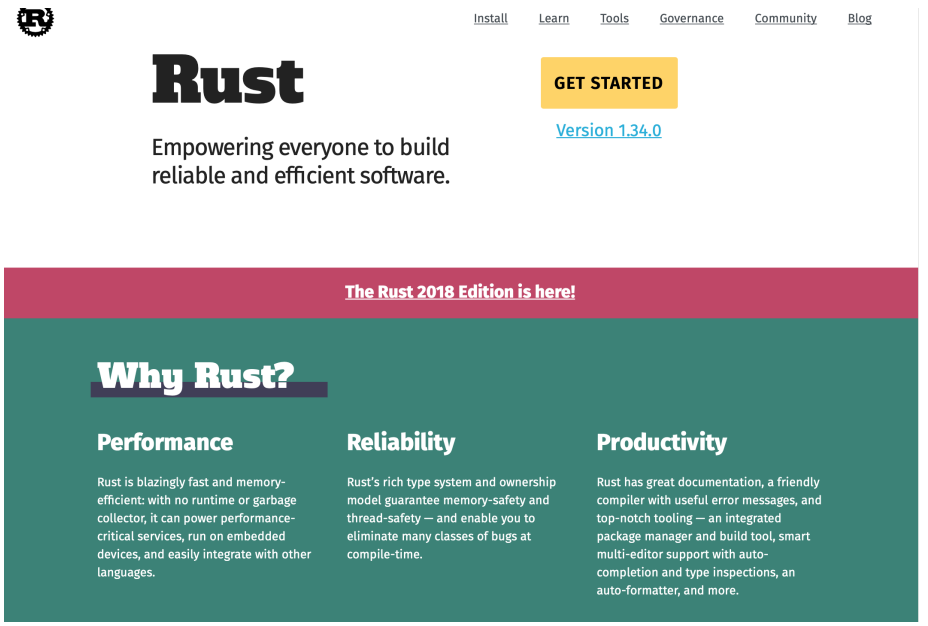
An Overview of the Rust Programming Language

Jim Royer

CIS 352

April 29, 2019

www.rust-lang.org



References

- *The Rust Programming Language* by S. Klabnik and C. Nichols, 2018. `https://doc.rust-lang.org/book/`
- *Rust*, N. Matsakis: `http://www.slideshare.net/nikomatsakis/guaranteeing-memory-safety-in-rust-39042975`
- *Rust: Unlocking Systems Programming*, A. Turon, `http://www.slideshare.net/InfoQ/rust-unlocking-systems-programming`
- *Rust's ownership and move semantics*, T. Ohzeki, `http://www.slideshare.net/saneyuki/rusts-ownership-and-move-semantics`
- *The Rust Programming Language*, A. Crichton, a Google Tech Talk, `https://www.youtube.com/watch?v=d1uraoHM8Gg`
- *CIS 198: Rust Programming, University of Pennsylvania, Spring 2016*, `http://cis198-2016s.github.io`
- *Programming Rust*, by Jim Blandy and Jason Orendorff, O'Reilly Media, 2017.
- `https://insights.stackoverflow.com/survey/2018/#technology-_-most-loved-dreaded-and-wanted-languages`

I shamelessly filch images and entire slides from these folks.

Sample code: Factorial, 1

A recursive version of factorial

```
fn fact_recursive(n: u64) -> u64 {
  match n {
    0 => 1,
    _ => n * fact_recursive(n-1)
  }
}

fn main () {
  for i in 1..10 {
    println!("{}\t{}",
             i, fact_recursive(i));
  }
}
```

- `u64` ≡ the type of unsigned 64bit integers
- "`n: u64`" ≡ `n` is of type `u64`
- "`-> u64`" ≡ the fn returns a `u64`-value
- `match` ≡ Haskell's `case` (with pattern matching)
- `1..10` ≡ an iterator for the numbers 1 through 9 *(Yes, I mean 9.)*

Sample code: Factorial, 2

A iterative version of factorial

```
fn fact_iterative(n: u64) -> u64 {
  let mut i      = 1u64;
  let mut result = 1u64;
  while i<=n {
    result *= i;
    i += 1;
  }
  return result;
}

fn main () {
  for i in 1..10 {
    println!("{}\t{}",
             i, fact_iterative(i));
  }
}
```

- "`let mut i`" ≡ declares a *mutable* variable `i`.
- "`let j`" ≡ declares a *immutable* variable `i`.
- `1u64` ≡ u64-version of 1.
- The compiler figures out the types of `i` and `result`.

