301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

***AP-23**: RUST #1*

The RUST programming language

- Brief history
- Memory safety
- Avoiding Aliases + Mutable
- Ownership and borrowing
- Lifetimes
- Enums, Structs, Generics, Traits…
- Unsafe
- Smart Pointers
- Concurrency

Brief History

- Development started in 2006 by Graydon Hoare at Mozilla.
- Mozilla sponsored RUST since 2009, and announced it in 2010.
- In 2010 shift from the initial compiler in **OCaml** to a self-hosting compiler written in **Rust**, **`rustc`**: it successfully compiled itself in 2011.
- **`rustc`** uses **LLVM** as its back end.
- Most loved programming language in the Stack Overflow annual surveys since 2016.
- February 8, 2021: The development of the language passes to the Rust Foundation (non-profit, independent) funded by da Mozilla, Microsoft, Google, AWS e Huawei.

On RUST goals and syntax

- **Rust** is a general purpose, system programming language with a focus on safety, especially safe concurrency, supporting both functional and imperative paradigms
- Main goal: ensuring safety without penalizing efficiency
- Concrete syntax similar to C and C++ (blocks, `if-else`, `while`, `for`), `match` for pattern matching
- *Despite the superficial resemblance to C and C++, the syntax of Rust in a deeper sense is closer to that of the ML family of languages as well as the Haskell language.*
- Nearly every part of a function body is an expression (including `if-else`).
- No Runtime required (GC, Dynamic typing/binding,…)
- More control (over memory allocation/destruction…)

More than that …

more control,
less safety

less control,
more safety

Rust

more control,
more safety

Rust overview

Performance, as with C
- Rust compilation to object code for bare-metal performance

But, supports memory safety
- Programs dereference only previously allocated pointers that have not been freed
- Out-of-bound array accesses not allowed

With low overhead
- Compiler checks to make sure rules for memory safety are followed
- Zero-cost abstraction in managing memory (i.e. no garbage collection)

Via
- Advanced type system
- **Ownership**, **borrowing**, and **lifetime** concepts to prevent memory corruption issues

But at a cost
- Cognitive cost to programmers who must think more about rules for using memory and references as they program

Memory safety

- Rust is designed to be **memory safe**, even in the presence of concurrency:
  - No null pointers
  - No dangling pointers
  - No double frees
  - No data races
  - No iterator invalidation
- These properties are guaranteed **statically**: if the program compiles it will never manifest those problems.
- Memory safety is obtained with a careful combination of several techniques: linguistic design choices, memory management policies, and powerful static (data-flow) analysis

Null pointers

- **Problem: accessing a variable which does not hold a value**
- Two approaches to guarantee that a **variable** holds a **value** when accessed:
  1. Check that it has been assigned, via data flow analysis
  2. Use **default values**
- In Java, solution 1. for local vars of methods, solution 2. for instance and static variables.

Why???

- Sol. 2 is much simpler, sol. 1 hardly applicable to “global variables”
- Numeric variables typically have 0 as default value
- **Tony Hoare** introduced **Null** references in ALGOL W.
  - “The billion dollar mistake”…
- **NullPointerException** most thrown exception in Java

Avoiding null pointers in Rust

- A **Null** value does not exist in Rust
- Data values can only be initialized through a fixed set of forms, requiring their inputs to be already initialized.
- Compile time error if any branch of code fails to assign a value to the variable.
- Static/global variables must be initialized at declaration time
- For ***nullable types***, a generic **Option<T>** type exist, playing the role of Haskell's Maybe or Java's Optional

```
enum std::option::Option<T> {
    None,
    Some(T)
}
```

Digression: Primitive types in Rust

- Numeric types:
  - **`i8 / i16 / i32 / i64 / isize`**
  - **`u8 / u16 / u32 / u64 / usize`**
  - **`f32 / f64`**
- **`bool`**
- **`char`** (4-byte unicode)
- Type inference for variables declarations with **let**
- No overloading for literals: type annotations to disambiguate
- Tuples: like in Haskell
- Arrays: with fixed length. **Runtime check for out-of-bound!**

```
fn main() {
    let k = 3; // 3u8, 3.0, 3.2f32, ...
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
    println!("The value of y is: {}",y);
    println!("The value of tup.1 is: {}",tup.1);
    let a: [i32;5] = [1,2,3,4,5];
    let b: [i32;6] = [3;6]; // [3,3,3,3,3,3]
}
```

Using Option

```
enum std::option::Option<T> {
    None,
    Some(T)
}
```

```
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        None
    } else {
        Some(dividend / divisor)
    }
}

fn try_division(dividend: i32, divisor: i32) {
    // `Option` values can be pattern matched, just like other enums
    match checked_division(dividend, divisor) {
        None => println!("{} / {} failed!", dividend, divisor),
        Some(quotient) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        } } }

fn main() {
    try_division(54,9); try_division(7,0);
    let opt_float = Some(0f32);
    // Unwrapping a `Some` variant will extract the value wrapped.
    println!("{:?} unwraps to {:?}", opt_float, opt_float.unwrap());
}
```

Dangling pointers: example in C++

- **Problem: Pointers to invalid memory location**
  - Pointers to explicitly deallocated objects;
  - Pointers to locations beyond the end of an array;
  - Pointers to objects allocated on the stack; …
- Unpredictable effects
  - Random results
  - Segmentation fault
  - Corruption of memory manager

```
// C++ code
string *s;
{
   string s1 = "scope 1";
   s = &s1;
}
{
   string s2 = "scope 2";
}
cout << *s << endl;
```

