Rust: Ownership, Borrowing (References), Lifetimes, Move/Copy/Clone semantics

Why care about these concepts?

- Reliability: These concepts allows the Rust compiler to prevent common resource errors including memory leaks, dangling pointers, double frees or accessing uninitialized memory.
- Convenience: The Rust compiler can automatically free resources with the help of these concepts.
- Performance: The Rust compiler can free resources without a garbage collector which is great for performance and necessary for real-time systems.

How to use these features?

- These concepts are not about actively doing something it’s mostly about checks the compiler enforces during compilation.
- You will need to understand these concepts to solve the error messages the compiler will throw at you.

Basic memory management terminology

- A variable is just a name for some memory location that holds a value of some type.
- Memory is allocated on the stack, heap or some static region.
- **Stack memory** is automatically allocated when a function is called and freed when the function returns
- **Heap memory** works by calling the memory allocator to allocate or free memory.
- **Static memory** lives the whole lifetime of the program.

Basic memory management terminology

- A **dangling pointer** is a pointer to uninitalized or freed memory.
- A **memory leak** is forgetting to call free on memory allocated on the heap.
- **Uninitialized memory** is about forgetting to allocate memory before using it.
- **Double frees** are errors occurring when calling free on memory twice either on the same variable or come copy of it.

What can go wrong on the stack?

- Since memory is automatically allocated and freed there are no memory leaks, uninitialized memory or double free problems.
- The function might return a pointer to a value on the stack leading to a dangling pointer
- Rust prevents this by simply checking that no such references are returned see *stack-dangling-pointer.rs*
- The error will be something like: error [E0515]: cannot return reference to local variable `number`
- Don’t return references to local function variables - copy or move the value out of the function

What can go wrong on the heap?

- A reference might be used after the memory was reallocated or freed leading to a dangling pointer.
- **The borrow checker** prevents the reallocation by not allowing a mutable and other reference at the same time. An immutable reference is needed to push on a vector and possibly reallocate it see heap-reallocation-dangling-pointer.rs
- The error will be something like error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
- Don’t do it and if you really need a mutable reference paired with other references use the std::cell module

What can go wrong on the heap?

- Rust prevents the use after free case by making sure no reference is used after its lifetime has ended that is the value was dropped see heap-dropped-dangling-pointer.rs
- The error message will be something like error[E0597]: `vec` does not live long enough
- Don’t do it or if you need really run into this problem you might need what is called shared ownership with the std::rc module.

What can go wrong on the heap?

- The borrow checker also does not allow a value to be moved to another variable that could reallocate or free the memory while there are references see heap-move-dangling-pointer.rs
- The error message will be something like error[E0505]: cannot move out of `vec` because it is borrowed
- Don’t move a value to another variable and then use a reference to it you created before

What is Ownership?

- Rust does automatically free memory so there are no memory leaks or double free calls
- Rust does this without a garbage collector
- The concept to do it safely is simple: Call a destructor whenever the lifetime of a value ends which is the end of the current scope {}
- The only problem is that certain values like a vector might be copied which would lead to a double free. With Copy is meant what people call shallow copy where you only copy the pointer and some metadata not the actual underlying buffer.

What is ownership?

- Therefore, when you copy a vector, the previous variable can't be used anymore. It's not called a Copy but a Move in this case. There will be no destructor called for the previous variable. See move-semantics.rs
- If you attempt to use a variable after a move the error will be something like error[E0382]: borrow of moved value: `vec
- Don't use a value after it was moved. If you need a deep copy call .clone() which is called a Clone. *This is almost always unnecessary and a mistake*.

What is ownership?

- Certain values are moved by default and others are copied by default.
- Primitive integers are copied by default because there is no potential double free.
- Collections and other types with pointers are typically moved.
- Your own structures are moved by default unless you implement the Copy trait with #[derive(Copy, Clone)] see copy-semantics.rs
- If don't have Copy implemented Rust will tell with ------ move occurs because `config` has type `Config`, which does not implement the `Copy` trait
- Don't implement Copy if you can live with references. You can also just implement Clone and call .clone() manually every time see clone-semantics.rs.

Summary and miscellaneous infos

- Ownership, borrowing and lifetimes are concepts that enable the Rust compiler to detect and prevent memory errors and handle memory for you.
- Safe Rust does guarantee memory safety. You can also use unsafe Rust inside an unsafe {}
- Rust understands when to free memory also in loops, if clauses, iterators, when you partially move out values out of a structure etc.
- If your program compiles you have those memory guarantees without a runtime garbage collector.