Sample code: Factorial, 3

A iterator version of factorial

```
fn fact_iterator(n: u64) -> u64 {
  (1..n+1).fold(1, |p, m| p*m)
}

fn main () {
  for i in 1..10 {
    println!("{}\t{}",
             i, fact_iterator(i));
  }
}
```

- `|p, m| p*m` ≡ $\lambda p, m.(p * m)$
- In fact, `fact_iterator` ≡ `foldl (\p m->p*m) 1 [1..n]`

Back to brag list from www.rust-lang.org, 1

- type inference
  Rust's type system is a cousin of ML's and Haskell's
- pattern matching
  as in ML and Haskell and Swift and . . .
- trait-based generics
  *in place of OO-classes, Rust uses a version of Haskell's type-classes. This is **much** less bureaucratic that the standard OO framework.*

Back to brag list from www.rust-lang.org, 2

- zero-cost abstractions
- efficient C bindings
- minimal runtime
  These are performance goals borrowed from C++.

Stroustrup on C++:

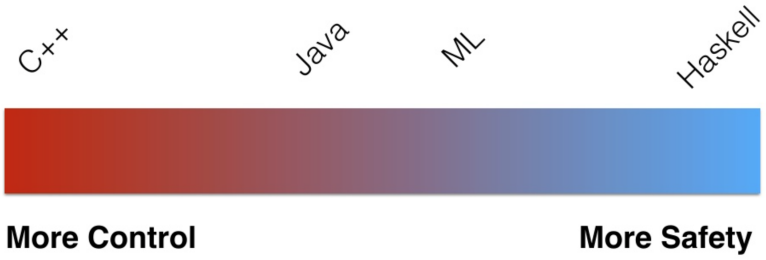
C++ implementations obey the zero-overhead principle:
What you don't use, you don't pay for.
And further:
What you do use, you couldn't hand code any better.

- ➤ The prices Stroustrup is worried about paying are the *time and space* used by running programs.
- ➤ . . . the time and effort binding wounds from repeated shooting yourself in the foot — that is ***not*** Stroustrup's concern.
- ➤ But Rust wants to avoid both sorts of prices.

Back to brag list from www.rust-lang.org, 3

- guaranteed memory safety
- threads without data races

*These are safety **guarantees**. Why this is a big deal.*



Rust: Control *and* safety

C++

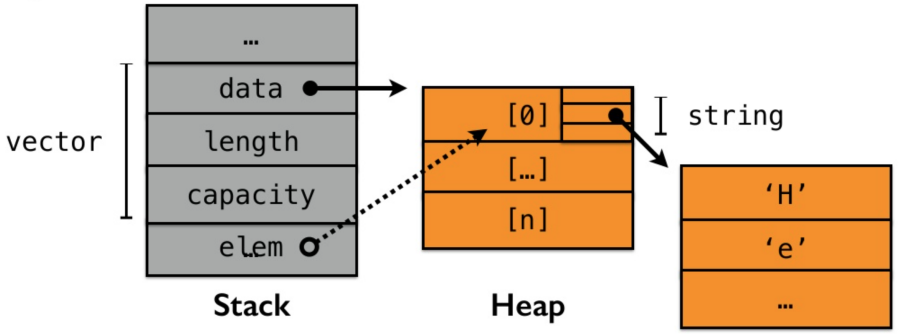
What is **control?**

```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
  …
}
```

Stack and inline layout.

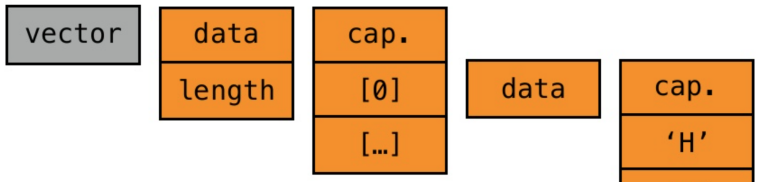
Interior references

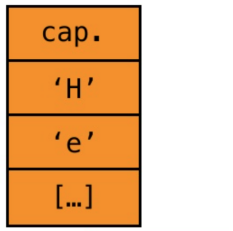
Deterministic destruction



Zero-cost abstraction

Ability to define **abstractions** that **optimize away to nothing**.





