



Lecture 7

Memory Safety in Rust & C

Introduction

There are two program properties:

Memory safety

- Memory pointers point to valid memory.
- A memory unsafe program may crash.

Memory Containment

- Memory does not leak.
- A leaky program may eventually run out of memory.

Introduction

- In CG languages (like Java and Python) memory safety is guaranteed within the language runtime.
- In non-GC languages, like C, C++ and Rust, these memory properties must either be guaranteed by the compiler via static analysis (Rust's borrow checker),
- or they must be carefully managed by the programmer at runtime (malloc/free, new/delete).

Example



- The following example provides an implementation of a vector library (or resizable array) specialized for integers.
- It contains at least 7 bugs relating to the properties of memory safety and containment.

Please note: this is written in C and I don't really write C.

Example



```
typedef struct {
    int* values;    int  len;    int  capacity;
}Vec;
Vec* vec_new() {
    Vec vec;    vec.values = NULL;    vec.len = 0;
    vec.capacity = 0;    return &vec;
}
void vec_insert(Vec* vec, int n) {
    if (vec->len == vec->capacity) {
        int new_capacity = vec->capacity * 2;
        int* new_values = (int*) malloc(new_capacity);
        assert(new_values != NULL);
        for (int i = 0; i < vec->len; ++i) {
            new_values[i] = vec->values[i];    }
        vec->values = new_values;
        vec->capacity = new_capacity;    }
    vec->values[vec->len] = n;
    ++vec->len; }
```

Example



```
void vec_deallocate(Vec* vec) {
    free(vec);
    free(vec->values);
}

void main() {
    Vec* vec = vec_new();
    vec_insert(vec, 107);

    int* n = &vec->values[0];
    vec_insert(vec, 110);
    printf("%d\n", *n);

    free(vec->values);
    vec_deallocate(vec);
}
```

Issues



1. *vec_new*: This is a dangling pointer. the stack frame is deallocated when the function returns, so any subsequent use of the pointer is invalid.
2. *vec_new*: initial capacity is 0. When capacity doubles, $2 * 0 = 0$.
3. *vec_insert*: incorrect call to malloc. We need to `malloc(sizeof(int) * new_capacity)`.
4. *vec_insert*: missing free on resize.
5. *vec_deallocate*: incorrect ordering on the free statements.
6. *main*: double free of `vec->values`.
7. *main*: invalid initialization of `n`.

Dangling pointer

- A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.
- Here are two examples where Pointer acts as dangling pointer

1. De-allocation of memory

```
// Deallocating a memory pointed by ptr causes
// dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int *ptr = (int *)malloc(sizeof(int));
    // After free call, ptr becomes a dangling pointer
    free(ptr);
    // No more a dangling pointer
    ptr = NULL;
}
```


Dangling pointer

2. Function Call


```
// The pointer pointing to local variable becomes
// dangling when local variable is not static.
#include<stdio.h>
int *fun()
{
    // x is local variable and goes out of
    // scope after an execution of fun() is over.
    int x = 5;
    return &x;
}
int main()
{
    int *p = fun();
    fflush(stdin);
    // p points to something which is not valid anymore
    printf("%d", *p);
    return 0;
}
```

Dangling pointers in Rust???


```
fn main() {  
    let string = foo();  
}  
  
fn foo() -> &string {  
    let string = String::from("Hello, World!");  
    println!("{}", string);  
    &string  
}
```

Looks possible – Are we in Trouble????

And the compiler says.....



```
error[E0106]: missing lifetime specifier
  --> src/main.rs:110:13
   |
5  | fn foo() -> &String {
   |             ^ expected lifetime parameter
   = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
```



Hack around with the code as much as you like

The compiler will always say NO, no, and NO!

You cannot write a dangling pointer in Rust!

Issues



1. *vec_new*: This is a dangling pointer. the stack frame is deallocated when the function returns, so any subsequent use of the pointer is invalid.
2. *vec_new*: initial capacity is 0. When capacity doubles, $2 * 0 = 0$.
3. *vec_insert*: incorrect call to malloc. We need to `malloc(sizeof(int) * new_capacity)`.
4. *vec_insert*: missing free on resize.
5. *vec_deallocate*: incorrect ordering on the free statements.
6. *main*: double free of `vec->values`.
7. *main*: invalid initialization of `n`.

