I/O & Serialization

CIS 198 Lecture 8

I/O

Traits!

```
pub trait Read {
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

    // Other methods implemented in terms of read().
}

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    // Other methods implemented in terms of write() and flush().
}
```

- Standard IO traits implemented for a variety of types:
  - `File`s, `TcpStream`s, `Vec<T>`s, `&[u8]`s.
- Careful: return types are `std::io::Result`, not `std::Result`!
  - `type Result<T> = Result<T, std::io::Error>;`

std::io::Read

```
use std::io;
use std::io::prelude::*;
use std::fs::File;

let mut f = try!(File::open("foo.txt"));
let mut buffer = [0; 10];

// read up to 10 bytes
try!(f.read(&mut buffer));
```

- `buffer` is an array, so the max length to read is encoded into the type.
- `read` returns the number of bytes read, or an `Err` specifying the problem.
  - A return value of `Ok(n)` guarantees that `n <= buf.len()`.
  - It can be `0`, if the reader is empty.

Ways of Reading

```
/// Required.
fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

/// Reads to end of the Read object.
fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize>

/// Reads to end of the Read object into a String.
fn read_to_string(&mut self, buf: &mut String) -> Result<usize>

/// Reads exactly the length of the buffer, or throws an error.
fn read_exact(&mut self, buf: &mut [u8]) -> Result<()>
```

- `Read` provides a few different ways to read into a variety of buffers.
  - Default implementations are provided for them using `read`.
- Notice the different type signatures.

Reading Iterators

```
fn bytes(self) -> Bytes<Self> where Self: Sized

// Unstable!
fn chars(self) -> Bytes<Self> where Self: Sized
```

- `bytes` transforms some `Read` into an iterator which yields byte-by-byte.
- The associated `Item` is `Result<u8>`.
  - So the type returned from calling `next()` on the iterator is `Option<Result<u8>>`.
  - Hitting an `EOF` corresponds to `None`.
- `chars` does the same, and will try to interpret the reader's contents as a UTF-8 character sequence.
  - Unstable; Rust team is not currently sure what the semantics of this should be. See issue #27802.

Iterator Adaptors

```
fn chain<R: Read>(self, next: R) -> Chain<Self, R>
    where Self: Sized
```

- `chain` takes a second reader as input, and returns an iterator over all bytes from `self`, then `next`.

```
fn take<R: Read>(self, limit: u64) -> Take<Self>
    where Self: Sized
```

- `take` creates an iterator which is limited to the first `limit` bytes of the reader.

std::io::Write

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    // Other methods omitted.
}
```

- `Write` is a trait with two required methods, `write()` and `flush()`
  - Like `Read`, it provides other default methods implemented in terms of these.
- `write` (attempts to) write to the buffer and returns the number of bytes written (or queued).
- `flush` ensures that all written data has been pushed to the target.
  - Writes may be queued up, for optimization.
  - Returns `Err` if not all queued bytes can be written successfully.

Writing

```
let mut buffer = try!(File::create("foo.txt"));

try!(buffer.write("Hello, Ferris!"));
```

Writing Methods

```rust
/// Attempts to write entire buffer into self.
fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }

/// Writes a formatted string into self.
/// Don't call this directly, use `write!` instead.
fn write_fmt(&mut self, fmt: Arguments) -> Result<()> { ... }

/// Borrows self by mutable reference.
fn by_ref(&mut self) -> &mut Self where Self: Sized { ... }
```

`write!`

- Actually using writers can be kind of clumsy when you're doing a general application.
  - Especially if you need to format your output.
- The `write!` macro provides string formatting by abstracting over `write_fmt`.
- Returns a `Result`.

```
let mut buf = try!(File::create("foo.txt"));

write!(buf, "Hello {}!", "Ferris").unwrap();
```

IO Buffering

- IO operations are really slow.
- Like, *really* slow:

```
TODO: demonstrate how slow IO is.
```

- Why?

IO Buffering

- Your running program has very few privileges.
- Reads are done through the operating system (via system call).
  - Your program will do a *context switch*, temporarily stopping execution so the OS can gather input and relay it to your program.
  - This is veeeery slow.
- Doing a lot of reads in rapid succession suffers hugely if you make a system call on every operation.
  - Solve this with buffers!
  - Read a huge chunk at once, store it in a buffer, then access it little-by-little as your program needs.
- Exact same story with writes.

BufReader

```
fn new(inner: R) -> BufReader<R>;

let mut f = try!(File::open("foo.txt"));
let buffered_reader = BufReader::new(f);
```

- `BufReader` is a struct that adds buffering to *any* reader.
- `BufReader` itself implements `Read`, so you can use it transparently.

BufReader

- `BufReader` also implements a separate interface `BufRead`.

```
pub trait BufRead: Read {
    fn fill_buf(&mut self) -> Result<&[u8]>;
    fn consume(&mut self, amt: usize);

    // Other optional methods omitted.
}
```

BufReader

- Because `BufReader` has access to a lot of data that has not technically been read by your program, it can do more interesting things.
- It defines two alternative methods of reading from your input, reading up until a certain byte has been reached.

```
fn read_until(&mut self, byte: u8, buf: &mut Vec<u8>)
    -> Result<usize> { ... }
fn read_line(&mut self, buf: &mut String)
    -> Result<usize> { ... }
```

- It also defines two iterators.

```
fn split(self, byte: u8)
    -> Split<Self> where Self: Sized { ... }
fn lines(self)
    -> Lines<Self> where Self: Sized { ... }
```

BufWriter

- `BufWriter` does the same thing, wrapping around writers.

