

# ECE326

## PROGRAMMING LANGUAGES

### **Lecture 28 : Introduction to Rust**

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

# Introduction

- Designed and developed at Mozilla Research
- First released in Summer 2010
  - Stable since Spring 2015
- Systems language focused on *safety*
  - Type safety, memory safety, safe concurrency
- Performance comparable to C/C++
- Compiler performs extensive safety checks
  - Compile time can be much slower than C/C++ compilers
- Syntactically similar to C/C++ and Haskell

# Installation

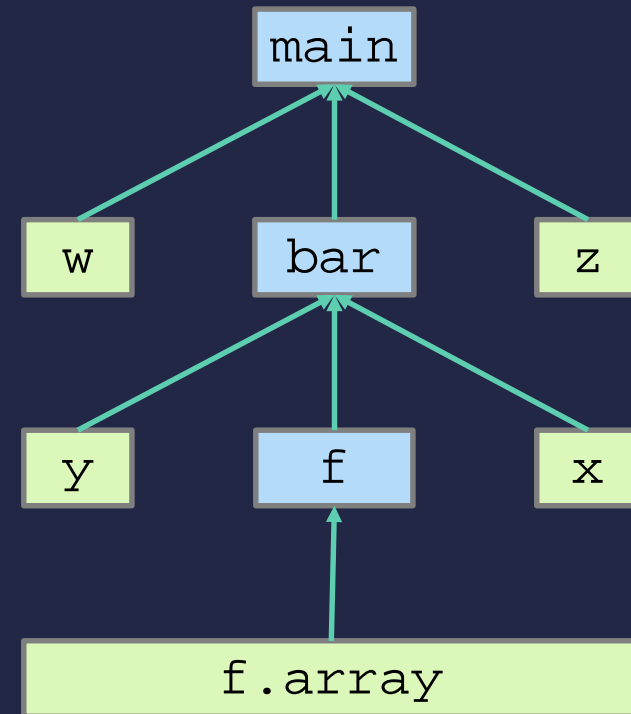
- Custom installed on UG machines
- Add RUSTUP\_HOME to environment variable
  - `setenv RUSTUP_HOME /cad2/ece326f/rust` # add to ~/.cshrc
  - `export RUSTUP_HOME=/cad2/ece326f/rust` # add to ~/.bashrc
- Run `rustc --version`
  - Make sure you get this output:
    - `rustc 1.38.0 (625451e37 2019-09-23)`
- <https://rustup.rs/>
  - Installs latest version of Rust
    - Follow its instruction to install for your home machine

# Alias in Rust

- No aliases
  - Cannot have two pointers pointing to same memory address
  - Guarantees memory safety without garbage collection
  - Compiler can deduce when to free memory
- Ownership
  - All lvalues have unique owners
    - E.g. the owner of local variables is their function
  - When the owner goes out of scope, it frees what it owns
  - Without alias, no cycles can be formed
    - Memory ownership will take the shape of a tree

# Ownership

```
struct Foo {  
    int * array;  
    Foo() : array(new int[5]) {}  
};  
  
int bar(int y) {  
    Foo f;  
    int x = y + 3;  
    return x + f.array[0];  
}  
  
int main() {  
    int * w = new int(5);  
    int z = bar(*w);  
    return z;  
}
```



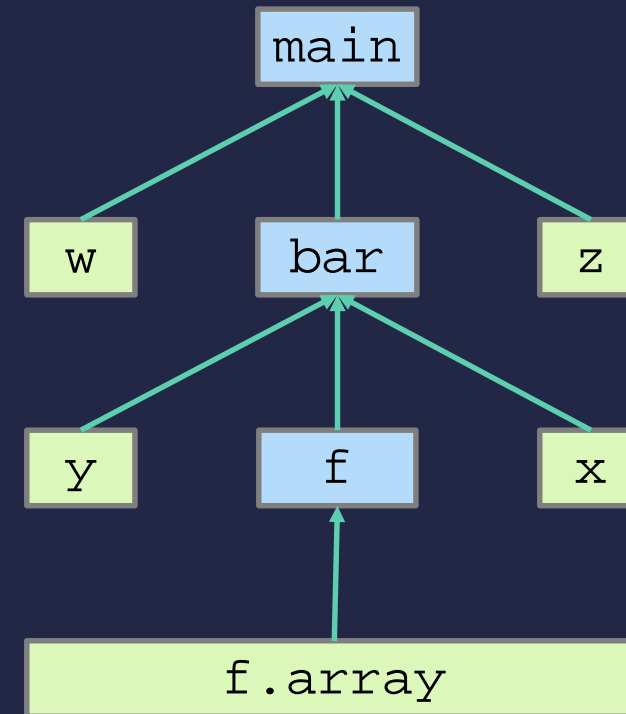
# Ownership

```
struct Foo {  
    int * array;  
    Foo() : array (new int[5]) {}  
    ~Foo() { delete array; }  
};
```

```
int bar(int y) {  
    Foo f;  
    int x = y + 3;  
    return x + f.array[0];  
}
```

```
int main() {  
    int * w = new int(4);  
    int z = bar(*w);  
    delete w;  
    return z;  
}
```

