Rust Crash Course

Part2

Agenda

- Memory regions
- References vs raw pointers
- usize

- Slices
- Strings

- Printing structs
- Options

Memory regions

- Data memory
  - For data that is fixed in size and static (i.e. always available through life of program).
  - Compilers make lots of optimizations with this kind of data, and they are generally considered very fast to use since locations are known and fixed.
- Stack memory
  - For data that is declared as variables within a function.
  - The location of this memory never changes for the duration of a function call; because of this compilers can optimize code so stack data is very fast to access.
- Heap memory
  - For dynamically created data.
  - Data in this region may be added, moved, removed, resized, etc.
  - Because of its dynamic nature it's generally considered slower to use

Example: data & stack memory

- Instantiating a struct
- The string literal `"Rust"` is read only, placed in data memory region
- The function call `String::from` creates a struct String that is placed in the stack

```
struct Person {
   name: String,
}

fn main() {
  // data is on stack
  let ferris = Person {
    // String struct is also on stack,
    // but holds a reference to data on heap
    name: String::from("Rust"),
  };
}
```

Example: heap memory (2)

- The same example, this time with an own new function

```
struct Person {
   name: String,
}

impl Person {

  // Function for creating a new Person on the Stack
  fn new(name: String) -> Person {
    Person { name }
  }
}

// fn main(), see previous slide
```

Example: heap memory

- Box is a data structure that allows us to move our data from the stack to the heap.
- Box is a struct known as a `smart pointer` (see later for more details) holds the pointer to our data on the heap.

```rust
struct Person {
   name: String,
}

fn main() {
  // instantiate Person and put it in a Box
  let person_boxed = Box::new( Person { name: String::from("BS-E"), } );

  println!("Name: {}", person_boxed.name);
}
```

Example: heap memory (2)

- The same example, this time with an own new function

```
struct Person {
   name: String,
}

impl Person {
   // Function for creating a new boxed Person
   fn new(name: String) -> Box<Person> {
      Box::new(Person { name })
   }
}
```

References

- A reference is fundamentally just a number that is the start position of some bytes in memory.
- Rust will validate the lifetime of references doesn't last longer than what it refers to (otherwise we'd get an error when we used it!).
- Lifetimes see later

Raw Pointers

- References can be converted into a more primitive type called a `raw pointer`.

- Much like a number, it can be copied and moved around with little restriction. Rust makes no assurances of the validity of the memory location it points to.

- Two kinds of raw pointers exist:
  - `*const T` - A raw pointer to data of type T that should never change.
  - `*mut T` - A raw pointer to data of type T that can change.

Raw Pointers (2)

hhu
Heinrich Heine
Universität Düsseldorf

- Raw pointers can be converted to and from numbers (e.g. usize).
- Raw pointers can access data with unsafe code (more on this later).

```
fn main() {
  let a = 42;
  let memory_location = &a as *const i32 as usize;
  println!("Data is here {}", memory_location);
}
```

References vs Raw Pointers

- Rust reference:
  - Similar to a pointer in C in terms of usage
  - But with much more compile time restrictions on
    how it can be stored and moved around to other functions.

- Rust raw pointer:
  - Also similar to a pointer in C
  - Represents a number that can be copied or passed around,
    and even turned into numerical types
    where it can be modified as a number to do pointer math.

`u64` vs `usize`

- As the documentation states usize is pointer-sized, thus its actual size depends on the architecture you are compiling your program for.
- As an example, on a 32 bit x86 computer, usize = u32 while on x86_64 computers, usize = u64.

When do you use `usize`

- Variables for indexing into collections
  - Index will have the correct size

```
let my_array = [10, 20, 30, 40, 50];
let index: usize = 2;
```

- Memory addresses and sizes
  - When working with raw pointers, `usize` is often used to represent memory addresses and memory sizes.

```
let ptr: *mut i32 = 0x1234 as *mut i32;
let size: usize = 10;
```

- Lengths and size calculations:
  - When working with memory blocks or data structures whose size can vary at runtime, `usize` is used to represent length and size information.

