Ownership, Pointers, and Memory

Advanced Systems Programming (M) 2018-2019 – Laboratory Exercise 4
Dr Colin Perkins, School of Computing Science, University of Glasgow

1 Introduction

The Advanced Systems Programming (M) course uses the Rust programming language (`https://rust-lang.org/`) to illustrate several advanced topics in systems programming. You're expected to learn the basics of programming in Rust as part of this course. This exercise reviews the basics of ownership, pointers, and memory management in Rust. **It is a formative exercise and is not assessed.**

2 Memory Safety, Ownership, References, and Lifetimes

The sample program shown in Figure 1, at the end of this document, provides an example of unsafe memory usage in the C programming language. It contains at least seven memory related bugs. Review the code to find all such problems making sure you understand the cause of each of the bugs.

One of the goals of the Rust programming language is to make such unsafe memory usage impossible, while not compromising performance. It does this by tracking the ownership and lifetime of data, and having the compiler automatically insert calls to free memory when data goes out of scope. Read the sections of the online Rust book explaining ownership, lifetimes, and smart pointers:

- `https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html`
- `https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html`
- `https://doc.rust-lang.org/book/ch15-00-smart-pointers.html`

The following Rust programs demonstrate key memory safety and ownership features of the language. Review them carefully.

Example 1: The following Rust program does not compile. Explain why not, and then fix the program so that it compiles and runs correctly. Consider when is the memory allocated to the vector, `v`, and when is it freed? What is the meaning of the `'a` specifiers in the definition of the `Person` structure?

```
#[derive(Debug)]
struct Person<'a> {
    name : &'a str,
    role : &'a str
}

fn print_employees(employees: Vec<Person>) {
    for e in &employees {
      println!("{:?}", e);
    }
}

fn main() {
    let mut v = Vec::new();

    v.push(Person{name : "Alice", role : "Manager"});
    v.push(Person{name : "Bob"  , role : "Sales"});
    v.push(Person{name : "Carol", role : "Programmer"});

    print_employees(v);

    println!("v.len() = {}", v.len());
}
```

Example 2: The following Rust program also does not compile. Explain why. What memory safety problem is it preventing?

```
fn smallest(v: &[i32]) -> &i32 {
    let mut s = v[0];
    for r in &v[1..] {
        if *r < s {
            s = *r;
        }
    }
    &s
}
```

```
fn main() {
    let n = [12, 42, 6, 8, 15, 24];
    let s = smallest(&n);
    println!("{}", s);
}
```

Example 3: The following Rust program shows how to allocate a `struct` on the heap. It demonstrates that `Box<T>` implements the `Deref` trait, making it possible to obtain a reference to its contents. This reference can then be stored in a structure that expects an `&T` reference. Consider when is the heap memory allocated to the two boxes deallocated? Implement the `std::ops::Drop` trait for `Point` to print a message when the data is dropped to confirm your understanding.

```
struct Point {
    x : f32,
    y : f32
}

struct Rectangle<'a> {
    ul : &'a Point,
    br : &'a Point
}

impl<'a> Rectangle<'a> {
    fn area(&self) -> f32 {
        let w = self.br.x - self.ul.x;
        let h = self.ul.y - self.br.y;
        w * h
    }
}

fn main() {
    let ul = Box::new(Point{x : 3.0, y : 8.0});
    let br = Box::new(Point{x : 5.0, y : 4.0});
    let rect = Rectangle{ul : &ul, br : &br};
    let a = rect.area();
    println!("area = {}", a);
}
```

Example 4: The following Rust program does not compile. Thinking in terms of what data is moved, what data is borrowed, and the rules around multiple references, explain why it is correct that it does not compile.

```
fn main() {
    let v = vec![4, 8, 19, 27, 34, 10];
    let r = &v;
    let aside = v;
    println!("{}", r[0]);
    println!("{}", aside[0]);
}
```

Example 5: The following Rust program *does* compile. Again, thinking in terms of ownership of data, moves, and the rules around multiple references, explain why this is the expected behaviour.

```
struct Point {
    x : f32,
    y : f32
}

fn main() {
    let p = Point{x: 3.0, y : 5.0};
    let x = &p.x;
    let y = &p.y;
    println!("x={}", x);
    println!("y={}", y);
}
```

3 Resource Management

The following two blog posts describe ways in which the Rust type system can be used to manage the state of resources, to guarantee that operations on those resources are performed in the correct order:

- `https://yoric.github.io/post/rust-typestate/`
- `https://blog.systems.ethz.ch/blog/2018/a-hammer-you-can-only-hold-by-the-handle.html`

Read the two blog posts, making sure you understand how the code works and what role the type system plays in providing behaviour guarantees. Discuss the code during the labs if you are unsure how it works. Try to get the code from the second blog post working on your own machine. Think about cases where similar patterns could be used to enforce correct behaviour in the code you have previously written.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// There are at least 7 bugs relating to memory on this snippet.
// Find them all!

// Vec is short for "vector", a common term for a resizable array.
// For simplicity, our vector type can only hold ints.
typedef struct {
  int* data;     // Pointer to our array on the heap
  int  length;   // How many elements are in our array
  int  capacity; // How many elements our array can hold
} Vec;

Vec* vec_new() {
  Vec vec;
  vec.data = NULL;
  vec.length = 0;
  vec.capacity = 0;
  return &vec;
}

void vec_push(Vec* vec, int n) {
  if (vec->length == vec->capacity) {
    int new_capacity = vec->capacity * 2;
    int* new_data = (int*) malloc(new_capacity);
    assert(new_data != NULL);

    for (int i = 0; i < vec->length; ++i) {
      new_data[i] = vec->data[i];
    }

    vec->data = new_data;
    vec->capacity = new_capacity;
  }

  vec->data[vec->length] = n;
  ++vec->length;
}

void vec_free(Vec* vec) {
  free(vec);
  free(vec->data);
}

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

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

  free(vec->data);
  vec_free(vec);
}
```

Figure 1: Unsafe Memory Usage in C. The sample program contains at least seven memory related bugs – find them. The example is taken from `http://cs242.stanford.edu/f18/lectures/05-1-rust-memory-safety.html`, which also has the solution.