Rewriting my toy blockchain in Rust

I started 2022 with a mini batch at the Recurse Center. I did a full batch in Fall 2020, and hanging out with people working on their side projects full time seems like a fun way to kick off the year.

My plan was to port my toy blockchain from Python to Rust (repo here, write-up here). The initial project had multiple iterations, starting with proof-of-work and later Merkle trees and elliptic curve cryptography - a suitable candidate for a 1-week rewrite.

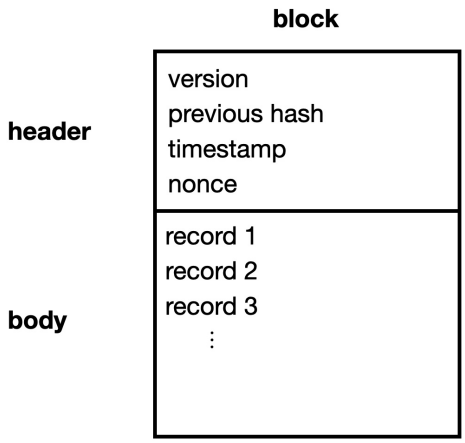
As with a lot of RC projects (for me anyway), things rarely play out as planned. I had moments of confusion with the Rust compiler, moved past that to contemplate working on speedups, but ended up compiling to WebAssembly to enable mining in the browser.

What I thought made for a nice ending was being able to connect the mini batch project with what I had previously worked on at RC.

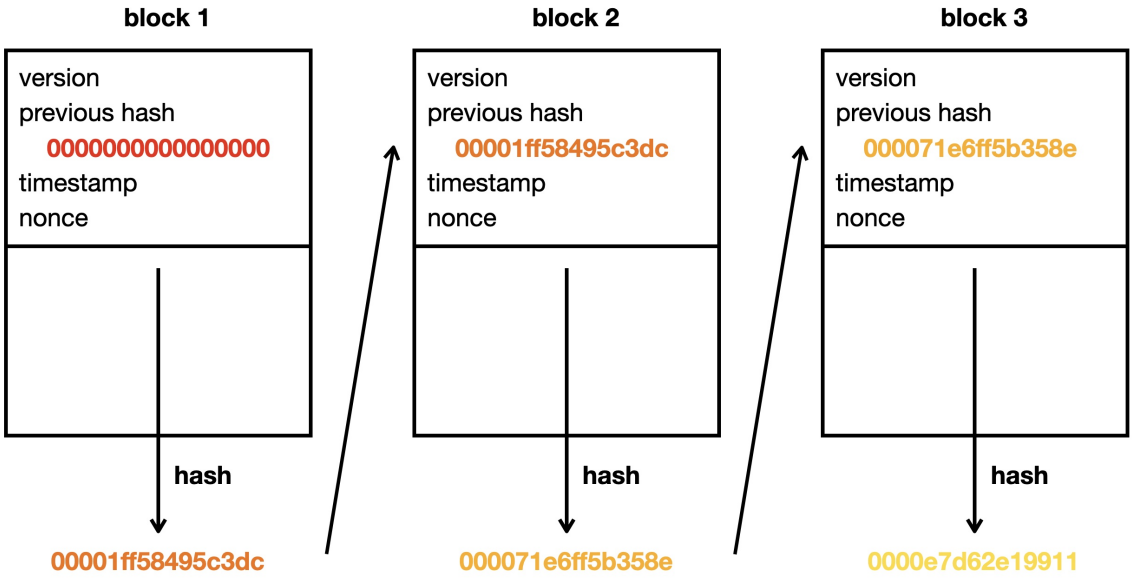
Blockchain 101

For our purposes, a blockchain is a distributed database where records are divided into 'blocks' and the blocks form a 'chain'.

Each block is made up of a header (where block attributes live) and a body (where records live).



The hash of the header is used to identify each block (commonly referred to as the **block hash**) and is included in the header of the next block (hence the chain).