Example

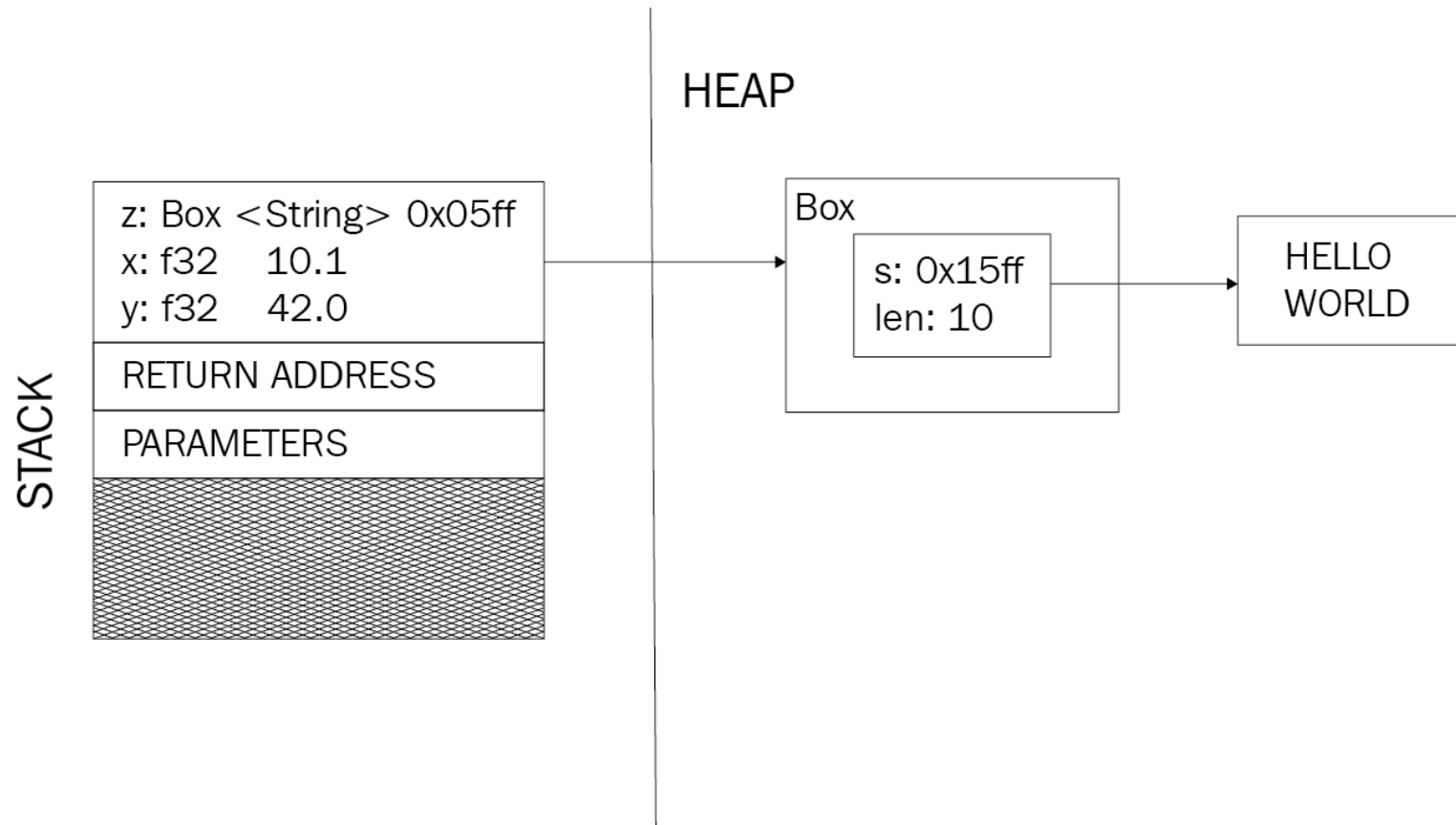


- If we naively translate the previous C code:

```
struct Vec2 {  
    values: Box<[isize]>,  
    len: usize,  
    capacity: usize  
}  
  
impl Vec2 {  
    fn new() -> &Vec2 {  
        let v = Vec2 {  
            values: Box::new([]),  
            len: 0,  
            capacity: 0  
        };  
        return &v;  
    }  
}  
  
fn main () {}
```

What on earth is a Box?

What on earth is a Box?



