Conccurent programing, lifetime and ownership

Introduction to Conccurent programing

Fist of all, we need to understand what is conccurent programing and what are the main challenge of it. Conccurent programing is about splitting tasks and working on them in parrallel,

To understand this, we will compare it to a person responsible to create decoration for a Christmas tree.

A classic program would be, a person working first on cutting the paper for the décoration, then fold it and hang on the tree. Rince and repeat. In conccurent programing, we would split the work. For example split it in four, so asking people 3 other people to help you. You each work on your decoration, then hang on the tree. SImple, no ?

Sadly, you only have one scissor, and only one of you can hang decoration on tree at a time. The scissor would be a shared ressource, and hanging decoration a tree a critical section.

Conccurent program always tried to address those issue. Mupltiple people hanging decoration on the tree could be dangerous, while there is only one scissor so the other have to wait on it.

A simple way to resolve this issue is using what we call a mutex. Mutex is like a lock on something, allowing only one person at a time working on it.

The concept of ownership

Now, we know what is an shared ressource. While the usage of a mutex it can be a bit heavy. In the end you could simple borrow the scissor then give it back when your finished.

Ownership is a programing concept introduced Each variable has an owner. This owner is responsible for the variable its lifetime and the memory management. This would mean we can have a variable, scissor that we can share.

There is mupltple way to share a variable in Rust. The simple way is to move it.

```
let owner_1  = "scissor";
let owner_2 = owner;
```

A simple assigment is enough to generate a move. Moving means changing the owner. The scissor are now under the responsibity of owner_2. Because it was moved, owner_1 cannot use it again. Memory wise, owner_1 was responsible for the block of memory containtg the scissor. Instead of moving it, we can instead lent it. In that case:

```
let owner_1  = "scissor";
let owner_2 = &owner;
```

Here owner_2 asked for a reference, which imply a borrow instead of a move. Memory-wise it would mean that the owner_2 has an access to the memory block containg “scissor”

This borrow can be mutable or not, meanig owner_2 is allowed to modify it or not.

```
let mut owner_1  = "scissor";
let owner_2 = &owner;
//Is allowed because we said it's a mutable borrow
owner_2.append("s")
```

```
let mut owner_1  = "scissor";
let owner_2 = mut & owner;
//Raise a compilation error because it is not a mutable borrow
owner_2.append("s")
```

Now, we know how borrow works. Remember that the owner is responsible for the memory, meaning also dropping (freeing) it.

The concept of lifetime

Now

Memory management

I have a dream. A programming language where I don't have to manage memory. Most programmers out there knows that there is an easy solution. The garbarge collector. Sadly, garbarge collector presents quiet a lot of issues. First of all, garbarge collecting is a complex action. You need to know when there is no more piece of code that is using a block of memory. Because of this complexity, garbarge collecting is a heavy process, unsuitable for light appliction or appliction running in an environnement with low memory availabe.

Now then, as I said, I don't want to manually manage memory. It is prone to error, memory leak and responsible for a lot of break or insecure code. I want to avoid this risk.

Luckily for me, there is a solution. But to explains how it works, we first need to introduce a few concept. Lifetime and ownership.

DME, Delightful Markdown Experience