Misc: Syntax, Crates, `std`

CIS 198 Lecture 7

`const`

```
const PI: f32 = 3.1419;
```

- Defines constants that live for the duration of the program.
- Must annotate the type!
- Constants "live" for the duration of the program.
  - Think of them as being inlined every time they're used.
  - No guarantee that multiple references to the same constant are the same.

`static`

```
static PI: f32 = 3.1419;
```

- As above: must annotate type.
- Typical global variable with fixed memory address.
- All references to static variables has the `'static` lifetime, because statics live as long as the program.
- `unsafe` to mutate.

```
let life_of_pi: &'static f32 = &PI;
```

- String literals are references (with lifetime `'static`) to `static` `str`s.

`static`

```rust
static mut counter: i32 = 0;
```

- You can create mutable static variables, but you can only mutate them inside `unsafe` blocks.
  - Rust forces you to declare when you're doing things that are... ~~morally questionable~~ potentially going to crash your program.

Modules & Crates

Modules

- We've seen these in the homework, but not talked about them.
- Everything in Rust is module-scoped: if it's not pub, it's only accessible from within the same module.
- Modules can be defined within one file:

```
mod english {
    pub mod greetings {
    }
    pub mod farewells {
    }
}

mod japanese {
    pub mod greetings {
    }
    pub mod farewells {
    }
}
```


Reference: TRPL 4.25

Modules

```rust
mod english {
    pub mod greetings { /* ... */ }
}
```

- Modules can be defined as files instead:
- `lib.rs`:

  ```rust
  mod english;
  ```

- `english.rs`:

  ```rust
  pub mod greetings { /* ... */ }
  ```

Modules

```
mod english {
    pub mod greetings { /* ... */ }
}
```

- Modules can also be defined as directories:
- `lib.rs`:

  ```
  mod english;
  ```

- `english/`
  - `mod.rs`:

    ```
    pub mod greetings;
    ```

  - `greetings.rs`:

    ```
    /* ... */
    ```

Namespacing

- When accessing a member of a module, by default, namespaces are relative to the current module:

```
mod one {
    mod two { pub fn foo() {} }
    fn bar() {
        two::foo()
    }
}
```

- But it can be made absolute with a leading `::` operator:

```
mod one {
    mod two { pub fn foo() {} }
    fn bar() {
        ::one::two::foo()
    }
}
```

useing Modules

- use has the opposite rules.
- use directives are absolute by default:

```
use english::greetings;
```

- But can be relative to the current module:

```
// english/mod.rs
use self::greetings;
use super::japanese;
```

- pub use can be used to re-export other items:

```
// default_language.rs

#[cfg(english)]
pub use english::*;

#[cfg(japanese)]
pub use japanese::*;
```

Using External Crates

- For external crates, use `extern crate` instead of `mod`.

```
extern crate rand;

use rand::Rng;
```

Making Your Own Crate

- We've been writing lib crates - but how do we export from them?
- Anything marked `pub` in the root module (`lib.rs`) is exported:

```
pub mod english;
```

- Easy!

Using Your Own Crate

- Now, you can use your own crate from Cargo:

```
[dependencies]
myfoo = { git = "https://github.com/me/foo-rs" }
mybar = { path = "../rust-bar" }
```

- Or:

```
[dependencies.myfoo]
git = "https://github.com/me/foo-rs"
```

- And use them:

```
extern crate myfoo;

use myfoo::english;
```

Cargo: you got your bins in my lib

- We've seen both lib and bin (executable) crates in homework
  - Executable-only crates don't export any importable crates.
  - But this isn't *really* a distinction!
- Cargo allows *both* `:/src/lib.rs` and `:/src/main.rs`.
  - Cargo will also build `:/src/bin/*.rs` as executables.
- Examples go in `:/examples/*.rs`.
  - Built by `cargo test` (to ensure examples always build).
  - Can be called with `cargo run --example foo`.
- Integration (non-unit) tests go in `:/tests/*.rs`.
- Benchmarks go in `:/benches/*.rs`.