The Python function to initialize the header is relatively straightforward - a simple conversion from ints to bytes followed by a concatenation. Note that all the attributes are pre-determined except for the nonce.

```
VERSION: int = 0

def init_header(previous_hash: bytes, timestamp: int, nonce: int) -> bytes:
    """Initialize header."""
    return (
        VERSION.to_bytes(1, byteorder="big")
```

```
        + previous_hash
        + timestamp.to_bytes(4, byteorder="big")
        + nonce.to_bytes(32, byteorder="big")
    )
```

The nonce is chosen such that the leading bytes of the block hash are zero. The more zeroes we want, the more difficult it is to find the nonce. In the example below we require the two leading bytes be zero, or four leading zeroes in hex. The first ten integers doesn't get us there.

```
previous_hash = (0).to_bytes(32, byteorder="big")
timestamp = 1634700000

def sha256_2x(header: bytes) -> bytes:
    """Apply SHA-256 twice."""
    return hashlib.sha256(hashlib.sha256(header).digest()).hexdigest()

for nonce in range(10):
    header = init_header(previous_hash, timestamp, nonce)
    print(f"nonce: {nonce}, hash: {sha256_2x(header)}")
```

```
nonce: 0, hash: 489dbdf5fd580fdc0fc372546a83467e1001718409f00c962590727fba21cf2a
nonce: 1, hash: 584b740dee7dd5b18d906338040e90a735ce2e4c89034e756960de16d8ad6635
nonce: 2, hash: 6501e86d4d5fb1fcd51e48e396c88ef0bf38c64d7981c5d9d05b048c588434b3
nonce: 3, hash: adf71f94f264f496b2b344c8c9afa9aa177b85261342130f1034cdc66165375b
nonce: 4, hash: a2a302d40611fb0806fc4e623bbf744212ebefacefd5e6d54f7cfe27b7fc637c
nonce: 5, hash: 65c80fd5c4d9f87ebbaab9da9a908615f73966019dc70eca7c3a7db6a21bf986
nonce: 6, hash: f03aba3135591c0ef0b042bfdfc095102e8008401dddcac122f227267739b1c9
nonce: 7, hash: cd4ebbe1ceceae965262f5e8037912fd100a10fe10bfea0465209df97a4a82fd
nonce: 8, hash: 4d81346ceea40e57411c4d1dc5468a9ad675853ca87e666d0d81bb05bef5a88b
nonce: 9, hash: e1b0f93e7d5c889a250b719840598847de66b85585e9b409b5683e96c7ac6765
```

With 70,822 we get what we want.

```
nonce = 70822

header = init_header(previous_hash, timestamp, nonce)
print(f"nonce: {nonce}, hash: {sha256_2x(header)}")
```

```
nonce: 70822, hash: 00001ff58495c3dc2a1aaa69ebca4e9b3e05e63e5a319fb73bcdccbcdbba1e72
```

Writing this as a Python function, we have a loop that incrementally iterates though potential nonce values and returns when the desired leading bytes are zero.

```
def run_proof_of_work(previous_hash: bytes, timestamp: int) -> Tuple[int, bytes]:
    """Find nonce that results in leading zero bytes in the block header."""
    nonce = 0

    while True:
        header = init_header(previous_hash, timestamp, nonce)
        guess = hashlib.sha256(hashlib.sha256(header).digest())

        if guess.hexdigest()[:4] == "0000":
            return nonce, guess.digest()

        nonce += 1
```

Don’t be a hero

Following Andrej Karpathy‘s advice to use best practices, I looked up how to concatenate bytes in Rust on Stack Overflow. I found the following post.