Example: memory access using raw pointer

- In this example we want something similar like in C writing into raw memory using a struct pointer
- `ListNode` defines meta data for our heap management in hhuTOS

```
struct ListNode {
   // size of the memory block
   size: usize,

   // &'static mut type semantically describes an owned object behind
   // a pointer. Basically, it's a Box without a destructor
   next: Option<&'static mut ListNode>,
}

impl ListNode {

   // Create new ListMode on Stack (must be 'const')
   const fn new(size: usize) -> Self {
      ListNode { size, next: None }
   }
```

Example: memory access using raw pointer (2)

Heinrich Heine Universität Düsseldorf

- We create a `ListNode` on the stack, initialize its data as needed
- Then we use a raw pointer for writing its content at a given address

```
impl LinkedListAllocator {

   unsafe fn add_free_block(&mut self, addr: usize, size: usize) {

      // create a new ListNode (on stack)
      let mut node = ListNode::new(size);

      // set next ptr of new ListNode to existing 1st block
      // ‘take’ transfers ownership
      node.next = self.head.next.take();

      // create a raw pointer of type ListNode at given address ‘addr’
      let node_ptr = addr as *mut ListNode;

      // write content of ‘node’ in raw memory using raw pointer
      node_ptr.write(node);
```

Slices

- Slice: `&[T]` represents a view into a contiguous sequence of elements of type T
- Slices are lightweight abstractions that provide a safe and efficient way to work with a portion of a collection without needing to copy the data.

```
fn main() {
  let numbers = [1, 2, 3, 4, 5];
  // Take a slice of the array
  let slice = &numbers[1..4]; // Slice from [1,..4(

  // Iterate over the slice and print each element
  for &num in slice {
    println!("{}", num);
  }
}
```

Mutable Slices

- Same es before but now elements are mutable.

```
fn main() {
  let mut numbers = [1, 2, 3, 4, 5];

  // Take a slice of the array
  let mut_slice = &mut mutnumbers[1..4]; // Slice from [1,..4(

  mut_slice[0] = 10;
}
```

Thin & Fat Pointers

Heinrich Heine
Universität Düsseldorf

- **Thin pointer**: are used for references to sized types.
  - These are types whose size is known at compile time, such as integers, structs, arrays, etc.
  - They consist of a simple memory address pointing to the data on the heap or stack.
  - Example: &T

- **Fat pointer**: are used for references to unsized types
  - These are types whose size is not known at compile time
  - They consist of both a memory address and metadata about the referenced data
  - Example: `&[T]` for a slice (see later)

String

- String: is a growable, mutable, owned string type
  - Lives in the heap
  - Is mutable and can alter its size and contents
- Variables of type String are fat pointer, 3 x 8 byte
  - Pointer to actual data on the heap, it points to the first character
  - Length of the string (# of characters)
  - Capacity of the string on the heap

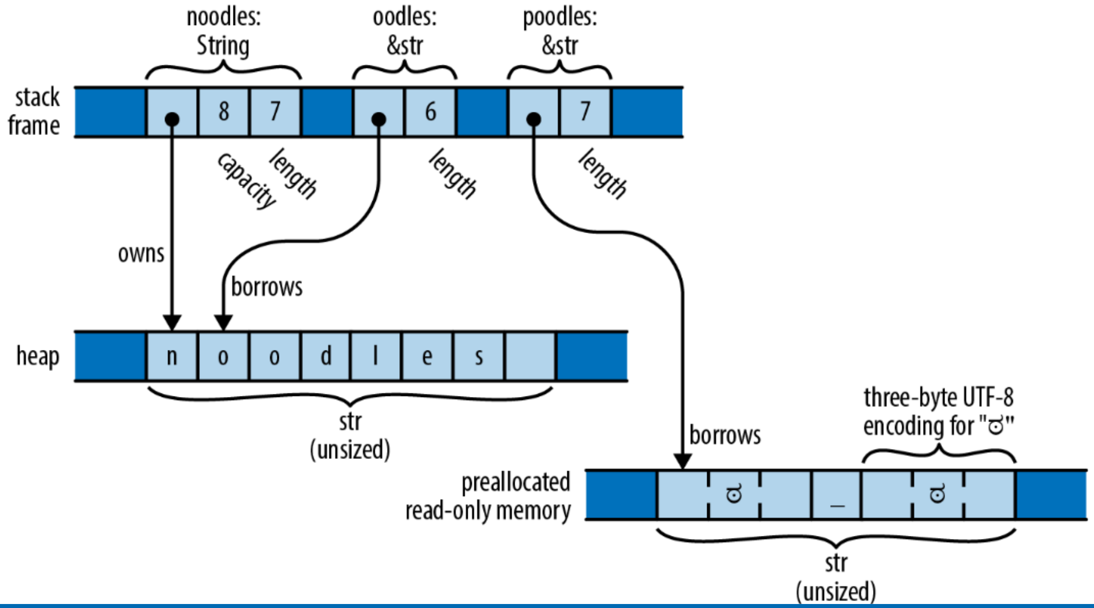
```rust
// Create a new mutable empty String in the heap
let mut s = String::new();

// Push characters onto the String
s.push('h');
s.push_str("ello");
```