Prints "scope 1" if compiled with x86-64 clang 13.0.1, but it prints "scope 2" if compiled with x86-64 gcc 11.2 (see https://godbolt.org/)

Double free: example in C++

- **Problem: A memory location in the heap is reclaimed twice**
- This can happen in languages with explicit deallocation of memory (like C, C++)
- A double-free error could corrupt the state of the memory manager, causing a program to crash or modification of execution flow
- It could be exploited for software attacks

```
// C++ code
auto *s1 = new string("example");
auto *s2 = s1;
// ...
delete s1;
delete s2;
```

Race Condition: example in C++

- **Problem: unpredictable results in concurrent computations**
- The following multithreaded code typically prints values smaller than 20000, because of race conditions

```
// C++ code
int main() {
    int counter = 0;
    const auto task = [&] {
        for (int i = 0; i < 100000; ++i) {
        counter++;
        }
    };
    thread thread1(task);
    thread thread2(task);
    thread1.join();
    thread2.join();
    cout << counter << endl;
    return 0;
}
```

Memory management

- As usual, Rust uses a STACK of activation records, and a HEAP for dynamically allocated data structures.
- Rust favors stack allocation (default).
- The user is forced to be aware of where the data are stored: No implicit **boxing**.

```
fn main() {
    let x = 3;   // 'let' allocates a variable on the stack
    let y = Box::new(3); // y is a reference to 3 on the heap
    println!("x == y is {}", x == *y); // "x == y is true"
}
```

- Modern languages either use Garbage Collection, or leave the programmer the responsibility of managing the heap
- Pros and cons:
  - GC slows down or interrupts the execution; could be unfeasable for real-time systems
  - Memory management by programmer can introduce subtle errors
- Rust avoids both, providing deterministic management of resources, with very low overhead, using **RAII**

Immutability by default

By default, Rust variables are immutable

- Usage checked by the compiler

mut is used to declare a resource as mutable.

```
fn main() {
    let a: i32 = 0;
    a = a + 1;
    println!("a == {}", a);
}
```

```
fn main() {
    let mut a: i32 = 0;
    a = a + 1;
    println!("a == {}", a);
}
```

```
rustc 1.14.0 (e8a012324 2016-12-16)
error[E0384]: re-assignment of immutable variable `a`
 --> <anon>:3:5
  |
2 |     let a: i32 = 0;
  |         - first assignment to `a`
3 |     a = a + 1;
  |     ^^^^^^^^^ re-assignment of immutable variable

error: aborting due to previous error
```

```
rustc 1.14.0 (e8a012324 2016-12-16)
a = 1
Program ended.
```

Resource Acquisition Is Initialization

- The Resource Acquisition Is Initialization (**RAII**) programming idiom states that Resource allocation is done during **object initialization**, by the constructor, while resource deallocation (release) is done during **object destruction** (specifically **finalization**), by the destructor.
- Popular in modern C++. Small objects better allocated on stack. Large resources are on the heap (or elsewhere) and are *owned* by an object on the stack. The object is then responsible for releasing the resource in its destructor.
- The object is bound to the scope (function, block) where it is declared; when the scope closes it is reclaimed, together with any owned resource.
- Each resource has a unique owner.

Ownership System

- Rust has an **ownership system**, which supports RAII in a strict way
- Based on the concepts of ownership and borrowing
- Ownership can be summarized by three rules:

[O1] Every value is owned by a variable, identified by a name (possiby a path);

[O2] Each value has at most one owner at a time;

[O3] When the owner goes out-of-scope, the value is reclaimed / destroyed.

Move semantics of assignment

- By default, an assignment between variables has a **move semantics**: the ownership is moved from the RHS to the LHS (by [O2])

```
fn main() {
    let x = Box::new(3);
    let _y = x;  // underscore to avoid 'unused' warning
    println!("x = {}", x); // error!
}
```

- For primitive types and types implementing the **Copy trait**, assignment has a **copy semantics**
- [O2] is satisfied because a new value is created

```
fn main() {
    let x = 3;
    let _y = x;
    println!("x = {:?}", x); // OK
}
```

```
fn main() {
    let x = Option::Some(3);
    let _y = x;
    println!("x = {:?}", x); // OK
}
```

Move semantics of parameter passing

- The same with parameter passing and function return

```
fn foo<T>(z: T) -> T { // polymorphic identity function
    z
}
fn main(){
    let x = Box::new(3);
    let _y = foo(x);
    println!("x == {}", x);  // error
}
```

```
fn main(){
    let x = 3;
    let _y = foo(x);
    println!("x == {}", x); // OK
}
```

- Any value passed to the function will be reclaimed when it returns, as the formal parameters gets out of scope
- Only the returned value can survive (tuples to return more)

```
fn main(){
    let mut x = Box::new(3);
    x = foo(x);
    println!("x == {}", x); // OK
}
```

Ownership: Unique Owner

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                    a: 0,
                    b: 0
                });
    take(res);
    println!("res.a = {}", res.a);
}

fn take(arg: Box<Dummy>) {
}
```

Compiling Error!

Ownership is moved from res to arg

arg is out of scope and the resource is freed automatically

Double free: not in Rust

```
// Codice C++
auto *s1 = new string("esempio");
auto *s2 = s1;
// ...
delete s1;
delete s2;
```

- Remember the C++ code
- Rust does not allow for explicit memory deallocation.
- Because of RAII, memory is freed automatically when the owner goes out of scope
- By rule [O2], each value has only one owner.
- The move semantics of assignment guarantees that s2 only owns the string, thus when s1 goes out of scope nothing is reclaimed.

```
// Rust code
let s1 = String::new("esempio");
let s2 = s1;
```