From C++ to Rust

A high level overview about two Systems Programming Languages

Gerald Stanje

November 23, 2015

Content

- Systems Programming
- Memory management in C++
- Static Analysis
- Rust
- Ownership in Rust
- Borrow in Rust
- Concurrency in Rust
- Cargo

System Programming

- Programmer needs very explicit control over the hardware
  - How much memory is used?
  - What code is generated?
  - When the memory is allocated or freed?
- Used to build: operating systems, compilers, device drivers, factory automation, robots, high performance mathematical software, games

Memory management in C++ is hard

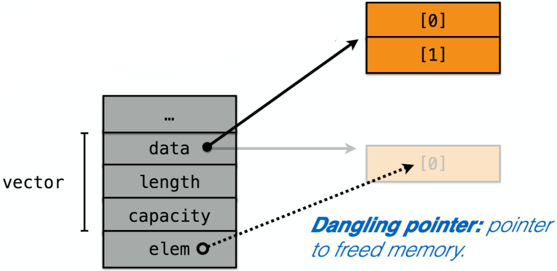
Video

C++ Safety

```
int main() {
    vector<string> vec;
    vec.push_back("FC");
    auto& elem = vec[0];
    vec.push_back("Bayern␣Munich");
    cout << elem << endl;
}
```

C++ Safety

```
int main() {
    vector<string> vec;
    vec.push_back("FC");
    auto& elem = vec[0];
    vec.push_back("Bayern␣Munich");
    cout << elem << endl;
}
```



Dangling pointer: *pointer to freed memory.*

Static Analysis

- Lots of free, open source and commercial offerings for static analysis of C++ source
  - cppcheck
  - clang-analyse
  - coverity
- Analyzers for code guideline profiles
  - bounds
  - types
  - lifetimes
- Downside: False Positives → cannot see if a certain condition can never happen if e.g. it depends on input.

What is Rust?

- systems programming language
- blazingly fast
- compiled binary
- immutable by default
- prevents almost all crashes: no segmentation faults, no null pointers, no dangling pointers
- eliminates data races: two parallel processes access memory location in parallel, at least one process writes to the memory.
- uses LLVM in the backend

Ownership in Rust

- Exactly one owner per allocation
- Memory freed when the owner leaves scope
  - All references must be out of scope too
- Ownership may be transferred (move)
  - Invalidates prior owner

```
fn print(a: Vec<i32>) {}

fn main() {
    let s = Vec::new();
    a.push(1);
    a.push(2);

    print(s);
    print(s); // error: s is no longer the
    // owner of the vector
}
```

Shared Borrow in Rust

- Ownership may be temporary (borrow)
  - References are created with &
- Borrowing prevents moving
- Shared references are immutable

```
fn print(a: &Vec<i32>) {}

fn main() {
    let a = Vec::new();
    a.push(1);
    a.push(2);

    print(&a);
    print(&a);
}
```

Mutable Borrow in Rust

- There can only be one unique reference to a var. that is mutable
- Borrows values are valid for a lifetime

```
fn muliply(vec: &mut Vec<i32>) {
    for e in vec.iter_mut() { *e *= 2; }
}

fn main() {
    let mut vec: Vec<i32> = vec![1, 2];
    {
        let mut vec2 = &mut vec;
        muliply(&mut vec2);
    }
    muliply(&mut vec);
}
```

Concurrency in Rust

- Using ownership to prevent data races
- Parallelism is achieved at the granularity of an OS thread
- Safety is achieved by requiring that a ’move’ owns captured variables

```
fn main() {
    let mut a = Vec::new();
    // ’move’ instructs the closure to move out of
    // its environment
    thread::spawn(move || {
        a.push("foo");
    });
    a.push("bar"); // error: using a moved value
}
```

Concurrency in Rust

- Threads can communicate with channels

```
fn main() {
    let (tx, rx) = mpsc::channel();
    let x = Box::new(5); // allocate 5 on the heap

    thread::spawn(move || {
        let result = 5 + *x;
        tx.send(result);
    });

    let result = rx.recv().unwrap();
    println!("{}", result);
}
```

Cargo

- Package manager, similar to pip in Python
- Download and manage dependencies
- Build the project
  - Create Cargo.toml

    ```
    $ cargo new hello_word --bin
    ```

  - Cargo.toml

    ```
    [package]
    name = "hello_world"
    version = "0.1.0"
    authors = ["Gerald␣Stanje"]
    [dependencies]
    regex = "0.1.33"
    ```

  - Build project

    ```
    $ cargo build --release
    Compiling hello_world v0.1.0 (file:///ho
    me/geri/code/hello_world)
    ```