] https://github.com/DSWhiteHat/c_borrow_checker
[8] https://docs.rs/lang-c/latest/lang_c/
[9] https://github.com/vickenty/lang-c