When I played around with `.concat()` to see whether the bytes were being moved or copied, I encountered counter-intuitive behavior depending on whether I was concatenating two references or one. This compiles OK (all code runs in main).

```
let a: &mut [u8] = &mut [0xdeu8, 0xadu8];
let b: &[u8] = &[0xbeu8, 0xefu8];
let c: Vec<u8> = [a, b].concat();

println!("a before: {:x?}", a);
println!("c before: {:x?}\n", c);

a[0] = 0x00u8;

println!("a after : {:x?}", a);
println!("c after : {:x?}", c);
```

```
a before: [de, ad]
c before: [de, ad, be, ef]

a after : [0, ad]
c after : [de, ad, be, ef]
```

However the compiler was not happy when I only have one reference.

```
let a: &mut [u8] = &mut [0xdeu8, 0xadu8];
let c: Vec<u8> = [a].concat();

println!("a before: {:x?}", a);
```

```
println!("c before: {:x?}\n", c);

a[0] = 0x00u8;

println!("a after : {:x?}", a);
println!("c after : {:x?}", c);
```

```
error[E0382]: use of moved value: `a`
  --> src/main.rs:92:1
   |
86 | let a: &mut [u8] = &mut [0xdeu8, 0xadu8];
   |     - move occurs because `a` has type `&mut [u8]`, which does not implement the `Copy` trait
87 | let c: Vec<u8> = [a].concat();
   |                    - value moved here
...
92 | a[0] = 0x00u8;
   | ^^^^ value used here after move

For more information about this error, try `rustc --explain E0382`.
```

What gives? 🤔

Ask the experts

I looked in the docs but found few clues. The cool thing about RC is you get to ask RC Rust experts. I posed the question on #rust channel on RC Zulip, and got very helpful responses from Jesse Luehrs and Mikkel Paulson.

The basic idea is that the compiler doesn't allow copying of mutable references because there cannot be multiple mutable references to the same data. Copying immutable references, on the other hand, is OK.

In concatenating mutable and immutable references, the compiler will unify the types since all elements of the collection need to be the same type. Casting from mutable to immutable is safe, so the result is immutable. Immutable references implement copying and thus the elements are copied into a single contiguous block.

In the case of the single immutable reference, the compiler can't "help you to get your code to work" since the mutable reference doesn't implement copying; it is moved instead.

Taking this further, concatenating mutable references together gets you a mutable reference and hence a similar error to the single mutable reference example.

```
let a: &mut [u8] = &mut [0xdeu8, 0xadu8];
let b: &mut [u8] = &mut [0xbeu8, 0xefu8];
let c: Vec<u8> = [a, b].concat();

println!("a before: {:x?}", a);
println!("c before: {:x?}\n", c);

a[0] = 0x00u8;

println!("a after : {:x?}", a);
println!("c after : {:x?}", c);
```

```
error[E0382]: use of moved value: `a`
  --> src/main.rs:93:1
   |
86 | let a: &mut [u8] = &mut [0xdeu8, 0xadu8];
   |     - move occurs because `a` has type `&mut [u8]`, which does not implement the `Copy` trait
87 | let b: &mut [u8] = &mut [0xbeu8, 0xefu8];
88 | let c: Vec<u8> = [a, b].concat();
   |                   - value moved here
...
93 | a[0] = 0x00u8;
   | ^^^^ value used here after move

For more information about this error, try `rustc --explain E0382`.
```

My take on the borrow checker went from “why are you doing this to me?” to “thanks for helping out!” after wrapping my head around what was going on.

Stack vs heap

I initially thought that the copying happened because the result is on the heap. It turns out this was a red herring - I get the same error when the result lives on the stack. To better illustrate this, here’s the example above i.e. the heap version (with vectors).

```
let a: &mut [u8] = &mut [0xdeu8, 0xadu8];
let b: &[u8] = &[0xbeu8, 0xefu8];
let c: Vec<u8> = [a, b].concat();

println!("a before: {:x?}", a);
println!("c before: {:x?}\n", c);

a[0] = 0x00u8;

println!("a after : {:x?}", a);
println!("c after : {:x?}", c);
```

```
a before: [de, ad]
c before: [de, ad, be, ef]

a after : [0, ad]
c after : [de, ad, be, ef]
```

Now the stack version (with arrays).

```
let a: &mut [u8] = &mut [0xdeu8, 0xadu8];
let b: &[u8] = &[0xbeu8, 0xefu8];
let c: &[u8] = &[a, b].concat();

println!("a before: {:x?}", a);
println!("c before: {:x?}\n", c);

a[0] = 0x00u8;

println!("a after : {:x?}", a);
println!("c after : {:x?}", c);
```

```
a before: [de, ad]
c before: [de, ad, be, ef]

a after : [0, ad]
c after : [de, ad, be, ef]
```

Re: terminology, I had been using the terms copy vs clone interchangeably. It turns out copying is automatically generated by the compiler, whereas cloning happens through calls to `.clone()`. In other words, copy is implicit and clone explicit.

