**Rust Codecs and Processing Binary Data**

[[](https://medium.com/@technocrat21?source=post_page-----2609002a02b6--------------------------------)](https://medium.com/@technocrat21?source=post_page-----2609002a02b6--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----2609002a02b6--------------------------------)](https://medium.com/coderhack-com?source=post_page-----2609002a02b6--------------------------------)

[Amay B](https://medium.com/@technocrat21?source=post_page-----2609002a02b6--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----2609002a02b6--------------------------------)

·

4 min read

·

Sep 15

10

1



Photo by [Markus Spiske](https://unsplash.com/@markusspiske?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Codecs are mechanisms to encode and decode between binary data and Rust types. They allow you to serialize Rust structs and enums into raw byte streams and deserialize raw byte streams back into Rust types.

**Built-in Codecs**

Rust has built-in support for a few basic codecs:

* Strings: String, str
* Numbers: u8, i32, f64, etc.

These allow you to represent strings, numbers, and other primitives as byte sequences.

**External Crates for Codecs**

For more complex formats, we can use external crates:

* **serde**: Generic serializer framework
* **serde\_json**: JSON format
* **serde\_yaml**: YAML format
* **serde\_cbor**: CBOR format
* **bincode**: Compact binary format
* **prost**: Protocol Buffers
* **capnp**: Cap’n Proto
* And many more!

These crates provide Codecs to serialize and deserialize Rust types to and from various binary formats. For example, with serde\_json, we can convert between JSON string data and Rust structs.

In the next sections, we’ll explore how to work with binary data in Rust, see examples of these Codecs in action, implement our own custom Codec, and more!Here is section 2 of the outline in Markdown format:

**II. Working with Binary Data in Rust**

**A. Representing Binary Data**

We have a few options for representing raw binary data in Rust:

* [u8]: A vector of raw byte values.
* &[u8]: A slice of raw byte values.
* Cursor<&[u8]>: A read/write cursor over a byte slice. This allows reading/writing binary data without loading the entire dataset into memory at once.

For example:

let bytes = vec![1, 2, 3]; // [u8]  
let slice = &bytes[..]; // &[u8]  
let cursor = Cursor::new(slice); // Cursor<&[u8]>

**B. Reading Binary Data**

There are a few ways to read binary data in Rust:

* File::open(): Open a file and get a byte slice reference to the contents.
* std::fs::read(): Read an entire file into a byte vector.
* Cursor::new(): Create a read cursor over an existing byte slice.
* Cursor::read(): Read data from a cursor into some target type using an appropriate Codec.

For example:

let mut f = File::open("data.bin").unwrap();  
let mut buffer = vec![];  
f.read\_to\_end(&mut buffer).unwrap();  
  
let cursor = Cursor::new(&buffer);  
let value = cursor.read\_i32::<LittleEndian>().unwrap();

**C. Writing Binary Data**

We can write binary data in similar ways:

* File::create(): Create a new file and get a Write cursor to it.
* Cursor::write(): Write from some source type to a cursor using an appropriate Codec.
* std::fs::write(): Write an entire byte vector to a file.

For example:

let mut f = File::create("data.bin").unwrap();  
let cursor = Cursor::new(&mut f);  
cursor.write\_i32::<LittleEndian>(10).unwrap();

This covers the basics of representing, reading, and writing raw binary data in Rust! Let me know if you would like me to elaborate on any part of this section.Here is section 3 in markdown format with code examples:

**III. Examples**

**A. Serializing/Deserializing with Serde**

We can define Rust structs with Serde attributes to enable serialization and deserialization.

use serde::{Serialize, Deserialize};  
  
#[derive(Serialize, Deserialize)]   
struct User {  
 name: String,  
 age: u8,  
 phones: Vec<String>  
}  
  
// Serialize to JSON  
let user = User {  
 name: "John".to\_string(),  
 age: 30,  
 phones: vec!["555-1234".to\_string(), "555-5678".to\_string()]  
};  
let user\_json = serde\_json::to\_string(&user).unwrap();  
  
// Deserialize from JSON  
let de\_user: User = serde\_json::from\_str(&user\_json).unwrap();

We can also use Serde to serialize/deserialize to/from YAML, CBOR, Protobuf, and more.

**B. Writing a custom Codec**

To implement a custom binary format, we define structs implementing the serde::ser::Serializer and serde::de::Deserializer traits.

struct MyCodec;  
  
impl Encoder for MyCodec {  
 type Encoding = MyEncoding;  
  
 fn encode<T: Serialize>(&self, value: &T) -> MyEncoding {  
 // Encode value to MyEncoding  
 }  
}  
  
impl Decoder for MyCodec {  
 type Decoding = MyDecoding;  
  
 fn decode<T: DeserializeOwned>(&self, dec: MyDecoding) -> Result<T, Error> {  
 // Decode MyDecoding to T, return Result  
 }   
}   
  
// Usage  
let bytes = my\_codec.encode(&user);  
let de\_user = my\_codec.decode::<User>(bytes).unwrap();

We call .encode() to serialize to our format, and .decode() to deserialize from our format.

**C. Converting Between Formats**

We can read in data of one format, serialize to raw bytes, then deserialize from those raw bytes into another format.

// Read JSON   
let user: User = serde\_json::from\_str(json\_data);  
  
// Serialize to raw bytes with serde\_json  
let json\_bytes = serde\_json::to\_vec(&user).unwrap();  
  
// Deserialize from raw bytes with serde\_cbor   
let user: User = serde\_cbor::from\_slice(&json\_bytes).unwrap();  
  
// Now have user in CBOR format!

By first converting to an intermediate raw byte representation, we can convert between any two formats that have Serde serializers/deserializers.

**D. Cursor Usage**

We can use std::io::Cursor<&[u8]> to get a read cursor over some byte slice, and read from that cursor incrementally to decode data without loading the entire byte slice into memory at once.

let data = [1, 2, 3, 4, 5];  
let mut cursor = Cursor::new(&data);  
  
let b1: u8 = dec.decode(&mut cursor).unwrap(); // b1 = 1  
let b2: u8 = dec.decode(&mut cursor).unwrap(); // b2 = 2  
// ...

Similarly, we can get a std::io::Cursor<Vec<u8>> for writing, and encode to it incrementally without creating the entire byte vector at once.

**E. Binary Diffing**

We can hash binary data at the byte level to get a “fingerprint”, and then compare fingerprints to see if two binary files are different, without comparing the full contents.

// Hash function   
fn hash(input: &[u8]) -> u64 {  
 let mut hasher = DefaultHasher::new();  
 hasher.write(input);  
 hasher.finish()   
}  
  
// Get fingerprint   
let f1 = hash(&file1\_data);   
  
// Compare fingerprints   
if f1 == hash(&file2\_data) {  
 println!("Files are equal!");   
} else {  
 println!("Files are different!");  
}

This can be useful for efficiently checking if binary files have changed, without re-reading their entire contents. I hope this article has been helpful to you! If you found it helpful please support me with 1) click some claps and 2) share the story to your network. Let me know if you have any questions on the content covered.

Feel free to contact me at coderhack.com(at)gmail.com