&str

- &str: This denotes a reference to a string slice.
  - &: Indicates that it's a reference, meaning it doesn't own the data it points to. Instead, it borrows the data from another location.
  - str: Indicates that the data being referenced is a string slice.
- Immutable
- Lives in heap or ‘static memory
- Non-owned type -> memory is not freed if variable goes out of scope

String vs &str

```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "ಠ_ಠ";    // this is string literal
```



Aus: https://stackoverflow.com/questions/24158114/what-are-the-differences-between-rusts-string-and-str

Printing structs

Heinrich Heine Universität Düsseldorf

- Derive `trait Debug`

```
#[derive(Debug)]
struct Vector {
    x: i64,
    y: i64,
}
```

- Pretty-printed debug output: `{:#?}` or `{:?}`

```
fn main() {
  let v1 = Vector { x: 42, y: 41 };
  println!("v1 = {:?}", v1);
}
```

```
$ ./test
v1 = Vector { x: 42, y: 41 }
```

```
fn main() {
  let v1 = Vector { x: 42, y: 41 };
  println!("v1 = {:#?}", v1);
}
```

```
$ ./test
v1 = Vector {
    x: 42,
    y: 41,
}
```

Implementing `trait Debug`

```
struct Vector {
    x: i64,
    y: i64,
}

impl fmt::Debug for Vector {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "Vector [0x{:x}, 0x{:x}]", self.x, self.y)
  }
}

fn main() {
  let v1 = Vector { x: 42, y: 41 };
  println!("v1 = {:?}", v1);
}
```

```
$ ./test
v1 = Vector [0x2a, 0x29]
```

Generics

- Generic types are used in the representation of nullable values, error handling, …
- Rust generally can infer the final type by looking at our instantiation, but if it needs help you can always be explicit using the ::<T> operator

```
struct BagOfHolding<T> {
  item: T,
}

fn main() {
  let i32_bag  = BagOfHolding::<i32>  { item: 42 };
  let bool_bag = BagOfHolding::<bool> { item: true };

  println!("{} {}", i32_bag.item, bool_bag.item);
}
```

Options

- Pointers are never null! What if you actually *want* something to be null?
- Use an `Option<T>`! Here's the definition of `Option`, from the standard library:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

- Two possible cases: the option is either `None`, or it is `Some`.
- If an Option is `Some`, the value in the `Some` variant will always be a valid value of type `T`.

Options: example

```rust
// This type annotation is not necessary.
let x: Option<i32> = Some(4);
assert!(x.is_some());
let y = x.unwrap(); // get the value out of Option
assert_eq!(y, 4);

// This type annotation IS necessary!
let z: Option<i32> = None;
assert!(z.is_none());

let w = Some(String::from("hello"));
match w {
   Some(s) => println!("{} world!", s),
   None => panic!("didn't expect to get here"),
}
```

Credit

Heinrich Heine Universität Düsseldorf

- If not mentioned, content on these slides is from following sources
- Slides from Rohan Kumar, Rahul Kumar, and Edward Zeng, used in the course CS 162: Operating Systems and Systems Programming,
  Prof. John Kubiatowicz at Berkely University, USA
- And from https://tourofrust.com