Drop order difficulties

Drop

The term `drop` refers to what is often called `destructor` in OOP languages. It is a piece of code that runs at most once exactly before the object is destroyed. In Rust, a user-provided `drop` code can be provided by implementing the `Drop` trait for your type:

```
pub trait Drop {
    fn drop(&mut self);
}
```

Structures in Rust are dropped recursively, in this order:

1. The `Drop` implementation (if any) is run for the outer type
2. Each field of the struct is recursively dropped starting from the 1. point
3. The object itself is destroyed by the language implementation

It is obvious that the only way to influence this process is by implementing the `Drop` trait.

Partial dropping

In the case of the `Surface<Window>` structure, it was needed to allow the user to retrieve the owned `Window` generic parameter if the user desired to work with the window after the library was done with it. We would like a method that looks like `pub fn drop_without_window(self) -> Window;`. The structure owns the window because we need to uphold an invariant that the window always outlives the surface.

First attempt

At first, the structure looked like:

```
pub struct Surface<Window> {
    window: Window,
    loader: ash::extensions::khr::Surface,
    surface: ash::vk::SurfaceKHR
}
impl<W> Surface<W> {
    pub fn drop_without_window(self) -> Window {
        self.window // this will not compile because moving window out of self is a compiler error

        // here the `self` parameter would be implicitly dropped, but it is now
        // in an inconsistent state that the compiler cannot reason about because
        // of non-trivial drop implementation
    }
}
impl<W> Drop for Surface<W> {
    fn drop(&mut self) {
        unsafe {
            self.loader.destroy_surface(self.surface, None);
        }
    }
}
```

However, manually implementing `Drop` for a type prevents destructuring of that type. That is the implementation of `drop_without_window` will not compile because that would prevent the `Drop` implementation of surface from running. This is a problem because this prevents us from extracting the `window` field out of `surface`, simply because we cannot prove to the compiler in a **safe** manner that the drop code does not depend on `window` field being valid and not moved out of.

Second attempt

Second solution was to move the fields with custom `drop` code into a separate inner struct:

```
struct InnerSurface {
    loader: ash::extensions::khr::Surface,
    surface: ash::vk::SurfaceKHR
}
impl Drop for InnerSurface {
    fn drop(&mut self) {
        unsafe {
            self.loader.destroy_surface(self.surface, None);
        }
    }
}
pub struct Surface<Window> {
    // wrong order of fields, window will be dropped before inner
    window: Window,
    inner: InnerSurface
}
impl<W> Surface<W> {
    pub fn drop_without_window(self) -> Window {
        self.window

        // Implicitly drop self.inner at the end of this scope, window is returned
    }
}
```

This code compiles and works. However, the `unsafe` block in the `drop` implemnetation has non-trivial invariants that need to be uphelp and they are not: `window` needs to outlive `inner`.

Drop order of fields in structs in Rust is defined to be in the order of declaration, which is sometimes non-obivous and needs good documentation so that nobody accidentally moves struct fields around. The example above declares `window` field before `inner` field which would result in the wrong drop order and cause problems.

This solution is viable, however, relying on the drop order in Rust has been slightly controversial, as it clashes with the notion that the declaration order of fields in struct does not imply their memory layout. Indeed, while drop order has been stabilized in RFC-1857, it is still recommended for clarity to use `std::mem::ManuallyDrop` when something non-trivial is happening with drop.

Third attempt

Third and final attempt was inspired by the recommendation in the documentation of `ManuallyDrop`. The code looks like this:

```
struct Surface<Window> {
    window: Option<Window>,
    loader: ash::extensions::khr::Surface,
    surface: ash::vk::SurfaceKHR
}
impl<W> Surface<W> {
    pub fn drop_without_window(mut self) -> Window {
        self.window.take().unwrap()
        // This will never panic as there is no way to create instance of Surface
        // without window set as Some. However, if you somehow do manage to create
        // such instance *without undefined behavior*, no undefined behavior will occur.
        // The compiler should also be able to reason that the value of window will
        // never be `None` and optimize the branch out.

        // here self is still in valid state and is implicitly dropped in full
    }
}
impl<W> Drop for Surface<W> {
    fn drop(&mut self) {
        unsafe {
            self.loader.destroy_surface(self.surface, None);
        }
    }
}
```

This code upholds all invariants, does not require additional `unsafe` code and makes it obvious that `window` is not a normal field but something with special logic. In the end, this code is not only safest of all the alternatives, but also the easiest to implement.

Drop ManuallyDrop

In C++

For comparison, this problem in C++ be much harder to solve correctly. Consider the following two programs:

C++

```
int main() {
    std::cout << ">> Moving out" << std::endl;
    {
        Window original(1);
        std::cout << "original " << original.a << std::endl << std::endl;

        Surface<Window> surface(std::move(original));
        std::cout << "original " << original.a << std::endl;
        std::cout << "in surface " << surface.window.a << std::endl << std::endl;

        Window moved = surface.destroy_without_window();

        std::cout << "original " << original.a << std::endl;
        std::cout << "in surface " << surface.window.a << std::endl;
        std::cout << "moved out " << moved.a << std::endl << std::endl;
    }

    std::cout << std::endl << ">> Not moving out" << std::endl;
    {
        Window original(2);
        std::cout << "original " << original.a << std::endl << std::endl;

        Surface<Window> surface(std::move(original));
        std::cout << "original " << original.a << std::endl;
        std::cout << "in surface " << surface.window.a << std::endl << std::endl;
    }
}
```

Rust

```
fn main() {
    println!(">> Moving out");
    {
        let original = Window::new(1);
        println!("original {:?}\n", original);
        let surface = Surface::new(original);
        // println!("original {:?}", original); // Compiler error, original was moved
        println!("in surface {:?}\n", surface.window);

        let moved = surface.destroy_without_window();
        // println!("original {:?}", original); // Compiler error, original was moved
        // println!("in surface {:?}", surface.window); // Compiler error, surface was moved
        println!("moved out {:?}\n", moved);
    }

    println!("\n>> Not moving out");
    {
        let original = Window::new(2);
        println!("original {:?}\n", original);

        let surface = Surface::new(original);
        println!("in surface {:?}\n", surface.window);
    }
}
```

Note: Full implementation of both these programs is available in the appendix.

In the Rust version, the compiler provides move semantics, protects us from ever using a value that was moved and the program behaves as expected. The `surface` is destroyed exacly when `destroy_without_window` is called.

In contrast, the C++ version requires us to implement explicit move constructor. The moved value is nothing but an instance of the original class with some marker value inside that tells us not to really destroy it in the destructor, but it still runs destructors for all the moved instances. Additionally, the surface is not destroyed in the line that cleary says `destroy_without_window`, it is destroyed when it goes out of scope at the end of the block. All of this places great strain on the programmer, who is much more error prone, instead of the compiler.