Consuming Rust bite by byte

Bite 1 - Data

Jim Fawcett

https://JimFawcett.github.io

Why Rust?

- Memory and Data Race safety
  - Enforced data ownership rules insure Memory and Data Race safety.
- Error Handling
  - Any function that can fail returns a result indicating success or failure. Code has to handle errors in well defined ways.
- Performance
  - Rust compiles to native code and does not need garbage collection, so it is as fast as C and C++.
- Simple Value Behavior
  - Rust supports value behavior without the need to define copy and move constructors and assignment operators.
- Extraordinarily effective tool chain

What is this?

- This is the first in a series of bites - brief presentations - about the Rust programming language:
  - Each presentation will be brief – a few slides
  - Each will focus on one part of the Rust language
  - The series will build in bite sized chunks: easy to grasp, quick to consume.

Bite #1 – Rust Data

- Our goal is to understand the terms:
  - Bind – associate an identifier with a value
  - Copy – bind to a copy of a Copy type
  - Move – transfer ownership of a value
  - Clone – make a clone of a !Copy type

Bite #1 – Binding to a value

- Bind – associate an identifier with a memory location
  - Every identifier has a type:
    - `let k : i32 = 42;`
    - let signifies a binding is being created
    - `i32` is the type of a 32 bit integer
    - 42 is a value placed in the memory location associated with k
  - A type is a set of legal values with associated operations.
  - Type inference:
    - `let k = 42;`
    - This binding is legal and has the same meaning as the previous binding.
    - In lieu of other information, Rust will assign the type `i32` to any unadorned integral value that can be correctly written to a 32 bit location.

Bite #1 – Binding to an identifier

- Binding to an identifier has several forms:
  - `let j:i32 = k;  // makes copy because k is blittable`
  - `let l = &k;     // l makes reference to k, called a borrow`
  - `let s:String = “a string”.into_string();`
  - `let t = s;      // moves s into t, e.g., transfers ownership`
    `                // because s is not blittable`

- Blittable
  - A blittable type occupies a single contiguous block of memory, and so can be correctly copied to a new location with a single memcpy.
  - Non-blittable types occupy more than one memory location, usually one contiguous block on the stack and one or more blocks on the heap.
    - Non-blittable types cannot be successfully copied with a single memcpy operation.

Bite #1 - Ownership

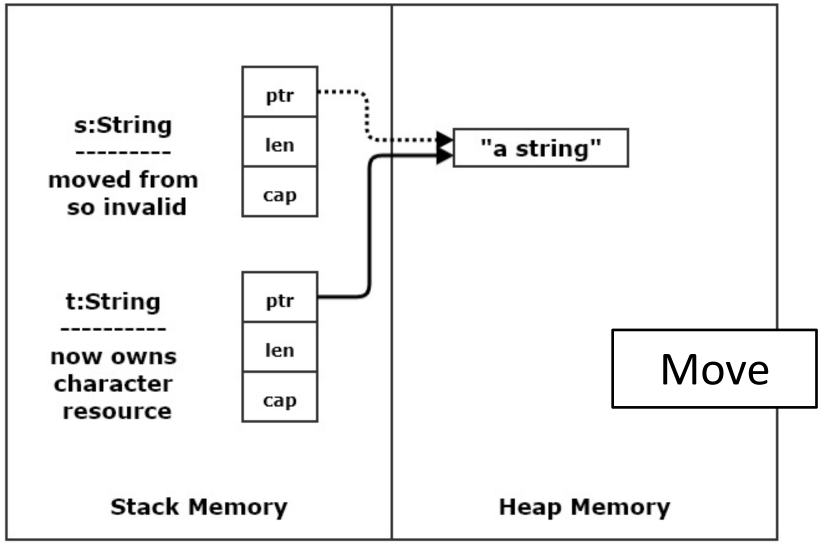
- Ownership in Rust is an interesting concept.
  - In Rust, data has one, and only one owner.
  - Ownership can be borrowed or transferred.
  - There are rules about ownership that we discuss in Bite #3.
  - Following Rust's ownership rules makes Rust code memory-safe.
    - Enforced by rustc, the Rust compiler
  - The rules also make Rust code free from data races
    - Rust will not compile code that is shared between threads unless it is guarded by a lock.
    - That, combined with single-ownership, ensures ordered access to shared data, one thread at a time.

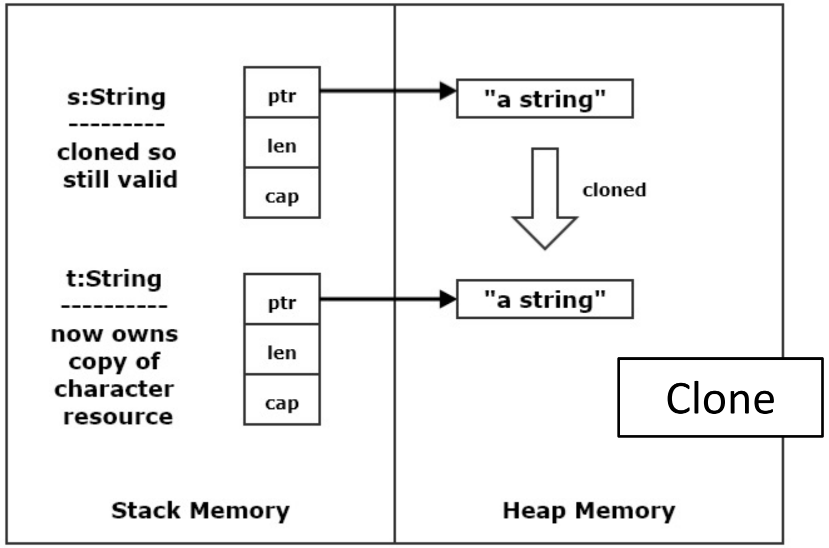
Bite #1 – Copy and Borrow

- A copy operation can occur only for values that satisfy the Copy trait.
  - A trait is, like an interface, a specification of a contract. Copy contract requires Rust code, when binding, to copy data with that trait.
  - To satisfy Copy, the data must be blittable.
  - Copies happen implicitly when an identifier is bound to a Copy type.
    - `let i = 3;  let j = i;  // copy`
- `Borrows - binding references to other identifiers`
  - `A reference is a safe pointer to the bound memory location.`
    - `let r = &i;`

Bite #1 – Move and Clone

- A move transfers a Move type's heap resources to another instance of that type
  - The string, s, shown in the top diagram is moved to t with the statement:
    - `let t = s;`
  - Move transfers ownership of resources.
- A clone copies a Move type's heap resources to a new instance of that type.
  - The string s, shown in the bottom diagram is cloned with the statement:
    - `let t = s.clone();`
  - Clone operation copies resources to target.





Exercises

1. Create an instance of a blittable type and show when it is copied.
    - Can you prove that it was copied?
2. Create an instance of a non-blittable type and show when it is moved.
    - Can you prove that it was moved?
3. Repeat the second exercise but clone the non-blittable type before moving it. Show that the clone is valid while the move source is not valid.

Hint:

- Integral types, chars, and floating-point types are blittable
- Strings, Vecs, VecDeques, and Maps are non-blittable.

References

Link	Description
ConsumingRustBite2 - UDB	Undefined behavior – example from C++ code
Rust Story - Data	Expanded discussion in Rust Story

That’s all until Bite #2

Bite #2 illustrates undefined behavior with C++ code, showing us why we need Rust.