南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Advanced Programming

Lab 2 of Rust

王薇，于仕琪

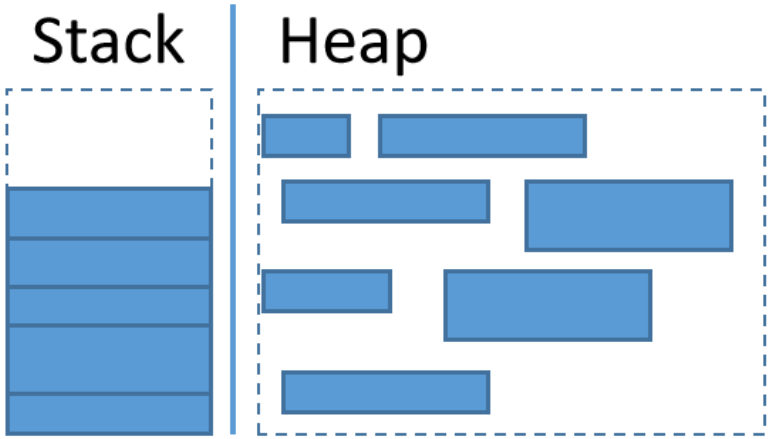
Topics

- **1. Memory details in RUST**
  - **stack vs heap (vec, String)**
  - **alignment rules on Structure**
- **2. Memory Security in RUST**
  - **ownership, reference, drop**
  - **Smart pointer**
    - **box, rc**
    - **The problem and solution of circular referencing caused by RC**
- **3. Practices**

Stack vs Heap in (1)

stack	heap
Stored in the **RAM** of the computer	Stored in the **RAM** of the computer
Fast allocation speed	**Slow** allocation speed
Stored are: **local variables, return addresses, function parameters**	Allocate a data block to the program, and the created data will be **pointed to by pointers**. **Fragmentation may occur when there are many allocations and releases**.
If it is **known at compile time how much data needs to be allocated** and **the data is not too large**, then stack can be used	If **the size of memory space is only known at runtime** or need to allocate **a large amount of data**, then heap can be used

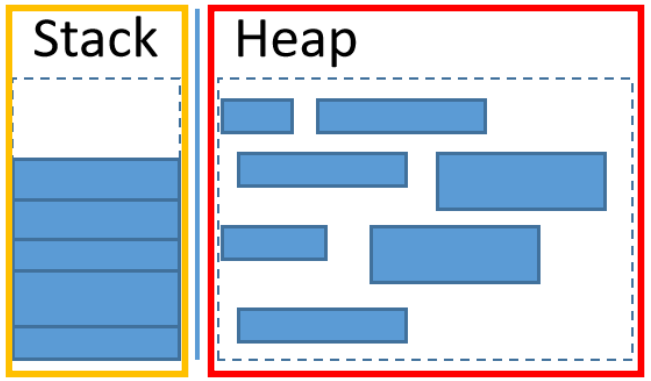
- stack： normal data type, array, .etc.



- heap： vec, string .etc.

Stack vs Heap (2)

```rust
fn main() {
        // ===============  (Stack)  ===============
        let array= [1, 2, 3, 4, 5];
        println!("=== Array DEMO ===");
        println!("addr of array in stack :       {:p}", &array);
        println!("addr of 1st element in array:   {:p}", &array[0]);
        println!("addr of 2nd element in array:   {:p}", &array[1]);
        println!("items in array: {:?}\n", array);

        // ===============  Vec (Heap)  ===============
        let vec=vec![6, 7, 8, 9, 10];
        println!("=== Vec DEMO ===");
        println!("addr of vec in stack:   {:p}", &vec);
        println!("addr of vec in heap:    {:p}", vec.as_ptr());
        println!("Vec.capacity: {}, lenght: {}", vec.capacity(), vec.len());
        println!("items in Vec: {:?}\n", vec);

        // ===============  String (Heap)  ===============
        let s=String::from("hello heap!");
        println!("=== String Demo ===");
        println!("addr of String in stack: {:p}", &s);
        println!("addr of String in heap : {:p}", s.as_ptr());
        println!("string.length: {}, capacity: {}", s.len(), s.capacity());
        println!("items in String: {:?}\n", s);
    }
```




```
=== Array DEMO ===
addr of array in stack :       0x7fff6f2fc07c
addr of 1st element in array:   0x7fff6f2fc07c
addr of 2nd element in array:   0x7fff6f2fc080
items in array: [1, 2, 3, 4, 5]

=== Vec DEMO ===
addr of vec in stack:   0x7fff6f2fc218
addr of vec in heap:    0x55c47e37ba40
Vec.capacity: 5, lenght: 5
items in Vec: [6, 7, 8, 9, 10]

=== String Demo ===
addr of String in stack: 0x7fff6f2fc3f8
addr of String in heap : 0x55c47e37ba60
string.length: 11, capacity: 11
items in String: "hello heap!"
```