Not just memory layout:
- Static dispatch
- Template expansion
- …

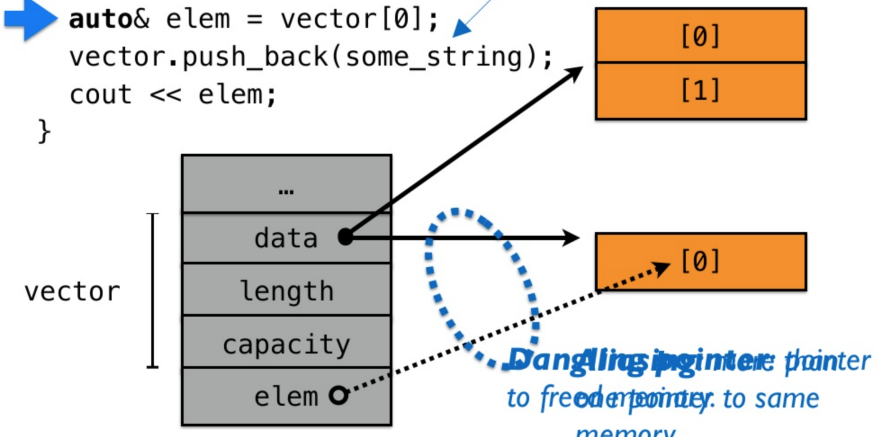
Java

C++

What is **safety?**

```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
  vector.push_back(some_string);
  cout << elem;
}
```

Mutating the vector freed old contents.



What about using a GC? (like Haskell, Java, Go, . . .)

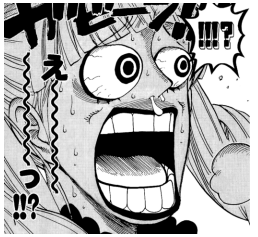
- Garbage collection:
  - The programmer allocates vectors, strings, etc.
  - The runtime system periodically sweeps through memory, looks for unreferenced data and deallocates it.
- ✘ Loss of control
- ✘ Runtime overhead
- ✘ Doesn't help with other safety issues: iterator invalidation, data races, etc.

So, what is Rust's solution? **Ownership**

What about using a GC? (like Haskell, Java, Go, . . .)

- Garbage collection:
  - The programmer allocates vectors, strings, etc.
  - The runtime system periodically sweeps through memory, looks for unreferenced data and deallocates it.
- ✘ Loss of control
- ✘ Runtime overhead
- ✘ Doesn't help with other safety issues: iterator invalidation, data races, etc.

So, what is Rust's solution? **Ownership** *(affine linear typing)*



CIS 352 Rust Overview 13 / 1

Observation from C++-land

Danger arises from…

Aliasing + **Mutation**

Hides dependencies.

Causes memory to be freed.

```
{   auto& e = v[0];    …    v.push_back(…);   }
```

∴ Outlaw doing both at once and appoint the compiler Sheriff.

Three Basic Patterns

Ownership	`fn foo(v: T) { ...}`
Shared Borrow	`fn foo(v: &T) { ...}`
Mutable Borrow	`fn foo(v: &mut T) { ...}`

The next bunch of slides are from
CIS 198: Rust Programming, University of Pennsylvania, Spring 2016,
`http://cis198-2016s.github.io/slides/01/`

Ownership

- A variable binding takes ownership of its data. [lifetimes]
  A piece of data can only have one owner at a time.
- When a binding goes out of scope, the bound data is released automatically.
  For heap-allocated data, this means de-allocation.
- Data must be guaranteed to outlive its references.

```
fn foo() {
    // Creates a Vec object.
    // Gives ownership of the Vec object to v1.
    let mut v1 = vec![1, 2, 3];
    v1.pop();
    v1.push(4);
    // At the end of the scope, v1 goes out of scope.
    // v1 still owns the Vec object, so it can be cleaned up.
}
```


Move Semantics

```
let v1 = vec![1, 2, 3];
let v2 = v1;   // Ownership of the Vec object moves to v2.
println!("", v1[2]); // error: use of moved value ‘v1‘
```

- `let v2 = v1;`
  - We don't want to copy the data, since that's expensive.
  - The data cannot have multiple owners.
  - **Solution:** move the Vec's ownership into v2, and declare v1 invalid.
- `println!("{}", v1[2]);`
  - We know that v1 is no longer a valid variable binding, ∴ error!
  - Rust can reason about this at compile time, ∴ compiler error.
  - Moving ownership is a compile-time semantic.
    It doesn't involve moving data during your program.

