ECE326

PROGRAMMING LANGUAGES

Lecture 13 : Introduction to Rust

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Introduction

- Designed and developed at Mozilla Research
  - By Graydon Hoare
- First released in Summer 2010
  - Stable since Spring 2015
- Focused on *safety*
  - Memory safety, type safety, thread safety

Memory Safety

- Protection from invalid memory access
- Memory *unsafe* languages
  - Allows arbitrary pointer arithmetic (C/C++)
- Memory errors
  - Buffer overflow
  - Use after free (dangling pointers)
  - Double free
- Solution
  - runtime error detection (e.g. Java)
  - Static program analysis (e.g. Rust)

Type Safety

- Protection from incorrect use of a value
- Example
  - untagged union (C/C++)
  - Union
    - All member variables share *the same* memory location
    - Can lead to type-unsafe usage!
  - Solution: tagged union
    - Adds a tag field to indicate which member is in use

```
union Foo {
        int i;
        float f;
};

// in main()
Foo u;
u.i = atoi(argv[argc-1]);
printf("%f\n", u.f);

> ./union 1237864534
1640970.750000
```

Thread Safety

- Protection from incorrect use of shared data
  - Only applicable to multi-threaded code
- Thread
  - Independent sequence of program instructions
- Thread unsafe operation can cause data corruption
  - Leads to data loss, security vulnerability, etc.
- Rust ensures data cannot be improperly shared

Rust

- A System programming language
  - Performance comparable to C/C++
- Compiler performs extensive safety checks
  - Compile time much slower than C/C++ compilers
- Syntactically similar to C/C++ and Haskell
- Multi-paradigm
  - Functional programming
  - Generic programming
  - NOT object-oriented programming

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 1.47.0 (18bf6b4f0 2020-10-07)`
- https://rustup.rs/
  - Installs latest version of Rust
    - Follow its instruction to install for your home machine

Alias in Rust

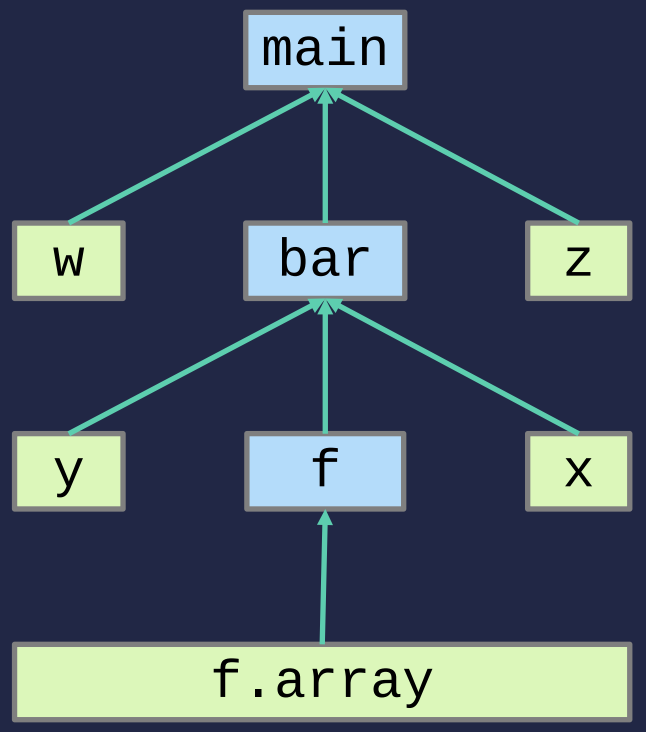
- No aliases
  - Except when it is 100% safe
  - Compiler can deduce when to free memory
  - Guarantees memory safety without garbage collection
- Ownership
  - All variables have unique owners
    - E.g. the owner of local variables is their function
  - When the owner goes out of scope, it frees what 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;
	/* ownership diagram */
	return x + f.array[0];
}

