Practical Rust

Environment Variables

In order to get/set environment variables, we use the `std::env` module.

We can get at an individual var using

```
main() -> Result<(), Box<dyn std::error::Error>> {
    use std::env;
    let var = env::var("HOME")?;
}
```

function signature

The var function takes a key of type K where K implements AsRef.

```
pub fn var<K: AsRef<OsStr>>(key: K) -> Result<String, VarError>
```

AsRef is used to do cheap reference-to-reference conversion. In practice, this means that you can use &str, &OsStr, &Path

OsStr comes from std::ffi::OsStr. It is a borrowed reference to an OsString.

A type that can represent owned, mutable platform-native strings, but is cheaply inter-convertible with Rust strings.

The need for this type arises from the fact that:

```
On Unix systems, strings are often arbitrary sequences of non-zero bytes,
in many cases interpreted as UTF-8.

On Windows, strings are often arbitrary sequences of non-zero 16-bit
values, interpreted as UTF-16 when it is valid to do so.

In Rust, strings are always valid UTF-8, which may contain zeros.
```

All of this is interesting, but you dont need to know any of it to use var...

vars

vars provides an iterator over all of the environment variables

```
for (key,value) in env::vars() {
    println!("{}:{}", key, value)
}
```

This method will panic if any key or value in the environment is not valid unicode. There is another call `env::vars_os` will return OsString pairs instead, and wont panic. but you have to deal with OsString instead of String.

Arguments

You can also use the `std::env` module to get at the calling arguments.

```
main() {
    for arg in env::args() {
        println!("{}",argument);
    }
    /// should be able to do this as well
    let args = env::args().collect::<Vec<_>>();
    println!("args {:?}", args);
}
```

That is nice but...

There are a couple of crates which I recommend for dealing with command line arguments. Ok, there is actually one crate:

- clap

Clap usage looks something like this:

```
use clap::{Arg, App, SubCommand};

fn main() {
    let matches = App::new("My Super Program")
                          .version("1.0")
                          .author("Kevin K. <kbknapp@gmail.com>")
                          .about("Does awesome things")
                          .arg(Arg::with_name("config")
                               .short("c")
                               .long("config")
                               .value_name("FILE")
                               .help("Sets a custom config file")
                               .takes_value(true))
                          .arg(Arg::with_name("INPUT")
                               .help("Sets the input file to use")
                               .required(true)
                               .index(1))
                          .arg(Arg::with_name("v")
```

```
                                    .short("v")
                                    .multiple(true)
                                    .help("Sets the level of verbosity"))
                            .subcommand(SubCommand::with_name("test")
                                        .about("controls testing features")
                                        .version("1.3")
                                        .author("Someone E.
<someone_else@other.com>")
                                        .arg(Arg::with_name("debug")
                                            .short("d")
                                            .help("print debug information
verbosely")))
                            .get_matches();

    // Gets a value for config if supplied by user, or defaults to
"default.conf"
    let config = matches.value_of("config").unwrap_or("default.conf");
    println!("Value for config: {}", config);

    // Calling .unwrap() is safe here because "INPUT" is required (if
"INPUT" wasn't
    // required we could have used an 'if let' to conditionally get the
value)
    println!("Using input file: {}", matches.value_of("INPUT").unwrap());

    // Vary the output based on how many times the user used the "verbose"
flag
    // (i.e. 'myprog -v -v -v' or 'myprog -vvv' vs 'myprog -v'
    match matches.occurrences_of("v") {
        0 => println!("No verbose info"),
        1 => println!("Some verbose info"),
        2 => println!("Tons of verbose info"),
        3 | _ => println!("Don't be crazy"),
    }

    // You can handle information about subcommands by requesting their
matches by name
    // (as below), requesting just the name used, or both at the same time
    if let Some(matches) = matches.subcommand_matches("test") {
        if matches.is_present("debug") {
            println!("Printing debug info...");
        } else {
            println!("Printing normally...");
        }
    }

    // more program logic goes here...
}
```

