Implementing the Rust Borrow Checker in C++

Fred Cook - 8463955

November 2021

Contents

List of Tables

List of Figures

List of acronyms

BST Binary Search Tree

GC Garbage Collector

1 What is rust and what makes it differernt?

Mozilla - the developers of the Rust language say: "Rust is an open-source systems programming language that focuses on speed, memory safety and parallelism." Rust also shares many other similarities with C and C++, these being that they are compiled languages, offer low level control, and (primarily relating to C++) the extensive use of "Zero cost abstractions" (Mozilla, n.d.).

Rust however aims to go further than it's "cousin" languages. The main thing that rust does to make it "better" than languages like C and C++ and set itself apart relates to memory safety. In an interview Graydon Hoare, the man who started developing the Rust language says "Primarily, it's just much safer, less likely to crash." when asked "What makes it (Rust) better than C?" (Avram, 2012).

Rust offers features not in other languages which allow it to achieve memory safety through guarantees, one of the main methods Rust uses to achieve this is the "Borrow Checker" (See section 3).

1.1 What is a "Zero Cost Abstraction?"

Bjarne Stroustrup (The creator of the C++ language) defines a set of ideals that make C++ what it is, these are: "A simple and direct mapping to hardware", and "Zero-overhead abstraction mechanisms" making it central to the core philosophy of C++ (Stroustrup, 2012). Stroustrup goes on to say "By "light-weight abstraction," I mean abstractions that do not impose space or time overheads in excess of what would be imposed by careful hand coding of a particular example of the abstraction." (Stroustrup, 2012). An example of a zero cost abstraction is as follows:

```
1 #include <array>
2 int main() {
3     std::array<int, 30> myArray;
4     myArray.at(25) = 50;
5     return myArray.at(25);
6 }
7
```

Figure 1: "Modern" C++ std::array zero-cost abstraction

This code is trivially simple in that all it does is create a std::array object which is a collection of type T, T in this case being "int", the container containing 30 elements. the 26th element is then assigned the value of 50, the program returns the value in the 25th index and terminates.

Similar behaviour can be achieved with traditional, "hand written" code as follows:

```
1  int main() {
2      int myArray[30];
3      myArray[25] = 50;
4      return myArray[25];
5  }
6
```

Figure 2: Traditional C / C++ array

The behaviour of these two programs is exactly the same and infact produce the exact same executable, we can see that the assembly output for both using x86-64 gcc version 11.2 is as follows

```
1  main:
2          mov     eax, 50
3          ret
4
```

Figure 3: Assembly output for figures 1 and 2

From this we can see that because these two implementations provide the same assembly output, and therefore same runtime performance as one another that the std::array abstraction has "zero-overhead", despite providing the same functionality and in-fact it provides more functionality than the traditional C-like implementation, one of these being bounds checking.

For example if we were to try the following and accessing an element out of the range of the container (undefined-behaviour) which renders the invalidates the program, although it may still run and may produce correct results Zubkov, 2015:

```
1  #include <array>
2  int main() {
3      std::array<int, 30> myArray;
4      myArray.at(25) = 50;
5      return myArray.at(100);
6  }
7
```

Figure 4: Demonstration of bounds checking with std::array

we get a compilation error, this being "terminate called after throwing an instance of "std::out_of_range"". This therefore demonstrates that the std::array object provides additional functionality to us, such as the "std::array<T,N>::at(size_type

pos)" method) which could be hand written, but has little or no advantages in terms of runtime performance to be made. The argument could be made that the bounds-checking operation could slow down the program, having to check if the index was within the array every time the at method is used, however, because these checks are made at compile time, there is 0 runtime performance difference again, demonstrated through the assembly output of the different examples being the same, making the argument moot

2 Rust and it's relation to C++

Rust is a relatively new language that aims to remove many of the barriers to entry in the "systems-level" programming space Mozilla (2018). Currently the systems-level market is primarily dominated by projects written in C (and by extension C++ to some degree), so much so that in his article "After All These Years, the World is Still Powered by C Programming" Munoz gives a scenario of waking up on an average day, using various appliances in the kitchen, watching the television etc, all of these systems are, according to the author most likely programmed in C Munoz, n.d.