**delete**  
statements  
automatically  
inserted by  
compiler after  
static analysis  
of ownership

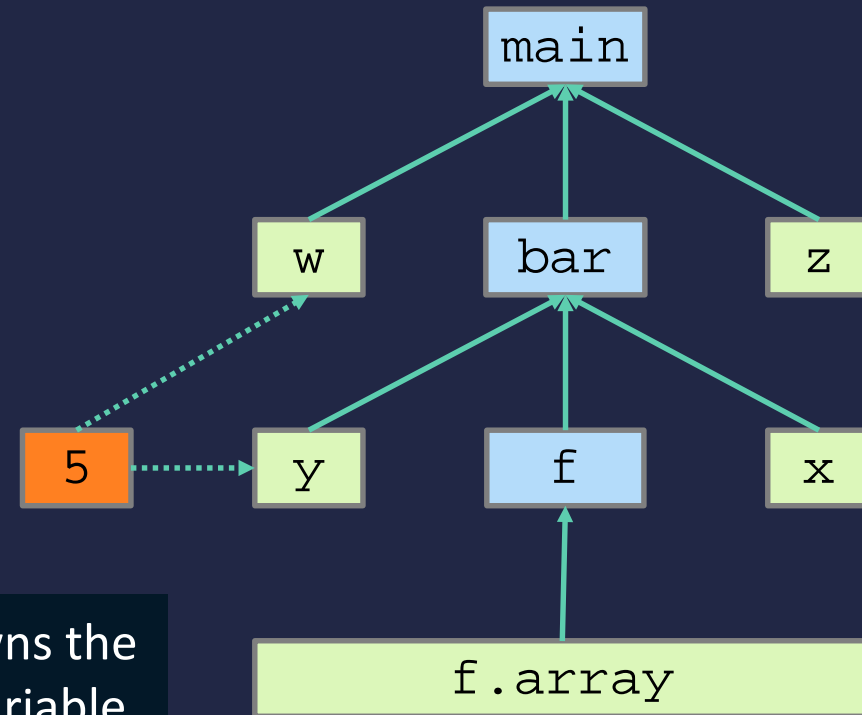


# Passing Variable

```
struct Foo {  
    int * array;  
    Foo() : array(new int[5]) {}  
};
```

```
int bar(int * y) {  
    Foo f;  
    int x = *y + 3;  
    return x + f.array[0];  
}
```

```
int main() {  
    int * w = new int(5);  
    int z = bar(w);  
    return z;  
}
```



Who owns the  
heap variable  
"5" now?

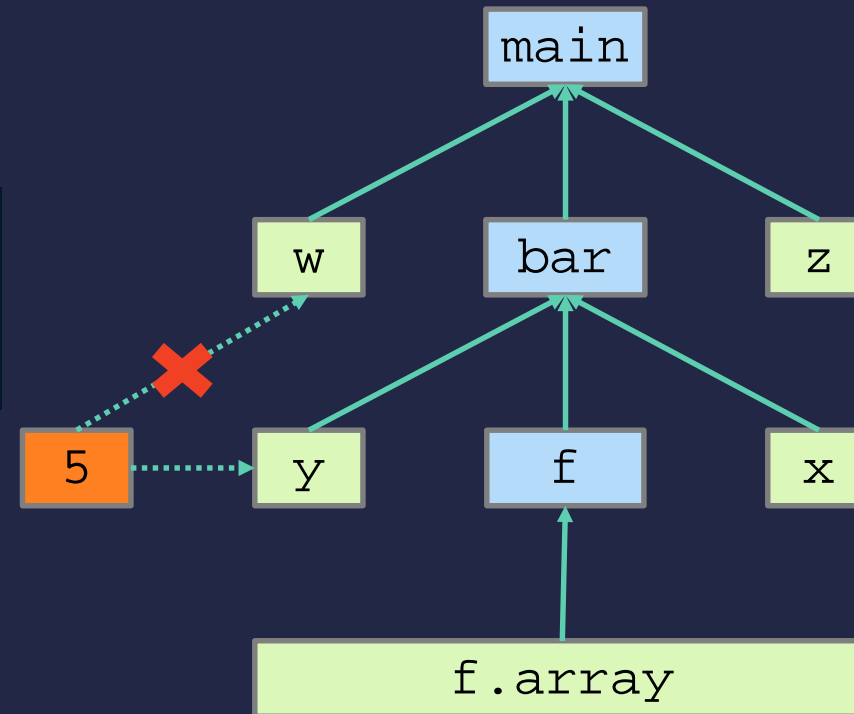
# 1. Takeover (Move)

```
struct Foo {  
    int * array;  
    Foo() : array(new int[5]) {}  
};
```

```
int bar(int * y) {  
    Foo f;  
    int x = *y + 3;  
    delete y;  
    return x + f.array[0];  
}
```

```
int main() {  
    int * w = new int(5);  
    int z = bar(w);  
    /* cannot use w anymore */  
    return z;  
}
```

