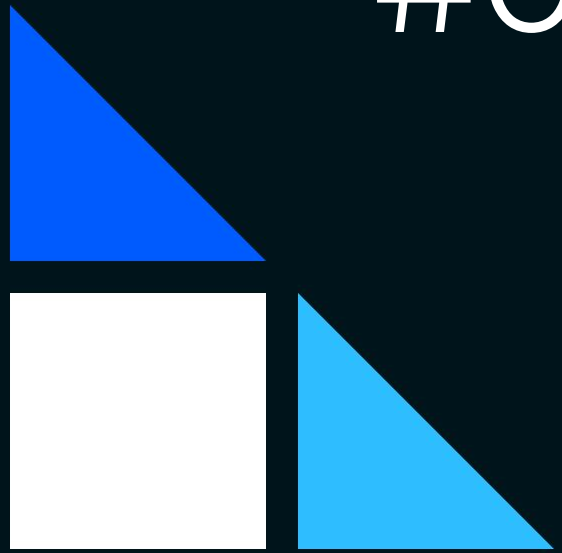
Blockchain In Rust

#05: Transactions 2



geeklaunch

not a geek? start today!

Iterators

Iterators help us process sequences of data. Rust's iterators are especially powerful.

To access the iterator of a vector, call its .iter() function. Then we have access to lazily-evaluated functions like map, flat_map, filter, for_each, any, all, etc.

Another powerful function is .collect::<B>(). This will evaluate the iterator and form it into whatever type B you give it. The type B must implement the FromIterator trait, but most of the standard library structures you'd expect this to work on do. For example: String, Vec, HashMap, HashSet, LinkedList, etc.

Implement: Transaction & Output

Errors

In Rust, there are essentially two types of errors: Result<T, E> and panic!. Result<T, E>s are just like any other data type, it's just part of the standard library and recommended that you use it. They take the form of either Result::Ok(T) or Result::Err(E), and you can destructure them as necessary. panic!s, on the other hand, are not usable just like any other data type. They actually will halt the program, like it crashed. Ideally, we want to use Result<T, E>s over panic!s.

See:

- https://doc.rust-lang.org/std/result/enum.Result.html
- https://doc.rust-lang.org/std/macro.panic.html

Null

No such thing as null in Rust!

The closest we can get is the standard library's Option<T>: an enum that communicates the existence of a value. It can take the form of Option::Some(T) or Option::None, where Option::None is probably the closest you will see to null in Rust programs.

See: https://doc.rust-lang.org/std/option/enum.Option.html

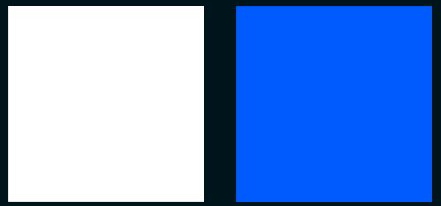
Updating our blockchain

Now we have to maintain a list of unspent outputs. This will just be a set of hashes of the unspent outputs. Note that this does not differentiate between two outputs that are to the same address for the same amount. This will be fixed later.

We have to validate three more conditions now:

- Can we spend the input?
- How many coins are in the output?
- Is the coinbase transaction valid? (We're going to skimp a bit on this check for now.)

Implement: Transaction checking

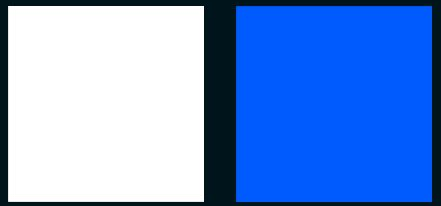


Writing a working example

We need to…

1. Create a genesis block with transactions
2. Mine it
3. Add it to the blockchain
4. Create another block with more transactions (particularly some that use transactions from the first block)
5. Mine that one
6. Add that one to the blockchain

Implement: Working example



Notes

THIS IS NOT SECURE! (Or efficient, really)

I am not a security professional.

That said, here are some things to take into account about the code we just wrote:

- The difficulty stored in a block is not validated.
- The value of the coinbase transaction is not validated.
- “Coin ownership” is neither enforced nor existent.
- Two otherwise identical outputs from different transactions are indistinguishable.
- And more!