RIIR: Blockchain edition

Thankfully the rest of the port was smooth. In the end we have the header initialization code:

```
VERSION: int = 0

def init_header(previous_hash: bytes, timestamp: int, nonce: int) -> bytes:
    """Initialize header."""
    return (
        VERSION.to_bytes(1, byteorder="big")
        + previous_hash
        + timestamp.to_bytes(4, byteorder="big")
        + nonce.to_bytes(32, byteorder="big")
    )
```

```
// Initialize header.
fn init_header(previous_hash: &[u8], timestamp: u32, nonce: u64) -> Vec<u8> {
    let timestamp_bytes: &[u8; 4] = &timestamp.to_be_bytes();
    let nonce_bytes: &[u8; 8] = &nonce.to_be_bytes();
    let padding: &[u8; 24] = &[0x00u8; 24];

    return [
        &[VERSION],
        previous_hash,
        timestamp_bytes,
        padding,
        nonce_bytes,
    ]
    .concat();
}
```

The proof-of-work code in Rust is a bit more verbose to keep the compiler happy.

```
def run_proof_of_work(
    previous_hash: bytes,
    timestamp: int,
) -> Tuple[int, bytes]:
    """Find nonce that results in two leading zero bytes in the block header."""
    nonce = 0

    while True:
        header = init_header(previous_hash, timestamp, nonce)
        guess = hashlib.sha256(hashlib.sha256(header).digest())

        if guess.hexdigest()[:4] == "0000":
            return nonce, guess.digest()

        nonce += 1
```

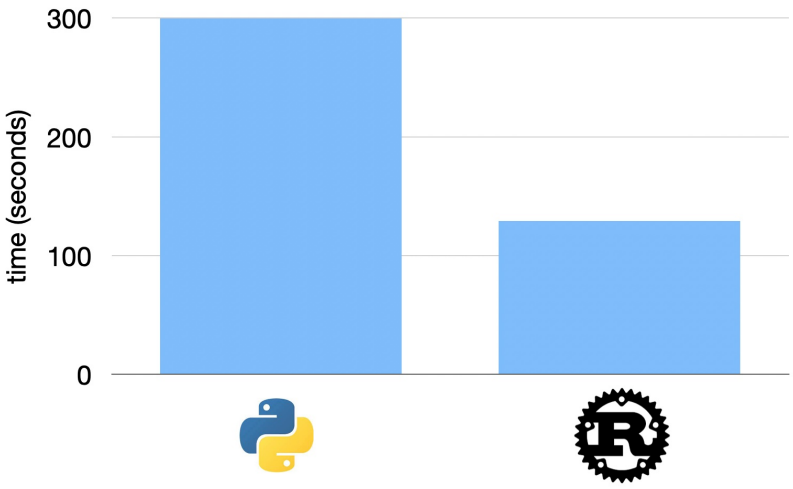
```
// Apply SHA-256 twice.
fn sha256_2x(bytes: &Vec<u8>) -> Vec<u8> {
    let mut intermediate;
    let mut hash = &bytes[..];

    for _ in 0..2 {
        intermediate = Sha256::digest(&hash);
        hash = &intermediate[..];
    }

    return hash.to_vec();
}

// Checks header has desired number of leading zero bytes.
fn has_leading_zeroes(header: &Vec<u8>, difficulty: usize) -> (bool, Vec<u8>) {
    let hash = sha256_2x(header);

    for i in 0..difficulty {
        if hash[i] != 0 {
            return (false, hash);
        }
```

```
    }

    return (true, hash);
}

// Find nonce that results in leading zero bytes in the block header.
fn run_proof_of_work(previous_hash: &[u8], timestamp: u32) -> (u64, Vec<u8>) {
    let mut nonce: u64 = 0;
    let mut header: Vec<u8>;

    loop {
        header = init_header(previous_hash, timestamp, nonce);
        let (is_target_header, current_hash) = has_leading_zeroes(&header, 2);

        if is_target_header {
            return (nonce, current_hash);
        }

        nonce += 1;
    }
}
```

Performance

During my first RC batch I had the chance to benchmark Python against Rust (repo here), to see 10-20x speedups with Rust.