By default, bar takes over ownership.



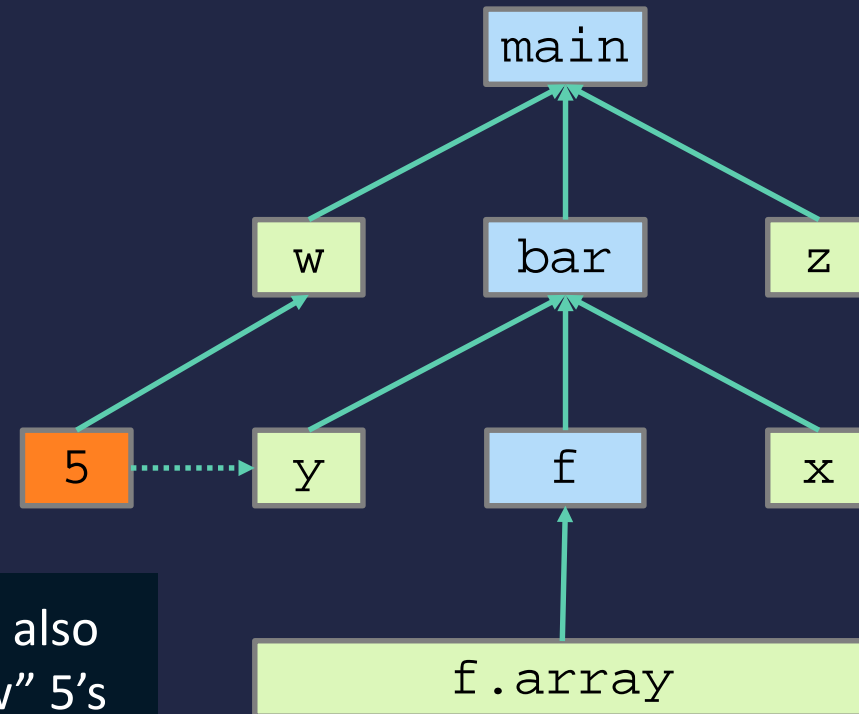


## 2. Borrow

```
struct Foo {  
    int * array;  
    Foo() : array(new int[5]) {}  
};
```

```
int bar(borrowed int * y) {  
    Foo f,  
    int x = *y + 3;  
    return x + f.array[0];  
}
```

```
int main() {  
    int * w = new int(5);  
    int z = bar(w);  
    delete w;  
    return z;  
}
```



bar can also  
"borrow" 5's  
ownership  
from main

# Ownership

- Borrow
  - Lender must *outlive* borrower
- Lifetime
  - Interval in which an entity is valid
  - Begins when a variable is created, ends when it's destroyed
- Passing variable
  - If variable can be copied (e.g. primitive types), pass by value
  - If parameter declared as borrow, lend variable *if possible*
  - Otherwise, performs a move (give up ownership)

# Hello World

- Like in C/C++, requires a main function
- Function declared using **fn** keyword
- `println!` is a *macro* function (denoted by `!` symbol)
- To compile, call `rustc -o hello main.rs`
  - `hello` is the name of executable

```
/* main.rs */  
// Rust uses same as C/C++ comments  
fn main() {  
    println!("hello world");  
}
```

# Function

- General syntax

```
fn funcname(argname: argtype...) -> returntype {  
    ... statements ...  
    returnvalue  
}
```

- In Rust, the last expression (no semicolon) is returned

```
fn foo(a: f64, b: f64) -> f64 {  
    a * b                // return value  
}
```

```
fn baz(a: f64, b: f64) -> f64 {  
    return a * b;        // same as above  
}
```

