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 an 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

Main Differences

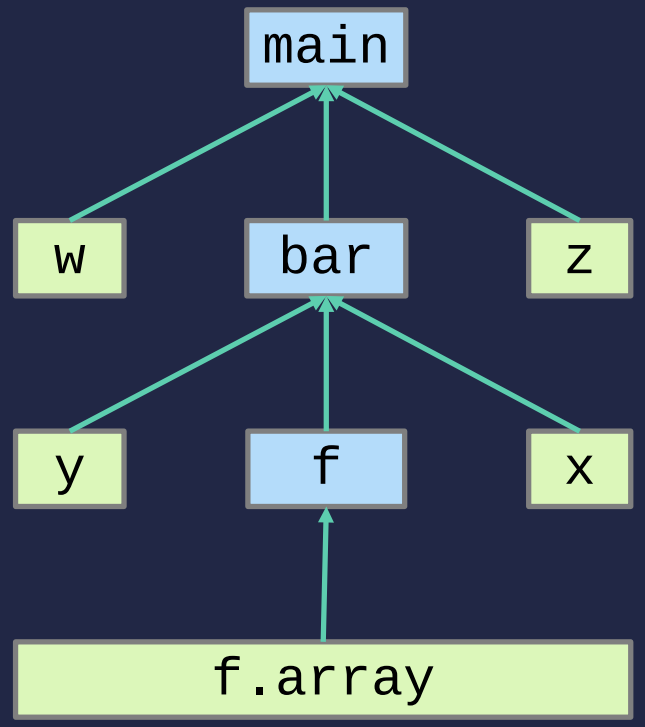
- No aliases
  - Cannot have two pointers pointing to same memory address
  - Guarantees memory safety without garbage collection
  - Compiler can deduce when to free memory
- Ownership
  - All lvalues have unique owners
    - E.g. the owner of local variables is their function
  - When the owner goes 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;
}
```



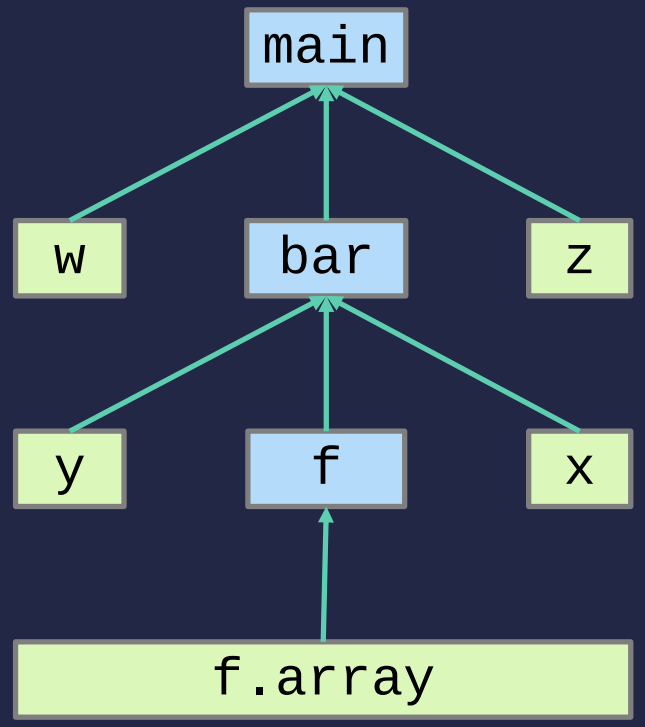
Ownership