Alignment rules on Structure

```rust
use std::mem::{size_of_val, align_of_val};
struct RustStyle {
    a:u8,   // 1 BYTE
    b:u32,  // 4 BYTES
    c:u16,  // 2 BYTES
    d:f64,  // 8 BYTES
}

#[repr(C)]
struct CStyle {
    a:u8,
    b:u32,
    c:u16,
    d:f64,
}
fn print_struct<T>(instance:&T, name:&str) {
    println!("\n=== {} Structure ===", name);
    println!("size: {} Bytes", size_of_val(instance));
    println!("align based on: {} Bytes", align_of_val(instance));

}
fn main() {
    let rust_struct=RustStyle { a:1, b:2, c:3, d:4.0 };
    let c_struct=CStyle { a:1, b:2, c:3, d:4.0 };
    print_struct(&rust_struct, "Rust optimized layout");
    print_struct(&c_struct, "C layout");
}
```

Default alignment rule:

- Same as the core principle of "natural alignment" in C/C++(member addresses must be integer multiples of their type size)
- Key difference: The Rust compiler optimizes field order by default (repr (Rust) mode), while C/C++maintains declaration order.

- You can **force the use of C language compatible layouts** through **# [repr (C)]**

Design considerations:

- Memory optimization: Reduce padding bytes through field rearrangement
- Performance optimization: Place frequently accessed fields on the same cache line
- Hardware adaptation: supports special alignment requirements for different architectures

```
=== Rust optimized layout Structure ===
size: 16 Bytes
align based on: 8 Bytes

=== C layout Structure ===
size: 24 Bytes
align based on: 8 Bytes
```

Memory Security(1)- one owner(1)

```rust
fn main() {
    let s1=String::from("hello");
    println!("addr of s1 in stack: {:p}", &s1);
    println!("addr of String in heap : {:p}", s1.as_ptr());

    let s2=s1;
    println!("addr of s2 in stack: {:p}", &s2);
    println!("addr of String in heap : {:p}", s2.as_ptr());

    // println!("addr of s1 in stack: {:p}", &s1);
    // println!("addr of String in heap : {:p}", s1.as_ptr());

    let s3=s2.clone();
    println!("addr of s3 in stack: {:p}", &s3);
    println!("addr of String in heap : {:p}", s3.as_ptr());

    println!("addr of s2 in stack: {:p}", &s2);
    println!("addr of String in heap : {:p}", s2.as_ptr());
}
```

ownership is moved from s1 to s2

create new place in both stack and heap

```
error[E0382]: borrow of moved value: `s1`
  --> lab14_demo3_own_move_ref.rs:10:47
   |
2  |     let s1 = String::from("hello");
   |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
...
5  |     let s2 = s1;
   |              -- value moved here
...
10 |     println!("addr of String in heap : {:p}", s1.as_ptr());
   |                                               ^^ value borrowed here after move
   |
   = note: borrow occurs due to deref coercion to `str`
help: consider cloning the value if the performance cost is acceptable
   |
5  |     let s2 = s1.clone();
   |                 ++++++++

error: aborting due to 1 previous error
```

```
addr of s1 in stack: 0x7ffe27426198
addr of String in heap : 0x56103287da40
addr of s2 in stack: 0x7ffe27426270
addr of String in heap : 0x56103287da40
addr of s3 in stack: 0x7ffe27426348
addr of String in heap : 0x56103287da60
addr of s2 in stack: 0x7ffe27426270
addr of String in heap : 0x56103287da40
```

Memory Security(1)- one owner(2)

```rust
fn main() {
    let s1=String::from("hello");
    println!("addr of s1 in stack: {:p}", &s1);
    println!("addr of String in heap : {:p}", s1.as_ptr());

    process_string(s1);
    //println!("addr of s1 in stack: {:p}", &s1);
    //println!("addr of String in heap : {:p}", s1.as_ptr());
}

fn process_string(s:String) {
    println!("--process_string--");
    println!("addr of s in stack: {:p}", &s);
    println!("addr of String in heap : {:p}", s.as_ptr());
}
```

ownership moved from s1 to s

```
  --> lab14_demo3_own_move_ref2.rs:12:22
   |
12 | fn process_string(s: String) {      // s取得所有权
   |    --------------    ^^^^^^ this parameter takes ownership of the value
   |    |
   |    in this function
   = note: borrow occurs due to deref coercion to `str`
help: consider cloning the value if the performance cost is acceptable
   |
6  |     process_string(s1.clone());
   |                       ++++++++

error: aborting due to 1 previous error
```

```
addr of s1 in stack: 0x7ffcd911f0a8
addr of String in heap : 0x557504f06a40
--process_string--
addr of s in stack: 0x7ffcd911f180
addr of String in heap : 0x557504f06a40
```

