Returning a reference to a to-be-populated vector

help

lut_99 August 5, 2025, 1:10pm 1

Hello all,

I'm currently working on a parser that will attempt build some AST that links to the original source text. I.e., the toplevel function looks a bit like:

```
fn parse<'s>(source: &'s [u8]) -> Result<Ast<'s>, Error<'s>>;
```

The `Error` also returns a reference because it, too, refers to a piece of the source text where an error occurred, so it can be rendered to the user á la Rust style.

Now, everything's well and good - until I start thinking about dependencies. The language I'm working on needs to support C-style imports; i.e., parse until you find an `#include`, replace that statement with the contents of the included file, and then continue with the rest of the file.

Nice idea in theory - but I can't get it to work lifetime-wise! I understand that I can't write to the buffer of input source text *while* I'm keeping AST's with a reference to it around, those would be invalidated when the buffer inevitably re-allocates. So, instead, I wanted to split it in two phases: first, parse a file, do a parsing pass to find includes, keep only those (discard the rest of the AST), recurse into the next file, append it to the first one, etc. I.e.: build a giant file first, then parse that as you normally would.

But, now my `Errors` start to become annoying. I've got something like the following to build the giant file:

```
fn build_file<'s>(path: PathBuf, source: &'s mut Vec<u8>) -> Result<(), Error<'s>> {
   let mut source_len: usize = source.len();
   let mut todo: Vec<PathBuf> = vec![path];
   while let Some(path) = todo.pop() {
       // Attempt to read the file
       match File::open(&path) {
           Ok(mut handle) => {
               if let Err(source) = handle.read_to_end(source) {
                   return Err(Error::FileRead { path: path.into(), source });
               }
           },
           Err(source) => return Err(Error::FileOpen { path: path.into(), source }),
       }

       // Attempt to find dependencies from the nested file
       let deps: Vec<PathBuf> = match parse_deps(&source[source_len..]) {
           Ok(deps) => deps,
           Err(source) => return Err(Error::Parse { path: path.into(), source }),
       };
       todo.extend(deps);

       // Before moving on, remember the next file starts here
       source_len = source.len();
   }
}
```

The idea is that we populate a buffer given by the user to which I can refer. Now, unfortunately, Rust gives me (something like) the error below:

```
error[E0502]: cannot borrow `*source` as mutable because it is also borrowed as immutable
  --> eflint-parser/src/load.rs:90:38
   |
81  | ...ild_file<'s>(path: PathBuf, source: &'s mut Vec<u8>) -> Result<...
   |                                        -- lifetime `'s` defined here
...
90  | ...       if let Err(source) = handle.read_to_end(source) {
   |                                ^^^^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
...
98  | ...et mut deps: Vec<PathBuf> = match parse_deps(&source[source_len..]) {
   |                                                 ------ immutable borrow occurs here
99  | ...   Ok(deps) => deps,
100 | ...   Err(source) => return Err(Error::Parse { path: path.into(), source }),
   |                           ---------------------------------------------- returning this value requires that `*source`
```

Pretty sure it has something to do with the mutable borrow being invariant, so Rust can't shorten it, and hence this `'s`-requirement by returning it it imposed onto the mutable borrow so it has to be valid for the entire lifetime of the function. Or something like that.

But I don’t see why this wouldn’t be sound to do; in essence, my program uses the mutable- and read-only borrows exclusively from each other (not at the same time), and *only* returns a longer-lived, read-only borrow when it returns.

Is there any way to express what I want to do? To shorten this mutable borrow’s lifetime so that I *can* iterate, but then still return a reference to the buffer upon errors?

I hope to hear from you guys.

Best,
Tim

quinedot August 5, 2025, 1:44pm 2

I'm pretty sure this is due to conditional return of a borrow.[1]

lut_99:

> Pretty sure it has something to do with the mutable borrow being invariant, so Rust can’t shorten it, and hence this `'s`-requirement by returning it it imposed onto the mutable borrow so it has to be valid for the entire lifetime of the function. Or something like that.

(Assuming I'm correct) the variance isn't a problem; it's creating a borrow that has to be valid for the entire lifetime of the function *and beyond* -- valid for all of `'a`.

If you can recreate the error from some known location, that may be an option on stable. There is also a crate that may help. The docs also include "Non-`unsafe` albeit cumbersome workarounds for lack-of-Polonius issues" (but I don't know if they apply here).

> My attempt at reproduction, which works with Polonius. ■■

1 Like

kpreid August 5, 2025, 1:53pm 3

You can solve this problem by using `typed-arena` as an *append-only* data structure, eliminating any possible borrow conflict because appending only needs a shared reference. Its use would look like:

```
fn build_file<'s>(path: PathBuf, sources: &'s typed_arena::Arena<Vec<u8>>) -> Result<(), Error<'s>> {
   ...
   while let Some(path) = todo.pop() {
       let source: Vec<u8> = match fs::read(&path) {
           ...
       let source_ref: &'s [u8] = sources.alloc(source);
       // ... parse and error from source_ref ...
```

(`Arena::alloc_extend()` might allow you to store the bytes directly in an `Arena<u8>`, but looking at the source code, I’m not sure whether it would be efficient at copying the large number of individual byte items.)

4 Likes

lut_99 August 5, 2025, 2:04pm 4

Thank you both for your fast replies!

Thanks for linking to Polonius-the-crab (the crate has a great theme, by the way xD) and associated reading, at least I now how to call the problem

I’ll think I’ll look into the Polonius crate first, understand what it’s doing. Otherwise, the arena also looks like a very good solution. I’ll let you know which I’ll settle on and why.

Thanks, again!

lut_99 August 5, 2025, 4:20pm 5

I’m using polinius-the-crab now, and so far, everything seems to work! I’d say it’s definitely a bit trickier to understand than typed-arena, but as an answer to my question it was a perfect fit and worked pretty much out-of-the-box.

That says, for my *overall* use-case, when I come back to this when I have time, I’ll reconsider typed-arena again - looking at their source code, looks like I can allocate new members *while* keeping read-references to old members around. So maybe I won’t even need this double-pass parse strategy that way.

Anyway, thanks a bunch for now!

jumpnbrownweasel August 5, 2025, 5:00pm 6

lut_99:

> I can allocate new members *while* keeping read-references to old members around

Just in case you didn’t notice, you can also keep `&mut` references to old members around (only one of them per member of course).

2 Likes

system Closed November 3, 2025, 5:00pm 7

This topic was automatically closed 90 days after the last reply. We invite you to open a new topic if you have further questions or comments.

Related topics

Topic		Replies	Views	Activity
Borrow error when adding lifetime field to a Result::Err help	8	734	September 23, 2019	
Simple question about borrowing and references help	12	488	January 12, 2023	
"nested" lifetimes in structs? Is this possible to express? help	7	784	October 18, 2020	
Mutable and immutable borrow and lifetime parameters? help	4	1329	January 12, 2023	
Unexpected repeated mutable borrow duration help	4	406	August 11, 2020	