# Variable

- Variable declaration in Rust is a *statement*
- Type can usually be automatically deduced
  - If assigned a literal, it has the type of the literal
  - If assigned a return value from a function, the return type

```
fn baz(a: f64, b: f64) -> f64 {  
    return a * b;    // same as above  
}
```

```
let a = 20;           // an integer type (e.g. i32 or i64)  
let b: i64 = 30;      // explicitly specified as i64  
let c = baz(2.0, 3.0); // f64 by type inference
```

# Strongly Typed

- Does not allow implicit conversion, even if its widening
- Use `as` operator to cast between types

```
let a: i32 = 1;  
let b: i64 = 2;  
let c = a * b;    // FAIL - does not allow implicit conversion
```

error: cannot multiply `i64` to `i32`

```
let c = a as i64 * b;  
let x = [1, 2, 3];  
println!("{}", x[a]);
```

error: cannot be indexed by `i32`

# Println Format

- Similar to Python's `string.format()`
- Each set of curly braces is an argument
- Type conversion automatically done (coerce to string)
  - Argument needs to implement `Display` or `Debug` trait
    - For now, it is similar to overriding a virtual function in base class

```
let s = "hello world";
```

```
let a = 5;
```

```
println!("{}", {}, s, a);
```

```
hello world, 5
```

# Immutability

- By default, variables in Rust are *constant* (immutable)
- If you want to change it, use **mut** keyword

```
let x = 5;  
println!("The value of x is: {}", x);  
x = 6;           // BAD!
```

error: cannot assign twice to immutable variable

```
let mut x = 5;  
x = 6;           // OK
```



# Constant

- Constant in Rust behaves like `constexpr` in C++
- It is a compile-time expression
- Can be declared in global scope (unlike `let`)
- Type must be specified

```
const MAX_POINTS: u32 = 100_000;
```

- underscore in literal has no semantic meaning
  - It helps programmer to more easily read bigger numbers

# Shadowing

- Rust allows same name to be used multiple times
- Previous bindings are “shadowed”, cannot be accessed
- Useful once code becomes more complex

```
let x = 5;  
let x = x + 1;  
let x = x * 2;  
println!("The value of x is: {}", x);
```

The value of x is: 12

```
let spaces = "  ";  
let spaces = spaces.len();    // OK to change type as well
```

# Primitive Types

- Boolean
  - true or false
- Floating Point
  - f32 or f64
- Integer
  - signed: i8, i16, i32, i64, isize
  - unsigned: u8, u16, u32, u64, usize
  - usize and isize depends on architecture (either 32 or 64 bits)
- Character
  - char
  - Can be unicode characters as well. Not just ASCII.
    - i.e. Unlike char in C, not guaranteed to be 1 byte

# String

- utf-8 encoded (thus supports unicode characters)
- String literal
  - The type is `&str` (reference to a string slice)

```
let s: &str = "hello world";
```

- String slice
  - Reference to immutable string data somewhere
    - String literals are stored in program binary
  - Similar to `const char *` in C++

# String

- String type
  - mutable string, content can be updated

```
let mut s1 = String::from("foo"); // convert to mutable
s.push_str("bar");                // append "bar" to s
```

```
let mut s2 = "lo".to_string(); // another way to convert
s.push('l');                    // push a single character
```

```
let mut s3 = String::new();      // creates an empty string
```

```
// formatted string (created from a macro function)
let s = format!("{}", s1, s2, s3);
```

# Tuple

- Rust has built-in tuple type, similar to Python

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
```

- Allows for unpacking

```
let tup = (500, 6.4, 1);  
let (x, y, z) = tup;
```

- Also allows access to each element

```
let a = x.0;  
let b = x.1;  
let c = x.2;
```

# Array

- Similar in syntax to C array
- Performs compile time bound checking
- Tracks its own length, similar to Java array

```
let a = [1, 2, 3, 4, 5];  
let months = ["January", "February", "March", "April", ...];  
let first = a[0];  
let second = a[1];
```

```
// array does not support Display trait, must use {:?}  
println!("{:?}: {}", a, a.len());
```

```
[1, 2, 3, 4, 5]: 5
```

# Function

- Does not support function overloading
- Does not support default parameters
- Like in C, uses block scoping
- Blocks are expressions in Rust!

```
let y = {  
    let x = 3;  
    x + 1  
};
```

```
println!("The value of y is: {}", y);    // 4
```



# Unit Type

- `()`
  - Looks like an empty tuple
  - Is a type of its own
  - It's purpose is to be “useless”
- Everything in Rust is an expression
  - Similar to many functional languages
- A function without return type returns it
  - Similar to void in C/C++