Exploring language-based security with Rust

Luigi Coniglio - s1987046

December 19, 2017

The table below shows the versions of the tools that have been used for this assignment:

Table 1: Tools version

Program	**Version**
cargo	0.23.0 (61fa02415 2017-11-22)
rustc	1.22.1 (05e2e1c41 2017-11-22)

B: Implement a text-file analyzer

1. Introduction

All the files related to this part of the assignment can be found in the folder *textstat*:

```
textstat
├── textstat_bytes
│   ├── Cargo.lock
│   ├── Cargo.toml
│   └── src
│       └── main.rs
├── textstat_utf8
│   ├── Cargo.lock
│   ├── Cargo.toml
│   └── src
│       └── main.rs
├── test_ascii.txt
└── test_utf8.txt
```

Two versions of the assignment have been realized: one iterates over the bytes of the input file, therefore reading ASCII characters, and a second version considering utf-8 codepoints. They can be found respectively in the *textstat_bytes* and *textstat_utf8* folders. Further in this report I will compare this two solutions evaluating their performances.
The files *test_utf8.txt* and *test_ascii.txt* are the input files that have been used for the tests. The first one consist of three consecutive repetitions of two (very different) great literary works: *"A Christmas Carol"* by Charles Dickens and *"La*

Divina Commedia" by Dante Alighieri. Both books have been taken from the Gutemberg Project[1] and make use of utf-8 encoding. The second file contains random sequences of ascii characters. Those two files have similar sizes.

2. Approach

Both parts of the assignment (part B and part C) have been realized using only Rust's standard library (i.e. without using external crates) and trying to comply as much as possible with the idiomatic construct of the language. Furthermore *unsafe* blocks have not been used.

Rust code often tends to eschew readability [2, 3] therefore, also for this reason, every action in the source code is documented.

For both parts B and C, I tried to preserve the reliability of the program taking the due precautions when dealing with edge cases. In some occasion I decided to opt for reasonable compromises: for instance, in the context of *textstat*, a very very large number of words could led to an overflow of the counter(s), which would cause the program to panic (in debug mode) or not (in release mode)[1] but, while a check for overflows could be done, in my opinion the problem is mitigated well enough just by using the type *usize* for counters (thus eliminating the performance drawback) which, on modern computers, would require an unrealistic number of words in order to overflow.

By default the statistics showed include a sorted list of the number of words per word size up to 10 characters and a sorted list of the 10 most used words. The bounds for these two lists can be easily modified changing the global constants *MAX_WLENGTH* and *MAX_TOPUSAGE_LIST* (which are both set to 10 by default).

3. Results

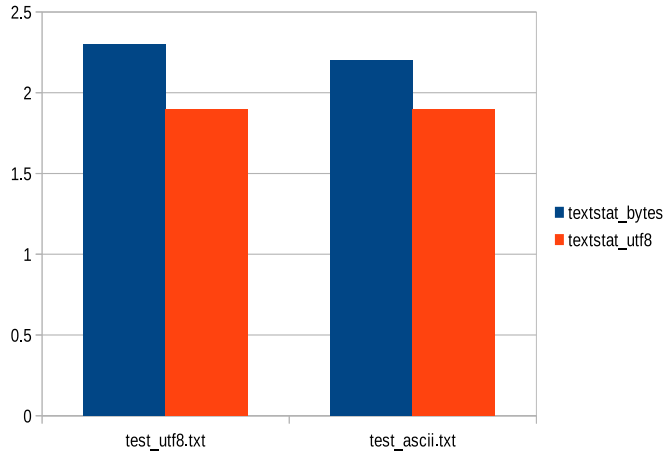
As indicated in the introduction I realized two version for the program *textstat*: one iterating over the bytes of the input file and another one taking advantage of Rust's modern approach to strings in order to iterate over utf-8 codepoints. While in Rust, the trait *Read* offers an handy iterator over the bytes of the input [4], the API offering an iterator over the *chars* is currently unstable [5, 6]. On the other hand, Rust offers a way to iterate over the characters of a *String* but, since for big files it would be unpractical to load all the file in memory as a *String*, *textstat_utf8* reads the file one line at a time.

Both approach have pros and cons: for instance *textstat_utf8* would load all the file in memory if this last doesn't contain any newline character.
I compared the performances of those two implementation using both an utf-8 encoded input and an input containing only ascii characters. The test have been done using the tool *time* and only considering user-mode time (thus partially eliminating the imperfection caused by scheduling etc.)

[1] http://www.gutenberg.net

Figure 1 shows the result of the tests: iterating over Rust *chars* seems to be slightly more performant than iterating over the bytes. The fact that a similar result is obtained using an ascii-only input file shows that most of the difference is not caused by *textstat_utf8* occasionally handling more than one byte at each iteration.