Example



- Here, if we naively translate the previous C code, this fails to compile:

```
error[E0106]: missing lifetime specifier
--> v.rs:8:17
   |
8  |         fn new() -> &Vec2 {
   |                       ^ expected lifetime parameter
   |
   = help: this function's return type contains a borrowed
value, but there is no value for it to be borrowed from
   = help: consider giving it a 'static lifetime
```

Example



- Rust can identify the dangling stack pointer issue.
- It analyzes the type signature of the function.
- Since the function takes no references as input, it's impossible to return a reference as output.
- **How can we fix this error?**

```
impl Vec2 {  
    fn new() -> Vec2 {  
        let v = Vec2 {  
            values: Box::new([0]),  
            len: 0,  
            capacity: 1  
        };  
        return v;  
    }  
}
```


Example



- Note that the capacity issue is a logic error that must be identified by the programmer.
- Next, we implement the insert method:

Example



```
fn vec_insert(&mut self, n: isize) {
    if self.len == self.capacity {
        let new_capacity = self.capacity * 2;
        let mut new_value = unsafe {
            let ptr = Heap::default()
                .alloc(Layout::array::<isize>
                    (new_capacity).unwrap())
                .unwrap() as *mut isize;
            Box::from_raw(slice::from_raw_parts_mut(ptr,
                new_capacity))
        };
        for i in 0..self.length {
            new_value[i] = self.values[i];
        }
        self.values = new_value;
        self.capacity = new_capacity;
    }
    self.values[self.len] = n;
    self.len += 1;
}
```

Example



- This method compiles and works correctly.
- It does not require an explicit free.
- Rust will automatically deallocate the old value of **values** when it is reassigned
- The programmer does not have to free allocated memory, this eliminates both the associated memory leaks as well as double frees.

Memory Allocation in Rust

- All memory allocations happen either implicitly on the stack by declaring a value
- Or explicitly on the heap when using Box or any pointer type derived from it

```
struct Point { x: f32, y: f32 }  
let p: Box<Point> = Box::new(Point{ x: 0.1, y: 0.2 });
```

- Rust determines the size of Point and does the appropriate malloc(sizeof(Point)).

Memory Deallocation in Rust

- We do not have to implement the `vec_deallocate` function.
- Rust automatically generates the appropriate destructors.

Initialization Invalidation

- Note that the following main function:

```
fn main() {  
    let mut vec: Vec2 = Vec2::new();  
    vec.insert(107);  
    let n: &isize = &vec.data[0];  
    vec.insert(110);  
    println!("{}", n);  
}
```

- Fails with the following error:

```
error[E0502]: cannot borrow `vec` as mutable because  
`vec.data[..]` is also borrowed as immutable  
--> v.rs:50:5  
    |  
49 |         let n: &isize = &vec.data[0];  
    |                               ----- immutable borrow  
occurs here
```

To sum up..

- The guarantees provided by Rust helped us fix memory-related errors in C.
- No matter how large your code base, Rust enforces these guarantees everywhere.
- All this, of course, comes at the price of fighting with Rust's borrow checker.

Box<T>

- Box<T> is a smart pointer that points to the data which is allocated on the heap of type T.
- Box<T> is an owned pointer.
- Boxes do not have a performance overhead, other than storing the data on the heap.
- When the Box goes out of the scope, then the destructor is called to release the memory.

```
fn main()
{
    let a = Box :: new(1);
    print!("value of a is : {}",a);
}
```

Output: value of a is : 1

Box<T>

```
fn main()
{
    let a = Box :: new(1);
    print!("value of a is : {}",a);
}
```

Output: value of a is : 1

Stack



Heap



Cons List “Construct function”

- Cons list is a data structure which is used to construct a new pair from the two arguments, and this pair is known as a List.
- Cons is a variant of the List enum. It's saying there are two possible cases of a linked list - an empty list or a head consisting of a u32 and a ~List