```
let f = try!(File::create("foo.txt"));
let mut writer = BufWriter::new(f);
try!(buffer.write(b"Hello world"));
```

- `BufWriter` doesn't implement a second interface like `BufReader` does.
- Instead, it just caches all writes until the `BufWriter` goes out of scope, then writes them all at once.

StdIn

```
let mut buffer = String::new();

try!(io::stdin().read_line(&mut buffer));
```

- This is a very typical way of reading from standard input (terminal input).
- `io::stdin()` returns a value of `struct StdIn`.
- `stdin` implements `read_line` directly, instead of using `BufRead`.

StdInLock

- A "lock" on standard input means only that current instance of `StdIn` can read from the terminal.
  - So no two threads can read from standard input at the same time.
- All `read` methods call `self.lock()` internally.
- You can also create a `StdInLock` explicitly with the `stdin::lock()` method.

```
let lock: io::StdInLock = io::stdin().lock();
```

- A `StdInLock` instance implements `Read` and `BufRead`, so you can call any of the methods defined by those traits.

`StdOut`

- Similar to `StdIn` but interfaces with standard output instead.
- Directly implements `Write`.
- You don't typically use `stdout` directly.
  - Prefer `print!` or `println!` instead, which provide string formatting.
- You can also explicitly `lock` standard out with `stdout::lock()`.

Special IO Structs

- `repeat(byte: u8)`: A reader which will infinitely yield the specified byte.
  - It will always fill the provided buffer.
- `sink()`: "A writer which will move data into the void."
- `empty()`: A reader which will always return `Ok(0)`.
- `copy(reader: &mut R, writer: &mut W) -> Result<u64>`: copies all bytes from the reader into the writer.

Serialization

rustc-serialize

- Implements automatic serialization for Rust structs.
  - (Via compiler support.)
- Usually used with JSON output:

```
extern crate rustc_serialize;
use rustc_serialize::json;

#[derive(RustcDecodable, RustcEncodable)]
pub struct X { a: i32, b: String }

fn main() {
    let object = X { a: 6, b: String::from("half dozen") };
    let encoded = json::encode(&object).unwrap();
    // ==> the string {"a":6,"b":"half dozen"}
    let decoded: X = json::decode(&encoded).unwrap();
}
```

- Also has support for hex- and base64- encoded text output.

Serde

- **Ser**ialization/**De**serialization.
- Next generation of Rust serialization: faster, more flexible.
  - But API is currently in flux! We're talking about serde 0.7.0, released yesterday. (Not on crates.io as of this writing.)
- Serde is easy in Rust nightly!
  - A compiler plugin creates attributes and auto-derived traits.
- Slightly harder to use in Rust stable:
  - Compiler plugins aren't available.
  - Instead, Rust code is generated before building (via `build.rs`).
    - `serde_codegen` generates `.rs` files from `.rs.in` files.
  - And you use the `include!` macro to include the resulting files.
- Separate crates for each output format:
  - Support for binary, JSON, MessagePack, XML, YAML.

Serde

- Code looks similar to `rustc_serialize`:

```
#![feature(custom_derive, plugin)]
#![plugin(serde_macros)]

extern crate serde;
extern crate serde_json;

#[derive(Serialize, Deserialize, Debug)]
pub struct X { a: i32, b: String }

fn main() {
    let object = X { a: 6, b: String::from("half dozen") };
    let encoded = serde_json::to_string(&object).unwrap();
    // ==> the string {"a":6,"b":"half dozen"}
    let decoded: X = serde_json::from_str(&encoded).unwrap();
}
```

Serde

- But there are more features!
- Serializers are generated using the visitor pattern, producing code like the following.
  - Which can also be written manually and customized.

```
use serde;
use serde::*;
use serde::ser::*;

struct Point { x: i32, y: i32 }

impl Serialize for Point {
    fn serialize<S>(&self, sr: &mut S)
            -> Result<(), S::Error> where S: Serializer {
        sr.serialize_struct("Point",
            PointMapVisitor { value: self, state: 0 })
    }
}
```

- ...

Serde

```
struct PointMapVisitor<'a> { value: &'a Point, state: u8 }

impl<'a> MapVisitor for PointMapVisitor<'a> {
fn visit<S>(&mut self, sr: &mut S)
    -> Result<Option<()>, S::Error> where S: Serializer {
  match self.state {
    0 => { // On first call, serialize x.
      self.state += 1;
      Ok(Some(try!(sr.serialize_struct_elt("x", &self.value.x))))
    }
    1 => { // On second call, serialize y.
      self.state += 1;
      Ok(Some(try!(sr.serialize_struct_elt("y", &self.value.y))))
    }
    _ => Ok(None) // Subsequently, there is no more to serialize.
  }
}
}
```

- Deserialization code is also generated - similar but messier.

Serde

- Custom serializers are flexible, but complicated.
- Serde also provides customization via `#[serde(something)]` attributes. `something` can be:
  - On fields and enum variants:
    - `rename = "foo"`: overrides the serialized key name
  - On fields:
    - `default`: use `Default` trait to generate default values
    - `default = "func"` use `func()` to generate default values
    - `skip_serializing`: skips this field
    - `skip_serializing_if = "func"`: skips this field if `!func(val)`
    - `serialize_with = "enc"`: serialize w/ `enc(val, serializer)`
    - `deserialize_with = "dec"`: deserialize w/ `dec(deserializer)`
  - On containers (structs, enums):
    - `deny_unknown_fields`: error instead of ignoring unknown fields