Constructors

There is exactly one way to create an instance of a user-defined type: name
fields at once:

```
struct Foo {
    a: u8,
    b: u32,
    c: bool,
}

enum Bar {
    X(u32),
    Y(bool),
}

struct Unit;

let foo = Foo { a: 0, b: 1, c: false };
let bar = Bar::X(0);
let empty = Unit;
```

That's it. Every other way you make an instance of a type is just calling a tot
does some stuff and eventually bottoms out to The One True Constructor.

Unlike C++, Rust does not come with a slew of built-in kinds of constructor.
Default, Assignment, Move, or whatever constructors. The reasons for this
boils down to Rust's philosophy of *being explicit*.

Move constructors are meaningless in Rust because we don't enable types
location in memory. Every type must be ready for it to be blindly memcopi
memory. This means pure on-the-stack-but- still-movable intrusive linked li
happening in Rust (safely).

Assignment and copy constructors similarly don't exist because move sema
semantics in Rust. At most `x = y` just moves the bits of y into the x variabl
facilities for providing C++'s copy- oriented semantics: `Copy` and `Clone`. Cl
equivalent of a copy constructor, but it's never implicitly invoked. You have
on an element you want to be cloned. Copy is a special case of Clone wher
just "copy the bits". Copy types *are* implicitly cloned whenever they're move
definition of Copy this just means not treating the old copy as uninitialized

While Rust provides a `Default` trait for specifying the moral equivalent of
incredibly rare for this trait to be used. This is because variables aren't imp
basically only useful for generic programming. In concrete contexts, a type
method for any kind of "default" constructor. This has no relation to `new` in
no special meaning. It's just a naming convention.

TODO: talk about "placement new"?