Cargo: Features

- Features of a crate can be toggled at build time:
  - `cargo build --features using-html9`

```toml
[package]
name = "myfacebumblr"

[features]
# Enable default dependencies: require web-vortal *feature*
default = ["web-vortal"]

# Extra feature; now we can use #[cfg(feature = "web-vortal")]
web-vortal = []

# Also require h9rbs-js *crate* with its commodore64 feature.
using-html9 = ["h9rbs-js/commodore64"]

[dependencies]
# Optional dependency can be enabled by either:
# (a) feature dependencies or (b) extern crate h9rbs_js.
h9rbs-js = { optional = "true" }
```

Cargo: Build Scripts

- Sometimes, you need more than what Cargo can provide.
- For this, we have build scripts!
  - Of course, they're written in Rust.

```
[package]
build = "build.rs"
```

- Now, `cargo build` will compile and run `:/build.rs` first.

Cargo: The Rabbit Hole

- Cargo has a lot of features. If you're interested, check them out in the [Cargo manifest format][] documentation.

Attributes

- Ways to pass information to the compiler.
- `#[test]` is an attribute that annotates a function as a test.
- `#[test]` annotates the next block; `#![test]` annotates the surrounding block.

```
#[test]
fn midterm1() {
    // ...
}

fn midterm2() {
    #![test]
    // ...
}
```

Attributes

- Use attributes to...
  - `#![no_std]` disable the standard library.
  - `#[derive(Debug)]` auto-derive traits.
  - `#[inline(always)]` give compiler behavior hints.
  - `#[allow(missing_docs)]` disable compiler warnings for certain lints.
  - `#![crate_type = "lib"]` provide crate metadata.
  - `#![feature(box_syntax)]` enable unstable syntax.
  - `#[cfg(target_os = "linux")]` define conditional compilation.
  - And many more!

Rust Code Style

Rust Code Style

- A style guide is being *drafted* as part of the Rust docs.
- The main reason for many of the rules is to prevent pointless arguments about things like spaces and braces.
    - If you contribute to an open-source Rust project, it will probably be expected that you follow these rules.
- The rustfmt project is an automatic code formatter.

Spaces

- Lines must not exceed 99 characters.
- Use 4 spaces for indentation, not tabs.
- No trailing whitespace at the end of lines or files.
- Use spaces around binary operators: `x + y`.
- Put spaces after, but not before, commas and colons: `x: i32`.
- When line-wrapping function parameters, they should align.

```
fn frobnicate(a: Bar, b: Bar,
              c: Bar, d: Bar)
              -> Bar {
}
```

Braces

- Opening braces always go on the same line.
- Match arms get braces, except for single-line expressions.
- `return` statements get semicolons.
- Trailing commas (in structs, matches, etc.) should be included if the closing delimiter is on a separate line.

Capitalization & Naming

- You may have seen built-in lints on how to spell identifiers.
  - `CamelCase`: types, traits.
  - `lowerCamelCase`: not used.
  - `snake_case`: crates, modules, functions, methods, variables.
  - `SCREAMING_SNAKE_CASE`: static variables and constants.
  - `T` (single capital letter): type parameters.
  - `'a` (tick + short lowercase name): lifetime parameters.
- Constructors and conversions should be worded:
  - `new`, `new_with_stuff`: constructors.
  - `from_foo`: conversion constructors.
  - `as_foo`: free non-consuming conversion.
  - `to_foo`: expensive non-consuming conversion.
  - `into_foo`: consuming conversion.

Advanced `format!`ing

- The `?` means debug-print. But what goes before the `:` part?
  - A *positional parameter*! An index into the argument list.

```
println!("{2} {} {} {0} {} {}", 0, 1, 2, 3) // ==> "2 0 1 0 2 3"
```

- Among the specifiers with no positional parameter, they implicitly count up: `{0} {1} {2} ....`
- There are also *named parameters*:

```
format!("{name} {}", 1, name = 2); // ==> "2 1"
```

`format!` Specifiers

- We've been printing stuff out with `println!("{:?}", bst);`
- There are more format specifiers than just `{}` and `{:?}`.
  - These all call traits in `std::fmt`:

Spec.	Trait	Spec.	Trait	Spec.	Trait
`{}`	Display	`{:?}`	Debug	`{:o}`	Octal
`{:x}`	LowerHex	`{:X}`	UpperHex	`{:p}`	Pointer
`{:b}`	Binary	`{:e}`	LowerExp	`{:E}`	UpperExp

`format!` Specifiers

- There are tons of options for each of these format specifiers.
- Examples:
  - `{:04}` -> `0010`: padding
  - `'{:^4}'` -> `' 10 '`: alignment (centering)
  - `#` indicates an "alternate" print format:
  - `{:#X}` -> `0xA`: including `0x`
  - `{:#?}`: Pretty-prints objects:

```
A {
    x: 5,
    b: B {
        y: 4
    }
}
```

- Complete reference: [std::fmt]

Operators

- Operators are evaluated left-to-right, in the following order:
  - Unary operators: `!` `-` `*` `&` `&mut`
  - `as` casting
  - `*` `/` `%` multiplicative arithmetic
  - `+` `-` additive arithmetic
  - `<<` `>>` shift arithmetic
  - `&` bitwise and
  - `^` bitwise xor
  - `|` bitwise or
  - `==` `!=` `<` `>` `<=` `>=` logical comparison
  - `&&` logical and
  - `||` logical or
  - `=` `..` assignment and ranges
- Also: `call()`, `index[]`

Operator Overloading

- Okay, same old, same old. We can customize these!
- Rust defines these - surprise! - using traits, in `std::ops`.
  - `Neg`, `Not`, `Deref`, `DerefMut`
  - `Mul`, `Div`, `Mod`
  - `Add`, `Sub`
  - `Shl`, `Shr`
  - `BitAnd`
  - `BitXor`
  - `BitOr`
  - `Eq`, `PartialEq`, `Ord`, `PartialOrd`
  - `And`
  - `Or`
- Also: `Fn`, `FnMut`, `FnOnce`, `Index`, `IndexMut`, `Drop`

`From` One Type `Into` Another

- Casting (`as`) cannot be overloaded - instead, we use `From` and `Into`.
  - `trait From<T> { fn from(T) -> Self; }`, called like `Y::from(x)`.
  - `trait Into<T> { fn into(self) -> T; }`, called like `x.into()`.
- If you implement `From`, `Into` will be automatically implemented.
  - So you should prefer implementing `From`.

```rust
struct A(Vec<i32>);
impl From<Vec<i32>> for A {
    fn from(v: Vec<i32>) -> Self {
        A(v)
    }
}
```

`From` One Type `Into` Another

- But sometimes, for various reasons, implementing `From` isn't possible - only `Into`.

```rust
struct A(Vec<i32>);

impl From<A> for Vec<i32> { // error: private type A in
    fn from(a: A) -> Self { // exported type signature.
        let A(v) = a; v     // (This impl is exported because
    }                       // both the trait (From) and the type
}                           // (Vec) are visible from outside.)

impl Into<Vec<i32>> for A {
    fn into(self) -> Vec<i32> {
        let A(v) = self; v
    }
}
```

Making References

- `Borrow`/`BorrowMut`: "a trait for borrowing data."[1]

```
trait Borrow<Borrowed> { fn borrow(&self) -> &Borrowed; }
```

- `AsRef`/`AsMut`: "a cheap, reference-to-reference conversion."[2]

```
trait AsRef<T>         { fn as_ref(&self) -> &T; }
```

- So... they're exactly the same?

[1] Trait std::borrow::Borrow

[2] Trait std::convert::AsRef

Making References

- No! While the have the same definition, `Borrow` carries additional connotations:
  - "If you are implementing Borrow and both Self and Borrowed implement Hash, Eq, and/or Ord, they must produce the same result."[1] [2]
- Borrow has a blanket implementation:
  - `impl<T> Borrow<T> for T`: you can always convert `T` to `&T`.
- `AsRef` actually has its own blanket implementation:
  - `impl<'a, T, U> AsRef<U> for &'a T where T: AsRef<U>`
  - For all `T`, if `T` implements `AsRef`, `&T` also implements `AsRef`.
- All this means you usually want to implement `AsRef`.

[1] Trait std::borrow::Borrow

[2] aturon on Borrow vs AsMut