```
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

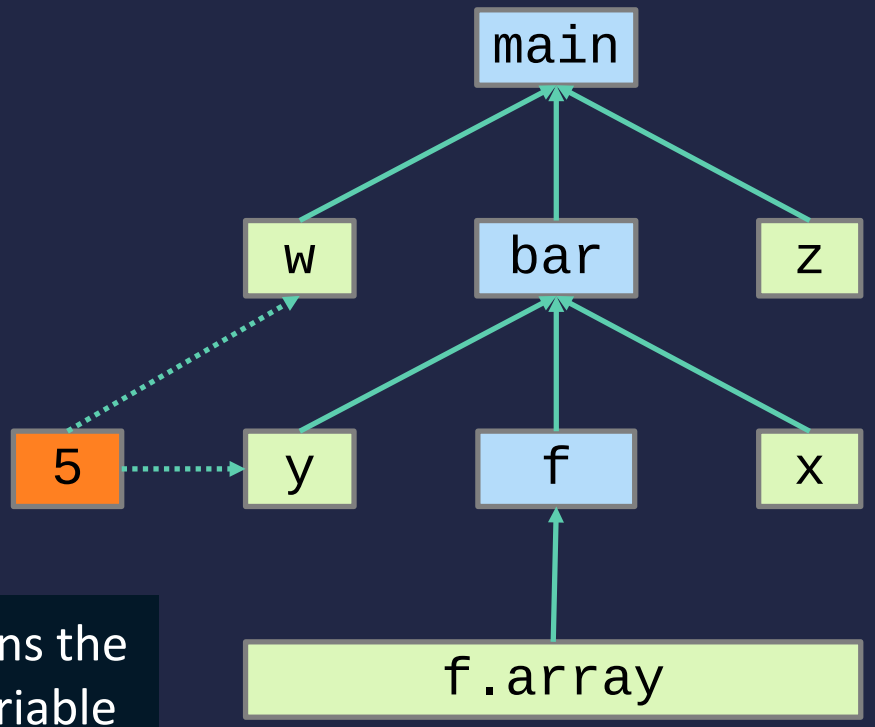


Passing Variable

```
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?

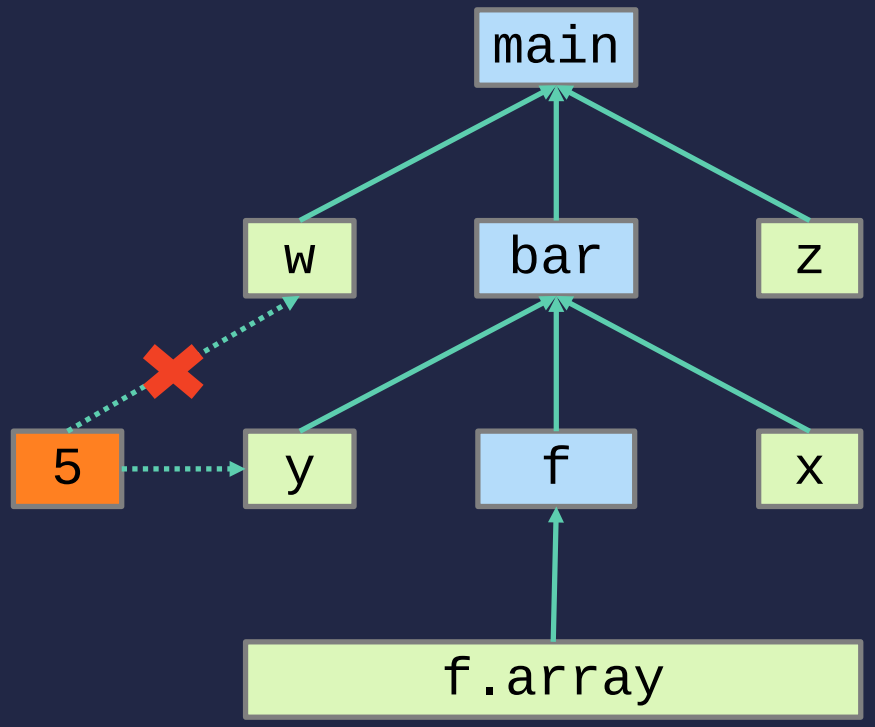
1. Takeover (Move)

```
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.

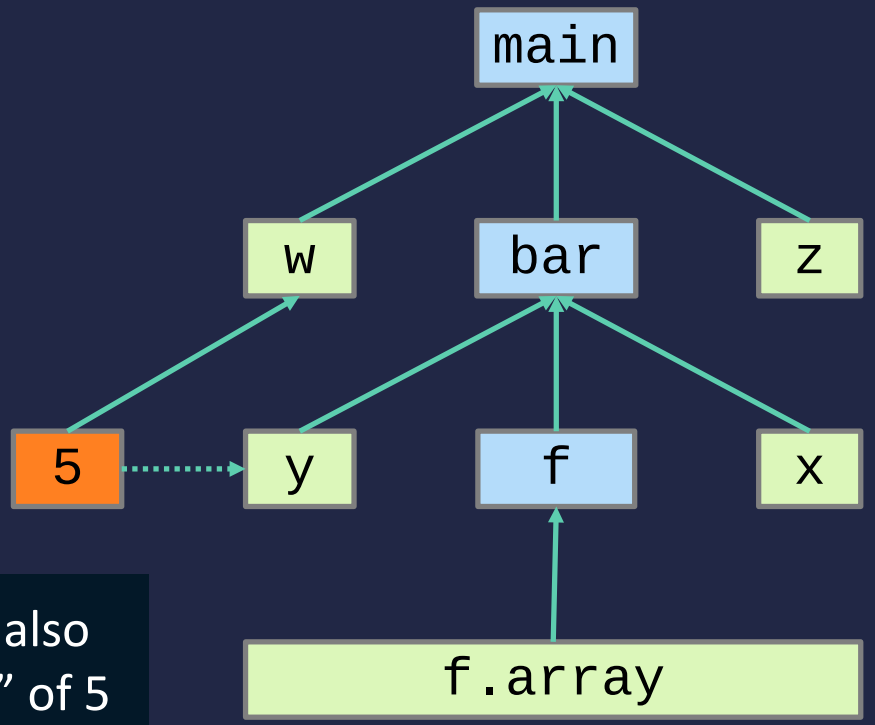


2. Borrow

```
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" of 5 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”);
}
```