Figure 1: Average execution time using *test_ascii.txt* and *test_utf8.txt* as input



This result was unexpected to me, but probably it is related to Rust optimizing iterations over a *String*.

C: Reimplement the binary search tree in Rust

1. Introduction

All the files related to this part of the assignment can be found in the folder *rustsint*:

```
rustsint
├── Cargo.lock
├── Cargo.toml
└── src
    ├── main.rs
    └── sortedcontainer.rs
```

The binary search tree has been realized as a separate module and can be found in the file *sortedcontainer.rs* together with a module dedicated to the tests.

2. Approach

All the statement regarding part B are valid for this part as well. For instance no external crates have been used nor unsafe blocks.
I decided to better organize my code implementing SortedContainer in a separate module. SortedContainer make use of generic data types thus can be reused

with any kind of data. Since simpler data types can be used, this approach was very handy while implementing the tests.

The links between each node are defined using Rust's Options [2].

All the operations on the tree are implemented recursively. In order to avoid redundancy, the common operation of "finding the position of a node with a given value" is performed by an helper private method *find_pos* which is in turn used by the methods *insert*, *contains* and *erase*.
Recursive methods in Rust pose few problems. For example, since each method takes as first argument a reference to *self*, passing another attribute of the structure would cause this last to be borrowed twice. In my case I solved this issue by the mean of helper functions that take only the values from *self* that they needs (another solution could have been using interior mutability, e.g. *Cell* or *RefCell*, to safely mutate fields of *self* while they are shared or, to temporarily move the attribute in object out of *self* replacing it with a temporary value, passing it to the method and then moving it back).

Bonus: I implemented the method *erase*. My implementation of this method uses the right in-order successor when eliminating a node with two children.

Fixes: I did some minor modifications to the original code. I introduced a check for empty inputs (which previously caused the program to panic) and changed the type of the age to *u32* in order to prevent negative ages.

3. Result and tests

The program can be tested providing manual input or by running the automated tests: *cargo test*.
The automated tests cover most of the edge cases (targeting the method *erase* in particular). The *test* module includes the following seven methods:

- *test_base_api*: regression tests targeting mostly a normal usage of the public API
- *test_double_erase*: tests a bad usage of the public API performing a double removal
- *test_erase_no_children*: tests the deletion of a node with no children
- *test_erase_with_one_child*: tests the deletion of a node with only one child
- *test_erase_with_two_children_1*: tests the deletion of a node with two children twice (if node is root, or not)
- *test_erase_with_two_children_2*: tests the deletion of a node with two children who each have two children as well
- *test_erase_with_two_children_3*: tests the deletion of a non-root node with two children who each have two children as well

[2] Since Options are a binary enums Rust optimizes them eliminates the space needed for the tag [8].

For ease of reading, for the most complex tests the configuration of the tree at each step is represented graphically in the comments.
The test module can be found at the end of *sortedcontainer.rs* (so that the module test can access the private fields of the *SortedContainer*).

My takeaway

Learning Rust was from long time ago in my todo-list. This assignment was a nice introduction to Rust and his principles and mechanism. Despite the challenging constraints imposed by Rust, the very good documentation made the language easy to approach.

Following there are few "hints" that I learnt to keep in mind while programming in Rust (mostly as a memo to myself):

- moving data temporarily can solve a lot of problems
- it is often useful to play with scopes
- don't forget to dereference boxed structures (so that the borrow checker can differentiate the fields)
- helper functions can be a solution

References

[1] H. Wilson , `http://huonw.github.io/blog/2016/04/myths-and-legends-about-integer-overflow-in-rust/`, c2016, last access: 14 December 2017.

[2] My experience rewriting Enjarify in Rust , `https://news.ycombinator.com/item?id=12532798`, last access: 14 December 2017

[3] C++ design goals in the context of Rust (2010) , `https://news.ycombinator.com/item?id=7924856`, last access: 14 December 2017

[4] Trait std::io::Read, method *bytes* , `https://doc.rust-lang.org/std/io/trait.Read.html#method.bytes`, last access: 14 December 2017

[5] Trait std::io::Read, method *chars* , `https://doc.rust-lang.org/std/io/trait.Read.html#method.chars`, last access: 14 December 2017

[6] Open issue for *Read::chars* , `https://github.com/rust-lang/rust/issues/27802`, last access: 14 December 2017

[7] *time* command , `http://man7.org/linux/man-pages/man1/time.1.html`, last access: 14 December 2017

[8] Learning Rust With Entirely Too Many Linked Lists ,`http://cglab.ca/~abeinges/blah/too-many-lists/book/first-layout.html`, last access: 14 December 2017