swarm

An object pooling system for Rust, optimized for perfomance.

The pooling system manages object instances of a cutom type, and provides update loops to iterate over them.

In order to create a new swarm pool, you need to define what your `pool object` and `swarm properties` types are going to look like. Your `pool object` must at leas implement the Default, Copy and Clone traits from the standard library. The `swarm properties`, on the other hand, does not depend on any traits. Swarm uses Copy and therfore only accepts Sized properties, this means types such as String and Vec aren't allowed. This is where the tools module comes in handy, it provides a few tools that deal with this. The tools have not been optimized for performance and use, but is there to get you started. There are other libraries specifically designed to deal with sized object types, consider using these instead of the tools module.

Basic swarm setup example

```
extern crate swarm_pool;
use swarm_pool::Swarm;
use swarm_pool::tools::sized_pool::SizedPool16;

// create an object you want to pool
#[derive(Default, Copy, Clone)]
pub struct MyPoolObject {           // Swarm uses Copy and therfore only
accepts Sized properties!
    pub name: &static str,          // This means types such as String and
Vec aren't allowed
    pub value: usize,               // The tools module has a few tools
that deal with this
    pub list: SizedPool16<u8>,      // SizedPool is a sized array that can
hold upto 16 items.
}

// create properties you want to share with pooled objects
pub struct MySwarmProperties;

fn main() {
    let swarm = Swarm::<MyPoolObject, MySwarmProperties>::new(10,
MySwarmProperties);
    assert!(swarm.capacity() == 10);
}
```

The swarm is now ready to be used. First of all we need to Spawn new pool instances. In reality all objects in the pool are allready created and are waiting to be used. This means that all objects (from 0 up to, but not including, the maximum capacity) can be accessed through the fetch() methode. The difference between

spawned and non-spawned pool objects is that spawned object are included in all of the Swarm pools iterator methodes and non-spawned object are not.

Spawning and looping

```
let mut swarm = Swarm::<MyPoolObject, _>::new(10, ());
let spawn1 = swarm.spawn().unwrap();
let spawn2 = swarm.spawn().unwrap();

assert_eq!(swarm.fetch_ref(&spawn1).value, 0);
assert_eq!(swarm.fetch_ref(&spawn2).value, 0);

swarm.for_each(|obj| {
    obj.value = 42;
});

assert_eq!(swarm.fetch_ref(&spawn1).value, 42);
assert_eq!(swarm.fetch_ref(&spawn2).value, 42);
```

The real power of this library is not just looping through a few object instances, it is controlling and cross referencing them. There are 2 powerful methodes that can be used to do so: `Swarm.for_all()` and `Swarm.update()`. Both have their advantages and disadvantages, `for_all` loop is fast (equal to a standard vec for loop) but cannot spawn nor kill pool objects, `update` is easy to use, gives full control, but is slow (less than half the speed).

Cross referencing using for_all & update

```
// change properties to contain references to our spawned pool objects
pub struct MySwarmProperties {
    john: Option<Spawn>,
    cristy: Option<Spawn>,
}

let properties = MySwarmProperties { john: None, cristy: None };

let mut swarm = Swarm::<MyPoolObject, MySwarmProperties>::new(10,
properties);
let s_john = swarm.spawn().unwrap();
let s_cristy = swarm.spawn().unwrap();

swarm.properties.john = Some(s_john.mirror());
swarm.properties.cristy = Some(s_cristy.mirror());

swarm.fetch(&s_john).name = "John";
swarm.fetch(&s_cristy).name = "Cristy";

// using the for_all methode
swarm.for_all(|target, list, props| {
```

```
    // john tells critsy to have a value of 2
    if list[*target].name == "John" {
        if let Some(cristy) = &props.cristy {
            list[cristy.pos()].value = 2;
        }
    }
    // cristy tells john to have a value of 1
    if list[*target].name == "Cristy" {
        if let Some(john) = &props.john {
            list[john.pos()].value = 1;
        }
    }
});

assert_eq!(swarm.fetch_ref(&s_john).value, 1);
assert_eq!(swarm.fetch_ref(&s_cristy).value, 2);

// using the update methode
swarm.update(|ctl| {
    let name = ctl.target().name;
    let cristy = ctl.properties.cristy.as_ref().unwrap().mirror();
    let john = ctl.properties.john.as_ref().unwrap().mirror();

    // john tells critsy to have a value of 4
    if name == "John" {
        ctl.fetch(&cristy).value = 4;
    }
    // cristy tells john to have a value of 5
    if name == "Cristy" {
        ctl.fetch(&john).value = 5;
    }
});

assert_eq!(swarm.fetch_ref(&s_john).value, 5);
assert_eq!(swarm.fetch_ref(&s_cristy).value, 4);
```

There are many more functionalities included in the Swarm and SwarmControl types. The documentation on the examples above or other functionalities this library provides are more in depth and should be read, for writing them out was a lot of work 😉