Memory Security(2)

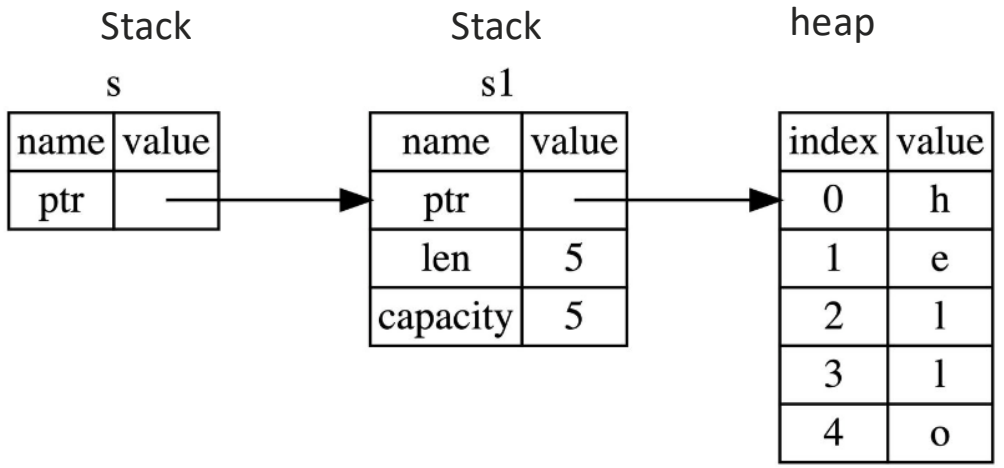
- Automatically released upon exiting the lifecycle, there is no memory leaks.

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/rust_demo$ cat lab14_demo3_own_move_ref2.rs
fn main() {
    let s1 = String::from("hello");
    println!("addr of s1 in stack: {:p}", &s1);
    println!("addr of String in heap : {:p}", s1.as_ptr());

    process_string(s1);
    //println!("addr of s1 in stack: {:p}", &s1);
    //println!("addr of String in heap : {:p}", s1.as_ptr());

}

fn process_string(s: String) {
    println!("--process_string--");
    println!("addr of s in stack: {:p}", &s);
    println!("addr of String in heap : {:p}", s.as_ptr());
}  ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/rust_demo$ rustc lab14_demo3_own_move_ref2.rs
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/rust_demo$ valgrind  ./lab14_demo3_own_move_ref2
==29193== Memcheck, a memory error detector
==29193== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29193== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==29193== Command: ./lab14_demo3_own_move_ref2
==29193==
addr of s1 in stack: 0x1ffefffa68
addr of String in heap : 0x4a958f0
--process_string--
addr of s in stack: 0x1ffefffb40
addr of String in heap : 0x4a958f0
==29193==
==29193== HEAP SUMMARY:
==29193==     in use at exit: 0 bytes in 0 blocks
==29193==   total heap usage: 7 allocs, 7 frees, 2,917 bytes allocated
==29193==
==29193== All heap blocks were freed -- no leaks are possible
==29193==
==29193== For lists of detected and suppressed errors, rerun with: -s
==29193== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/rust_demo$
```

Memory Security(3-1)

- **reference** type: use&T to represent the reference type of type T. Reference type is a data type that indicates that the value it holds is a reference.
  - &i32 represents the reference type of i32;
  - &&i32 represents the reference type referenced by i32;
  - **Default as immutable reference**;
  - Use symbol * to dereference references

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);  // &s1 is the reference of s1
    println!("The length of '{}' is {}.", s1, len);
}

// the type of parameter is reference
fn calculate_length(s: &String) -> usize {
    s.len()
}
```



Memory Security(3-2)

Rulers about reference:

- mutable references can only exist once at a time.
- mutable and immutable references cannot exist simultaneously.

```
//ERROR DEMO
let mut s = String::from("hello");
// mutable references can only exist once at a time;

let r1 = &mut s;
let r2 = &mut s;
println!("{}, {}", r1, r2);
```

