The Crust of Rust – Move Semantics, Initialization, and the Builder Pattern

Varoon Pazhyanur

Bloomberg L.P.

13 November 2020

In CPP the Rule of 5 (or 0,3,6,7, etc. depending on whom and when one asks) is the lifeblood of RAII and how CPP relates objects and memory. The CPP Core guidelines dedicate many subsequent clarifications. For example, one should prefer the compiler-generated defaults unless there is a good reason to do otherwise (C.20), IN particular, the destructor (C.37) and move operations (C.66) should be noexcept. Every class X must have the following:

```
class X{
    X(const X&)=default;
    operator=(const X&)=default;
    X(X&&)=default;
    X& operator=(X&&)=default;
    ~X()=default;
}
```

The Rust analog of the destructor is the Drop trait, which we ignore for now. The two ampersands denote an r value reference, a language construct introduced in CPP11 to tag a reference as being temporary in some sense. The move constructor and move assignment operators leave the input in a valid but unspecified state, meaning that all class invariants must be preserved. In particular, move semantics typically involves moving identities (pointers and references) or scopes, rather than values (I try to clear my data structures explicitly after moving out of them). Copy constructing a set, for example, might invoke a copy for every node. Move constructing a set, by contrast, may only involve calling std::swap on nullptr and a pointer to the root node of a tree (plus some minor clean up to preserve set invariants).

Motivating Examples

Our code frequently chains operations on data structures especially with unary and binary functions, so we can focus on those. E.x. Pandas dataframes, matrices/ndarrays, sets, maps. While copying an integer may be ok, needlessly copying entire matrices or dataframes can kill performance. CPP and Rust in particular have relatively thorough and high level semantics for explicitly controlling when to copy and when to "steal" an data structure's contents. Rust's ownership sustem helps to avoid common bugs and adhere to many of the CPP Core Guidelines.

1. Consider the operation of merging/unioning two sets, both ordered or unordered. As of CPP17, one can merge a set into another set by copying or moving. text The merge method on std::map is similar.
2. One can move to push back into a vector text or insert many elements into a vector text . See overload 2.
3. std::move is a static cast to an r value reference and can frequently be used to force move semantics.
4. Lambdas can use move semantics as of CPP14 with the help of std::move.

Move Semantics in Rust

In Rust, function parameters move by default, mimicing passing by r value reference from CPP. While it was a CPP convention to never access a variable after it has been moved out of, the borrow checker turns using an identity after it has been moved a compile time error. This permits safe, frequent move semantics. See, for example,

1. hyper::server::Builder v0.13.9 text
2. reqwest::RequestBuilder v0.10.9 text
3. amethyst::GameDataBuilder v0.15.3 https://docs.amethyst.rs/stable/amethyst/struct.GameDataBuilder.html
4. rayon::ThreadPoolBuilder v1.5.0 text
5. arrow::csv::reader::ReaderBuilder v2.0.0 https://docs.rs/arrow/2.0.0/arrow/csv/reader/struct.ReaderBuilder.html
6. clap::App v2.33.3 text
7. mysql::OptsBuilder v20.1.0 text

Other important uses of move semantics in Rust include the folliwng.

1. The primary conversion traits std::convert::From text and std::convert::TryFrom text.