Ownership does not always have to be moved

- Rust would be a pain to write if we were forced to explicitly move ownership back and forth.

```
fn vector_length(v: Vec<i32>) -> Vec<i32> {
    // Do whatever here,
    // then return ownership of ‘v‘ back to the caller
}
```

- The more variables you had to hand back (think 5+), the longer your return type would be!

Borrowing

- In place of transferring ownership, we can borrow data.
- A variable's data can be borrowed by taking a reference to the variable (i.e., aliasing); ownership doesn't change.
- When a reference goes out of scope, the borrow is over.
- The original variable retains ownership throughout.

```
let v = vec![1, 2, 3];
let v_ref = &v;  // v_ref is a reference to v.
assert_eq!(v[1], v_ref[1]); // use v_ref to access the data
                            // in the vector v.
// BUT!
let v_new = v;  // Error, cannot transfer ownership
                // while references exist to it.
```

Borrowing

```
// ‘length‘ only needs ‘vector‘ temporarily, so it is borrowed.
fn length(vec_ref: &Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on it.
    vec_ref.len()
}
fn main() {
    let vector = vec![];
    length(&vector);
    println!("{:?}", vector); // this is fine
}
```

- References, like bindings, are immutable by default.
- The borrow is over after the reference goes out of scope (at the end of `length`).
- (`usize` =The pointer-sized unsigned integer type.)

Borrowing

```
// ‘push‘ needs to modify ‘vector‘ so it is borrowed mutably.
fn push(vec_ref: &mut Vec<i32>, x: i32) {
    vec_ref.push(x);
}
fn main() {
    let mut vector: Vec<i32> = vec![];
    let vector_ref: &mut Vec<i32> = &mut vector;
    push(vector_ref, 4);
}
```

- Variables can be borrowed by mutable reference: `&mut vec_ref`.
  - `vec_ref` is a reference to a mutable `Vec`.
  - The type is `&mut Vec<i32>`, not `&Vec<i32>`.
- Different from a reference which is variable.
- You can have exactly one mutable borrow at a time.
  Also you cannot dereference borrows (changes ownership).

Borrowing Rules

1. You can’t keep borrowing something after it stops existing.
2. One object may have many immutable references to it (`&T`).
3. **OR** exactly one mutable reference (`&mut T`) (not both).

Borrowing Prevents: Use-after-free bugs

Valid in C, C++,...

```
let y: &i32;
{
    let x = 5;
    y = &x; // error: ‘x‘ does not live long enough
}
println!("{}", *y);
```

This eliminates vast numbers of memory safety bugs *at compile time!*

The ownership rules also prevent other things

Under the standard ownership rules:

- You **cannot** implement doubly-linked lists (and circular structures in general).
- You **cannot** call C libraries.
- ...

Unsafe Rust

```
unsafe {
   ...
}
```

- Relaxes some of the checking rules.
- Allows C libraries calls.
- Allows access to raw pointers.
- Allows you to implement language extensions, e.g., doubly-linked lists, garbage-collected pointers, etc.

Concurrency

- The ownership rules also turn out to be useful for implementing safe concurrency (e.g., threads, interprocess communication, etc.)
- Standard currency primitives are not built in to Rust — they can be defined via ownership rules.
- Easy to use, safe (e.g., data-race free) concurrent programming is a big deal.

Other goodies

- Crate and cargo — a modern, simple to use project manager to track dependencies, etc.
- Hygienic macros.
- Growing collection of libraries (nowhere as near mature or complete as C, C++, Java, etc.)
- Etc.
- See `https://www.rust-lang.org` for other resources.
- If you want to learn Rust, don't trust anything over three years old. *(Earlier versions were quite different.)*

Nota Bena

- There isn't a type-soundness proof for Rust yet.
  That is, the story Rust tells about what its types mean is nice — but can you prove that the story is correct?
- **Problems:**
  - Rust is a moving target.
  - Rust includes `unsafe`-blocks.