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> lab14_demo3_ref3.rs:6:14
  |
5 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
6 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0499`.
```

```
//ERROR DEMO
let mut s = String::from("hello");
// mutable and immutable references cannot exist
simultaneously
let r1 = &s;
let r2 = &s;
let r3 = &mut s;
println!("{}, {}, and {}", r1, r2, r3);
```

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> lab14_demo3_ref4.rs:7:10
  |
5 | let r1 = &s;
  |          -- immutable borrow occurs here
6 | let r2 = &s;
7 | let r3 = &mut s;
  |          ^^^^^^ mutable borrow occurs here
8 | println!("{}, {}, and {}", r1, r2, r3);
  |                            -- immutable borrow later used here

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0502`.
```

Memory Security(4) Smart pointers

Smart pointers are data structures in Rust that have pointer semantics and ensure the correctness and security of values throughout their lifecycle. Smart pointers are typically implemented using structures.

Type	Ownership Mechanism	Thread Safe	Primary Use Cases	Internal Mutability	Runtime Checks	Clone Behavior
Box<T>	**Single ownership**	Yes	Heap allocation, recursive types, trait objects	No	None	Deep copy (full value)
Rc<T>	**Reference-counted shared**	No	Single-thread shared ownership	No	Reference counting	Increments counter
Arc<T>	Atomic reference-counted	Yes	Thread-safe shared ownership	No	Atomic operations	Atomic counter increment
RefCell<T>	Single ownership	No	Interior mutability (single-thread)	Yes	Borrow checking	Panics on invalid borrows
Cell<T>	Single ownership	No	Copy-type interior mutability	Yes (swap-based)	None	Copies value
Mutex<T>	Lock-guarded access	Yes	Thread-safe mutable access	Yes	Lock acquisition	Guard-based access
RwLock<T>	Lock-guarded access	Yes	Thread-safe multiple readers or single writer	Yes	Lock acquisition	Guard-based access

Memory Security(4) Smart pointers : Box

Box allows assigning a value to the heap and then keeping a smart pointer on the stack pointing to the data on the heap; When there is a type of unknown size at compile time and you want to use its value in a context that requires an exact size.

```
let b = Box::new(5);   // 5 is stored in heap
println!("b = {}", b);
```

Box smart pointers can solve recursive typing problems:
Recursive type: **can have another value of the same type as a part of it**.
This creates a problem because Rust needs to know how much space types occupy at compile time. The value nesting of recursive types can theoretically continue infinitely, so Rust does not know how much space recursive types require. For example, the following example:

```
// ERROR DEMO
enum List {
   Cons(i32, List),
   Nil,
}

use List::{Cons, Nil};

fn main() {
   let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

```
// OK DEMO
enum List {
   Cons(i32, Box<List>),
   Nil,
}

use List::{Cons, Nil};

fn main() {
   let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3,Box::new(Nil))))));
}
```

```
   Compiling refff v0.1.0 (D:\Rust\temp\refff)
error[E0072]: recursive type `List` has infinite size
 --> src\main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List` representable
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +

For more information about this error, try `rustc --explain E0072`.
```

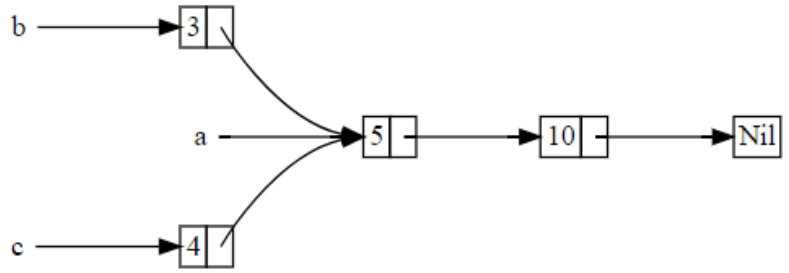
Memory Security(4) Smart pointers : RC(1)

- Rc is a reference count pointer, which records the number of references to a value to determine if it is still in use; If a value has zero references, it means there are no valid references and it can be cleaned up; Cannot be used in multithreading.

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let _b = Cons(3, Box::new(a));
    let _c = Cons(4, Box::new(a));
}
```

```
   Compiling refff v0.1.0 (D:\Rust\temp\refff)
error[E0382]: use of moved value: `a`
 --> src\main.rs:9:31
  |
7 |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
  |         - move occurs because `a` has type `List`, which does not implement the `Copy` trait
8 |     let _b = Cons(3, Box::new(a));
  |                           - value moved here
9 |     let _c = Cons(4, Box::new(a));
  |                           ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
```