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



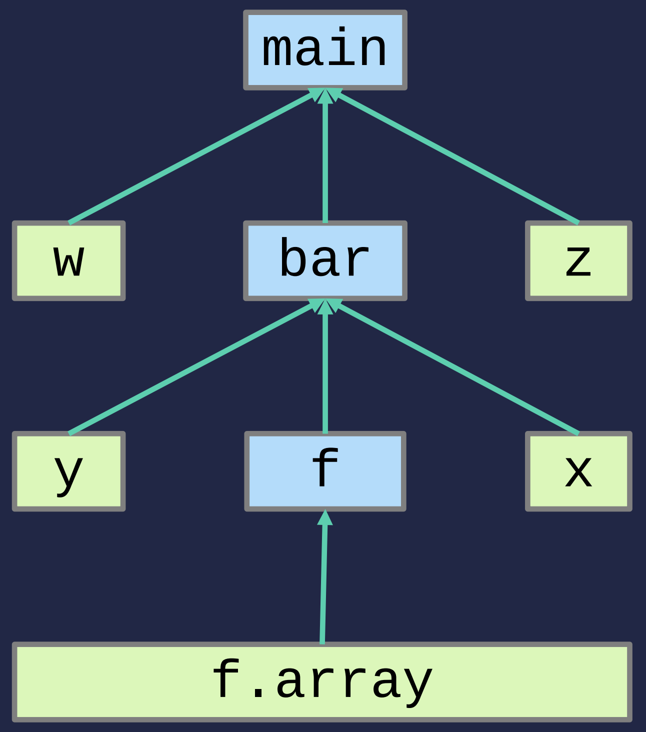
Ownership

```
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



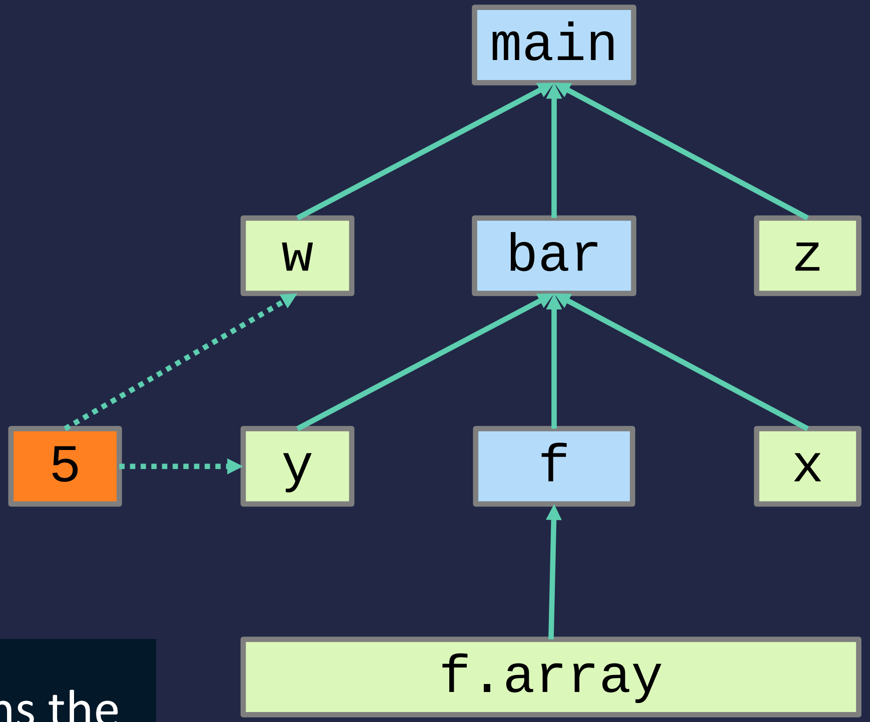
Passing Variable

```
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 integer, 5, now?