Running the proof-of-work code to mine 10 blocks with three leading zero bytes is approximately 5 minutes in Python and 2 minutes in Rust - I was expecting the speedup but had also thought I would need logarithmic scales.

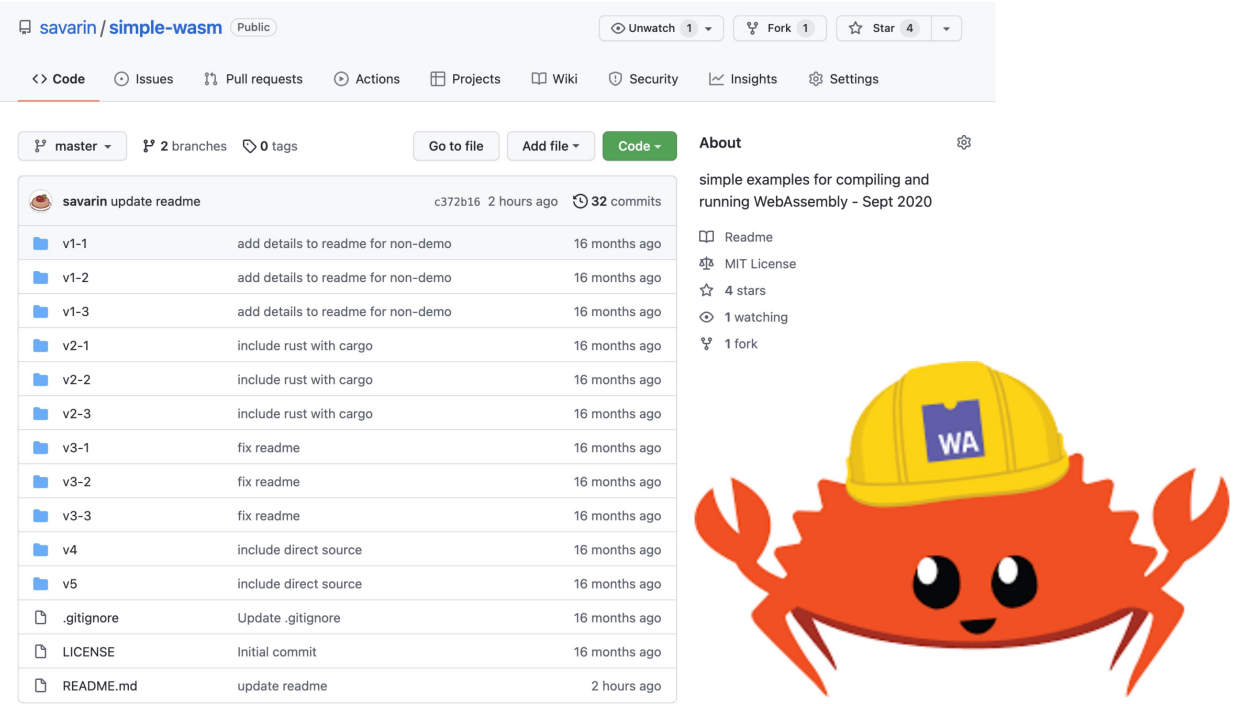


At this stage, I could spend time optimizing the code. Ideas include (thanks to Andrey Petrov) isolating int-to-byte conversions, using bitmaps to determine leading zero bytes, and reducing allocations by using arrays.