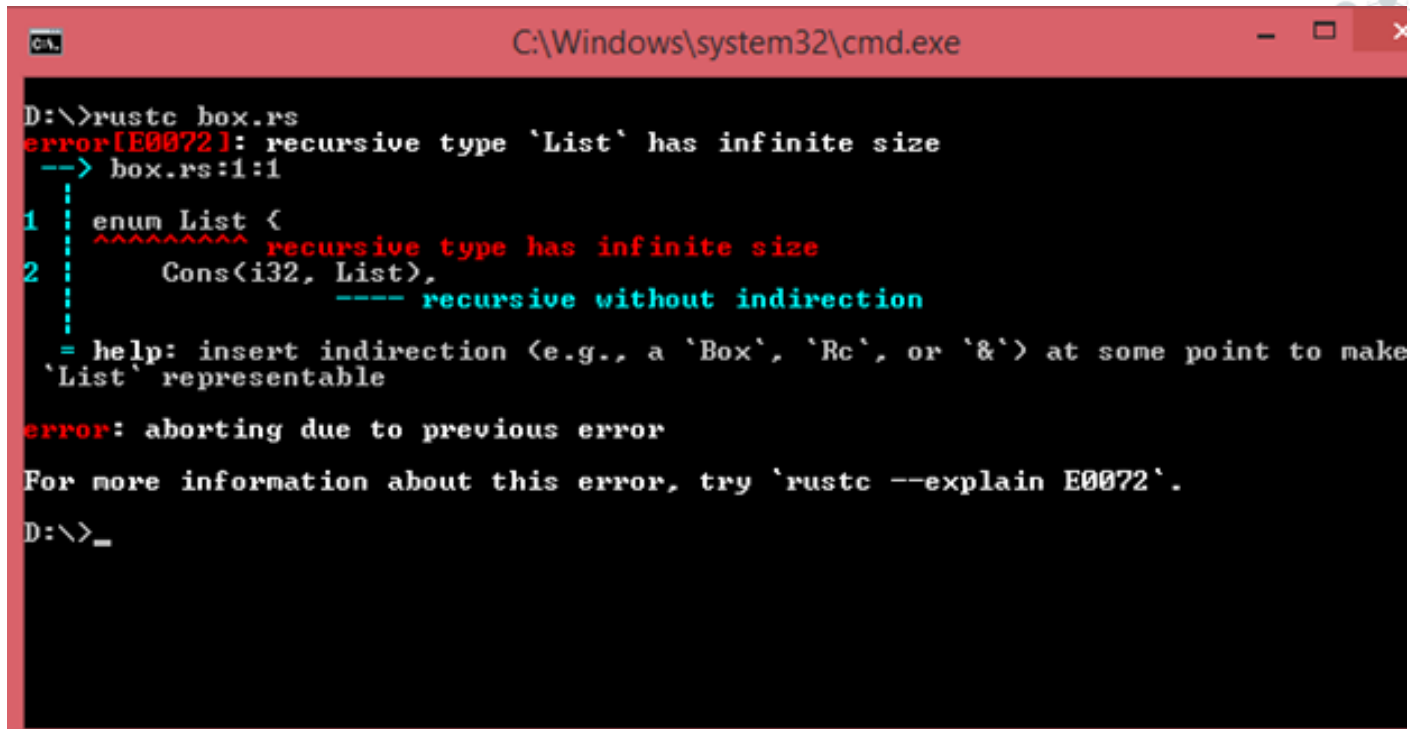
```
enum List
{
    cons(i32, List),
    Nil,
}
```

Cons List “Construct function”

```
enum List {  
    Cons(i32, List),  
    Nil,  
}  
use List::{Cons, Nil};  
fn main()  
{  
    let list = List::Cons(1, Cons(2, Cons(3, Nil))) ;  
    for i in list.iter()  
    {  
        print!("{}", i);  
    }  
}
```

Cons List “Construct function”

- Output:



```
C:\Windows\system32\cmd.exe

D:\>rustc box.rs
error[E0072]: recursive type `List` has infinite size
--> box.rs:1:1
1 | enum List {
  |          ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ---- recursive without indirection
   = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to make
   `List` representable
error: aborting due to previous error

For more information about this error, try `rustc --explain E0072`.
D:\>_
```

- Rust is not able to find out how much space is required to store the List value. The problem of an infinite size can be overcome by using the Box<T>.

Using Box<T> to get the size of a recursive type

```
#[derive(Debug)]
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))))));

    print!("{:?}", list);
}
Output:
Cons(1, Cons(2, Cons(3, Nil)))
```

How do you know if a Rust variable is allocated on the heap?

- If a variable has type `Box<T>`, then it's a pointer to some memory on the heap.
- Are there other types that are always on the heap?
 1. `Vec<T>` (an array on the heap)
 2. `String` (a string on the heap)
 3. `Box<T>` (just a pointer)



Just a pointer, Rust does not really understand what it points to.