Graydon Hoare also says that Rust's target audience is "frustrated C++ developers" Avram, 2012 and for this reason it's easy to see many of the similarities between the two languages and what Rust tries to do different to set it apart from one of the oldest languages still widely in use today, with C++ being the 4th most widely used language in the world at the time of writing according to the TIOBE index, which aims to track the popularity of different programming languages based on metrics such as number of search engine searches in a given time frame and proportion of lines of code written TIOBE, n.d.

3 The Rust Borrow Checker

The Rust Borrow Checker is a feature in rust that aims to prevent a series of memory safety mistakes such as "use after free" and "dangling pointers" without the need for a Garbage Collector (GC) (Stanford University, 2018). As this is done at compile time it has no runtime overhead.

The rules set out by the borrow checker are:

- Any borrow must last for a scope no greater than that of the owner
- *You may have either but not both of the following:*
  - *One or more references (& T) to a resource.* (a T is a type, such as int, character, vector, etc).
  - *Exactly one mutable reference (& mut T)*

  (Klabnik & Nichols, 2018)

An example of the borrow checker is as follows:

```
1  fn main() {
2      let mut x = 5;
3      let y = &mut x;
4
5      *y += 1;
6
7      println!("{}", x);
8  }
9
```

Figure 5: Demonstration of Rust's borrow checker

This program results in an error:

```
error: cannot borrow ‘x‘ as immutable because it is also borrowed as mutable
    println!("{}", x);
```

This happens because we have a "mutable reference" to "x" on line 3, meaning x and y both point to the same place in memory and can both modify the value there. When we then try to pass a reference to x to the "println!" function meaning we have both, a mutable reference, and a non mutable reference in scope at the same time, violating the rules mtnioned in section 3. In this case, the borrow checker prevents a possible race condition in the case that the modifications to y take place in the form of an asynchronous function, such as:

```
1          use futures::executor::block_on;
2  async fn increment(x:&mut u32) -> () {
3          *num = *num + 1;
4  }
5
6  fn main() {
7      let mut x = 5;
8      let y = &mut x;
9
10     let future = increment(y);
11
12     println!("{}", x);
13     block_on(future);
14 }
15
```

Figure 6: Demonstration of Rust's borrow checker with asynchronous function

Because of the nature of static analysis being conservative, 3 would have no problem executing normally and no race condition would occur, but Rust wants to ensure that if you wanted to refactor and make changes later that would have a race condition minimal structural changes would occur.

This is a common theme in Rust, it forces your code to be correct in all instances, so that even if there would be no issue in any possible similar situation.

4 The need for safety in programming

The need for safety in programming has been ignored for a long time. This had lead to a large number of projects that have become part of huge systems we rely on, such as the NHS infrastructure, being bug prone (Cummings, n.d.). This means that there's economic incentives, aswell as safety reasons to create safe, reliable software.

One example of error prone software causing huge economic damage was *"the CEO of Provident Financial announced a software glitch that led to the company collecting only a little more than half of loan debts on time, the stock prices tumbled 74% in a single day. The share price reduced from £17.42 to just £4.50. He resigned soon after."* (Cummings, n.d.).

5 Programming language design and implementation

5.1 The programming language typing spectrum

Programming languages have three criteria (among others) which strongly define it's behaviour. These three features can be plotted on a three-dimensional graph such as in figure the "Lambda Cube" (see figure **??**).

5.1.1 Where are C++ and Rust on this spectrum?

6 Translators, compilers and compilation

Translators are programs which convert source code (plaintext) into executable code. There are 3 types of translators:

- Compilers
- Interpreters
- Assemblers

Each of the three have their advantages and disadvantages, but the main two are compilers and interpreters. Compilers generally create faster executing code due to the fact that all the translation is done ahead of time, however,

this does it can take a long time to start running your code, interpreters remove this.