Rc smart pointers can solve the problem of a resource having multiple owners. For example:




```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}
use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let _b = Cons(3, Rc::clone(&a));
    let _c = Cons(4, Rc::clone(&a));
}
```

RC(2)

Using Weak to break cycle reference invoked by Rc.

```rust
use std::rc::Rc;
use std::cell::RefCell;

struct Node {
    name:String,
    partner:RefCell<Option<Rc<Node>>>,
}
impl Node {
    fn new(name:&str) ->Rc<Self> {
        Rc::new(Node {
            name:name.to_string(),
            partner:RefCell::new(None),
        })
    }
}
fn main() {

    let alice=Node::new("Alice");
    let bob=Node::new("Bob");
    // Establish cycle references
    *alice.partner.borrow_mut() =Some(Rc::clone(&bob));
    *bob.partner.borrow_mut() =Some(Rc::clone(&alice));
}
```

```rust
use std::rc::{Rc, Weak};
use std::cell::RefCell;

struct Node {
    name:String,
    // Break the cycle with Weak
    partner:RefCell<Option<Weak<Node>>>,
}

impl Node {
    fnnew(name:&str) ->Rc<Self> {
        Rc::new(Node {
            name:name.to_string(),
            partner:RefCell::new(None),
        })
    }
}
fn main() {
    let alice=Node::new("Alice");
    let bob=Node::new("Bob");

    // Alice(Bob) holds Bob(Alice)'s Wear reference
    *alice.partner.borrow_mut() =Some(Rc::downgrade(&bob));
    *bob.partner.borrow_mut() =Some(Rc::downgrade(&alice));

    // Strong references can be obtained through upgrade()
    if let Some(partner) =alice.partner.borrow().as_ref() {
        if let Some(p) =partner.upgrade() {
            println!("Alice's partner: {}", p.name);
        }
    }
}
```

Practices

- 1. Please complete the code on page 5(as shown in the code on the right), print out the addresses of each element(a,b,c,d) in the two structures(RustStyle, CStyle).
- Compare the differences between RUST and C when storing structure data.

```rust
use std::mem::{size_of_val, align_of_val};
struct RustStyle {
    a:u8,   // 1 BYTE
    b:u32,  // 4 BYTES
    c:u16,  // 2 BYTES
    d:f64,  // 8 BYTES
}

#[repr(C)]
struct CStyle {
    a:u8,
    b:u32,
    c:u16,
    d:f64,
}
fn print_struct<T>(instance:&T, name:&str) {
    println!("\n=== {} Structure ===", name);
    println!("size: {} Bytes", size_of_val(instance));
    println!("align based on: {} Bytes", align_of_val(instance));

}
fn main() {
    let rust_struct=RustStyle { a:1, b:2, c:3, d:4.0 };
    let c_struct=CStyle { a:1, b:2, c:3, d:4.0 };
    print_struct(&rust_struct, "Rust optimized layout");
    print_struct(&c_struct, "C layout");
}
```

Practices

- 2. If the following code will encounter an error during execution? If not, will the program cause a memory leak during runtime?
- Please rewrite this code in C++to require a pointer variable to be defined in the subfunction, which points to a heap space requested within the function and is returned within the function. The main function defines a pointer variable and assigns a value to it.
- Compare C/C++and Rust, use tools to determine which implementation will cause memory leakage, compare the differences between the two, and if there is a memory leakage, modify the code to solve the problem.

```rust
fn main() {
    let s =
gives_ownership();
    println!("{}", s);
}

fn gives_ownership() -> String {
    let some_string = String::from("Rust");
    some_string
}
```

Practices

- 3. The following code poses a memory risk. Please identify the issue and use a smart pointer solution to solve the memory problem

```
struct DangerousContainer {
    data: *mut i32,
}

impl DangerousContainer {
    fn new(value: i32) -> Self {
        let ptr = Box::into_raw(Box::new(value));
        DangerousContainer { data: ptr }
    }

    fn create_dangling() -> *mut i32 {
        let local_data = 42;
        &local_data as *const i32 as *mut i32
    }

    unsafe fn get(&self) -> i32 {
        *self.data
    }
}

fn main() {

    let container1 = DangerousContainer::new(10);
    let dangling_ptr = DangerousContainer::create_dangling();
    unsafe {
        println!("Dangling value: {}", *dangling_ptr);
    }

    let ptr = Box::into_raw(Box::new(20));
    let container2 = DangerousContainer { data: ptr };
    let container3 = DangerousContainer { data: ptr };

    // drop(container2);
    // drop(container3);
}
```