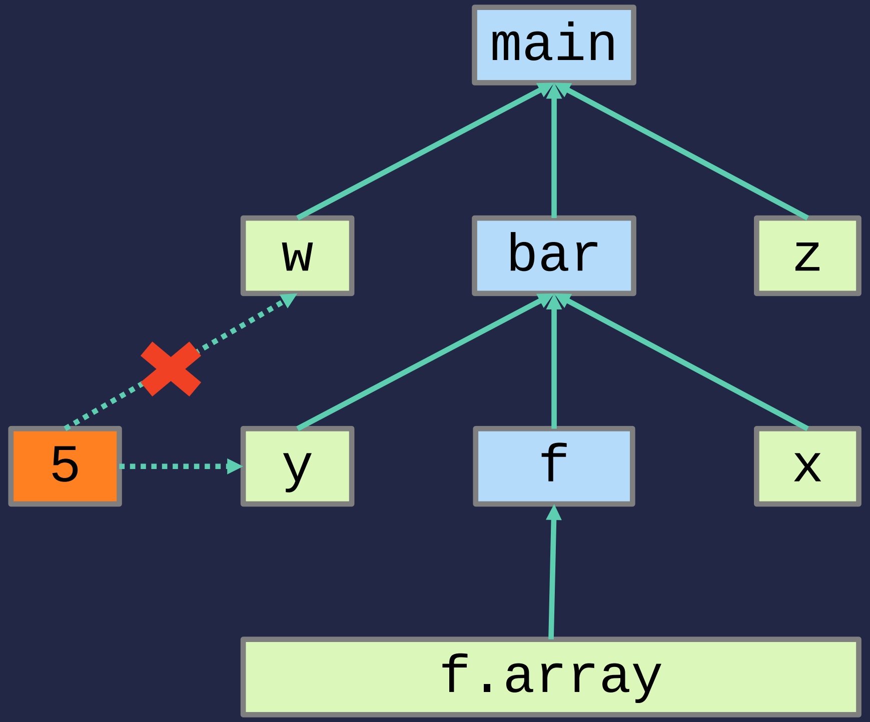
1. Takeover (Move)

```
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.



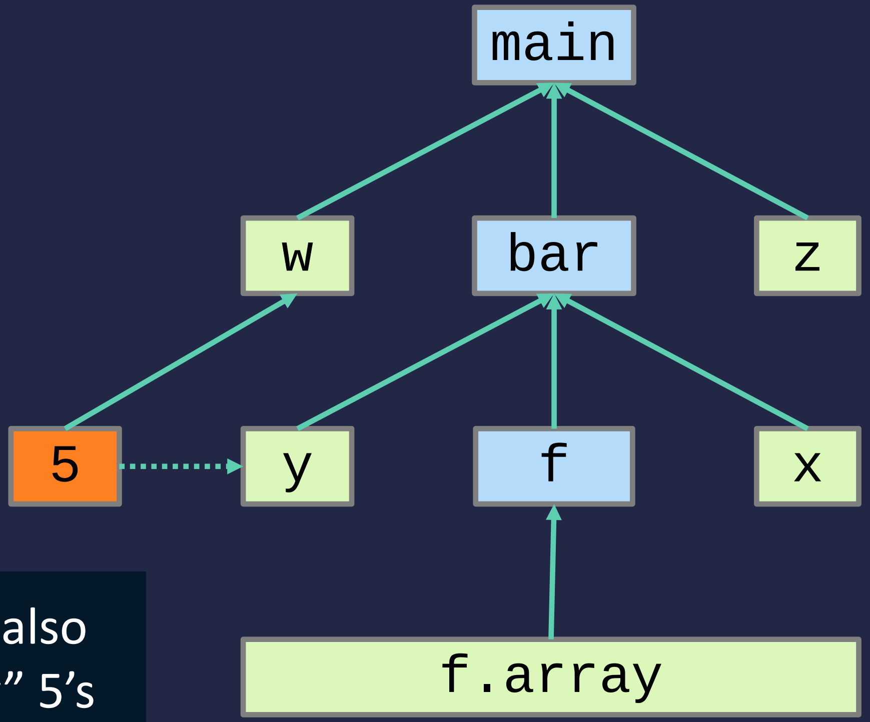
2. Borrow

```
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



Passing Variable

- Borrow
  - Lender must *outlive* borrower
- Order of options
  - If parameter declared as borrow, lend variable *if possible*
  - If variable can be copied (e.g. primitive types), pass by value
  - 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
```

Primitive Types

- Boolean
  - bool: true or false
- 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
- Floating Point
  - f32 or f64

String Literal

- 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++

Println Format

- Similar to Python's string.format()
- Each set of curly braces is an argument
- Type conversion automatically done
  - Argument needs to implement Display or Debug trait
    - Similar to overriding __str__ or __repr__ in Python

```
let s = "hello world";
let a = 5;

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

hello world, 5
```

Immutability

- By default, variables in Rust are 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
```

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
```

Constant

- 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 read bigger numbers more easily

Mutable String

- String type
  - mutable string, content can be updated

```
let mut s1 = String::from("foo");  // convert to mutable
s1.push_str("bar");                // append “bar” to s
>> foobar
let mut s2 = "lo".to_string();  // another way to convert
s2.push('l');                   // push a single character
>> lol
```

Mutable String

- String type
  - mutable string, content can be updated

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

let mut s2 = "lo".to_string();  // another way to convert
s2.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);
>> foobar-lol-
```

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 = tup.0;
let b = tup.1;
let c = tup.2;
```

Array

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

```
let a = [1, 2, 3];
let months = ["January", "February", "March", "April", …];
let first = a[0];   // uses index operator
let [x, y, z] = a;  // supports unpacking too

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

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

Casting

- Rust has strict type conversion rules
  - All type conversion must be explicit
- `as` operator
  - Casts one type to another (if allowed)
- Example: array index must be the type `usize`

```
let a = [1, 2, 3];
let b: i32 = 1;
let c = a[b]; // error: slice indices are of type `usize`
let c = a[b as usize]; // OK
println!("{}", c);
>> 2
```

Function

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

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

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

Unit Type

- ()
  - The value looks like an empty tuple
  - Its type is the unit type
  - Purpose is to be “useless”
- Almost everything in Rust is an expression
  - A property of functional programming languages
- A function without return type returns it
  - Similar to None in Python