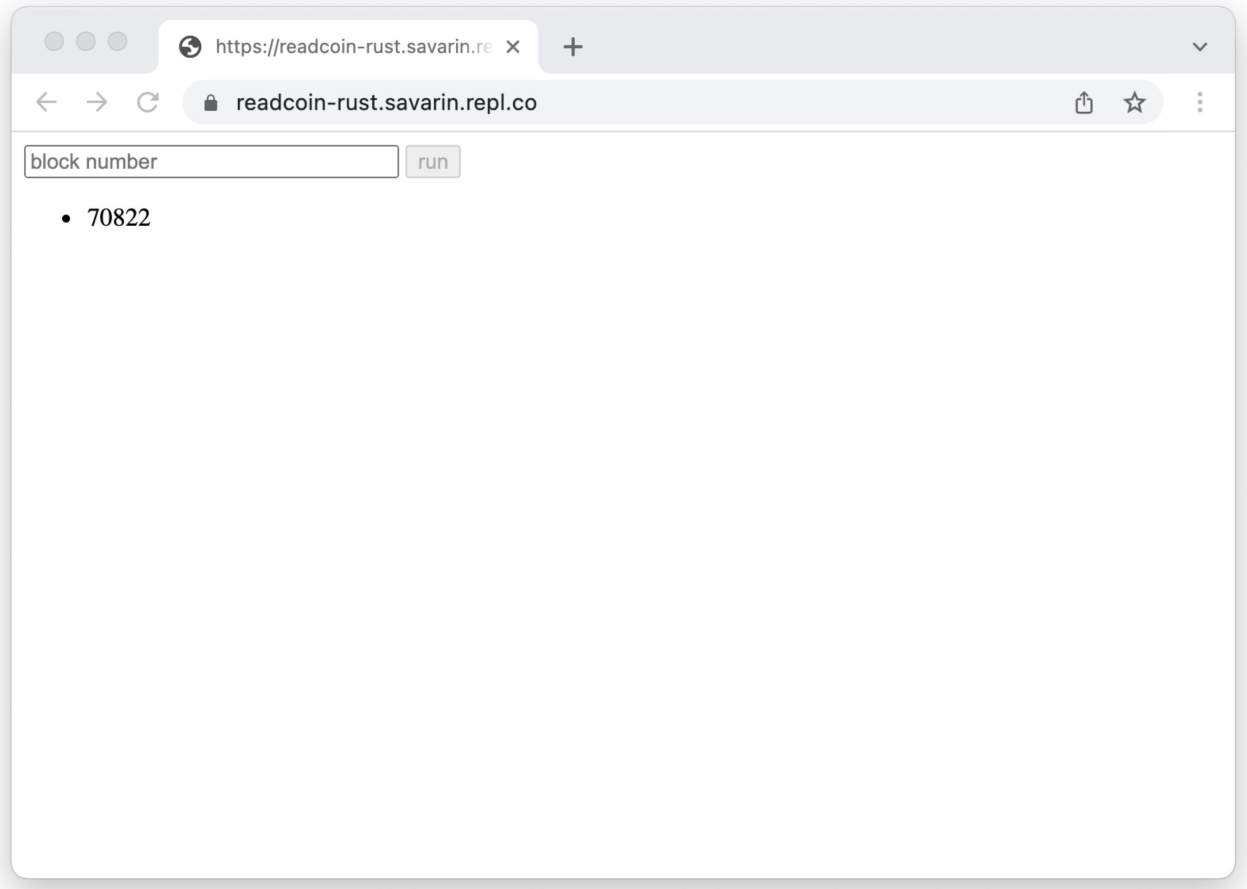
True to form, I went on a tangent...

Mining in the browser

I had also worked on compiling Rust to WebAssembly in my first batch (repo here), so maybe I could get the code to run in the browser!



Since WebAssembly only has ints and floats, generating block hashes would need a bit more work. To keep things simple, I wrote up a function that takes in the block number and (slowly) finds the desired nonce value (repo here, demo here).



Next steps

The Python version has socket programming to enable multiple miners be run concurrently and broadcast newly-minted blocks to each other.

https://vimeo.com/647162882

Perhaps getting this to run in different browser tabs makes for a nice follow-up. Next batch? 😉