The process of compilation is as follows:

6.1 Lexical analysis

Lexical analysis is the first stage of compilation and involves taking a stream of input, such as a source code file, and remove all unused pieces of text, such as white space, comments, etc, and output a series of "Lexemes", lexemes essentially representing the smallest pieces of text which represent something. For example, the code:

```
1 // beginning of source code
2 var my_number = 42;
3
```

Figure 7: Input source code for an example Lexer

Would become a series of lexemes, of "var", "my_number", "=", "42", and ";" (Nystrom, 2021).

Note that the commend on line 1 is completely removed.

6.2 Symbol table construction

The next stage in the process involves taking the output of the Lexer, and involves constructing a structure (often a Binary Search Tree (BST)) and creates a hierarchical structure which contains information about an indentifier (name) such as it's type, it's name, it's scope etc.

6.3 Syntax analysis

Syntax analysis (also known as parsing) we interpreter what the tokens (from the Lexer) mean and checking the validity of their structure. For example, in figure 7, if the semi-colon was removed, the lexer would be okay and continue as normal, but the syntax analyser would determine that the program is malformed, as in this language, a semi colon is needed at the end of every line and it would halt the compilation process.

6.4 Semantic analysis

Semantic analysis takes a valid series of lexemes from the syntax analyser and checks that they follow the rules of the language. This includes things such as no break statements outside of a loop. The semantic analyser performs other functions as well, such as type checking.

6.5 Code generation

Code generation could be considered as the final stage of the compilation process (optimisation as described in 6.6 is optional). It involves generating the executable machine code. Code generators do not produce an executable for interpreted languages such as python, but do for fully compiled languages such as C.

6.6 Optimisation

Optimisation is an optional stage in the compilation process, and involves the code optimiser indentifying parts of the program which could be replaced with other, optimised code. An example of this can be seen in figures 1, 2, and 3 in which the compiler notices that just about all of the code is actually unused and doesn't produce a side effect (something like a write to a file, or anything with persistence) and so it removes it.

7 The C++ Ecosystem

7.1 Compilers

7.2 Tooling

7.3 Language Development

Avram, A. (2012). Interview on rust, a systems programming language developed by mozilla. https://www.infoq.com/news/2012/08/Interview-Rust/

Cummings, R. (n.d.). The real cost of software bugs. https://www.softwaretestingnews.co.uk/the-real-cost-of-software-bugs/

Klabnik, S., & Nichols, C. (2018). *The rust programming language*. No Starch Press. https://doc.rust-lang.org/1.8.0/book/README.html

Mozilla. (n.d.). Rust language. https://research.mozilla.org/rust/

Mozilla. (2018). *The rust programming language*. https://doc.rust-lang.org/stable/book/title-page.html

Munoz, D. (n.d.). After all these years, the world is still powered by c programming. https://doc.rust-lang.org/stable/book/title-page.html

Nystrom, R. (2021). *Crafting interpreters*. Genever Benning.

Stanford University. (2018). Memory safety in rust. https://stanford-cs242.github.io/f18/lectures/05-1-rust-memory-safety.html

Stroustrup, B. (2012). Foundations of c++, 1–2. https://www.stroustrup.com/ETAPS-corrected-draft.pdf

TIOBE. (n.d.). "tiobe index for november 2021". *TIOBE index*. https://www.tiobe.com/tiobe-index/

Zubkov, S. V. (2015). *Undefined behavior*. https://en.cppreference.com/w/cpp/language/ub

Glossary

assembler Translates assembly code into machine code.

Binary Search Tree A Binary Search Tree (BST) is a sorted, recursive data structure which has a root node, and separates all values less than the root node into one half, and all values greater than the root node into the other half.

compiler A program which translates code (plain text) into assembly code. Examples include GCC and Clang.

Garbage Collector A Garbage Collector (GC) is a program or part of a program which automatically frees unused (garbage) memory during runtime - increases overhead for the program.

interpreter Translates the code to line by line, during execution (such as with Python and Javascript)

Word count: 1595