Clap actually has 4 ways of defining a cli: 1 - in code 2 - as a yaml file 3 - using the clap_app! macro 4 - the best way... using structopt

structopt

structopt

Here is an example...

```
use std::path::PathBuf;
use structopt::StructOpt;

#[derive(Debug, StructOpt)]
#[structopt(name = "example", about = "An example of StructOpt usage.")]
struct Opt {
    /// Activate debug mode
    // short and long flags (-d, --debug) will be deduced from the field's
name
    #[structopt(short, long)]
    debug: bool,

    /// Set speed
    // we don't want to name it "speed", need to look smart
    #[structopt(short = "v", long = "velocity", default_value = "42")]
    speed: f64,

    /// Input file
    #[structopt(parse(from_os_str))]
    input: PathBuf,

    /// Output file, stdout if not present
    #[structopt(parse(from_os_str))]
    output: Option<PathBuf>,

    /// Where to write the output: to `stdout` or `file`
    #[structopt(short)]
    out_type: String,

    /// File name: only required when `out` is set to `file`
    #[structopt(name = "FILE", required_if("out_type", "file"))]
    file_name: Option<String>,
}

fn main() {
    let opt = Opt::from_args();
    println!("{:?}", opt);
}
```

Executable Error Handling...

For those times when you are more interested in reporting errors than handling them, there is a great error crate called anyhow

Anyhow will handle any error that implements the std::error::Error trait.

```
use anyhow::{Context, Result, anyhow};

fn get_info() -> Result<String> {
   let config = std::fs::read_to_string("this_doesnt_exist.json")?;
   let conf2 = std::fs::read(path).with_context(|| format!("Failed to read
from {}",path))?;
   if config.chars().count() < 5 {
       return Err(anyhow!("config is arbitrarily too short for my
tastes"));
   }
   Ok(config)
}
```

If you absolutely need to, you can downcast. (although if you care about the types, maybe you should be using a custom error type like thiserror.)

```
match results.downcast_ref::<DataStoreError>() {
    Some(DataStoreError::Censored(_)) => Do something interesting
    None => do other thing
}
```

file system module

The std::fs module has a bunch of useful functions

cannonicalize a path

```
pub fn canonicalize<P: AsRef<Path>>(path: P) -> Result<PathBuf>
```

Example

```
use std::fs;

fn main() -> std::io::Result<()> {
    let path = fs::canonicalize("../a/../foo.txt")?;
    Ok(())
}
```

creating directory

```
pub fn create_dir<P: AsRef<Path>>(path: P) -> Result<()>
```

(corresponds to mkdir on unix and CreateDirectory on windows)

Example

```
use std::fs;

fn main() -> std::io::Result<()> {
    fs::create_dir("./some/dir")?;
    Ok(())
}
```

create directories

```
pub fn create_dir_all<P: AsRef<Path>>(path: P) -> Result<()>
```

Example

```
use std::fs;

fn main() -> std::io::Result<()> {
    fs::create_dir_all("/some/dir")?;
    Ok(())
}
```

Reading files

Read the entire contents of a file into a string.

This is a convenience function for using File::open and read_to_string with fewer imports and without an intermediate variable. It pre-allocates a buffer based on the file size when available, so it is generally faster than reading into a string created with String::new().

```
pub fn read_to_string<P: AsRef<Path>>(path: P) -> Result<String> // that
is a std::io::Result
```

Example

```
use std::io;

fn get_string() -> io::Result<String> {
    let mut buffer = String::new();
```

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

    Ok(buffer)
}
```

Writing data to a file

```
pub fn write<P: AsRef<Path>, C: AsRef<[u8]>>(path: P, contents: C) ->
Result<()>
```

```
use std::fs;

fn main() -> std::io::Result<()> {
    fs::write("foo.txt", b"Lorem ipsum")?;
    fs::write("bar.txt", "dolor sit")?;
    Ok(())
}
```