Introducing the Rust Programming Language (2/2)

Advanced Systems Programming (M) 2018-2019 – Laboratory Exercise 2
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 is a further introduction to the Rust programming language. **It is a formative exercise and is not assessed.**

2 Types, data structures, and traits in Rust

Some key features of the Rust programming language are those that allow it to express more advanced types and data structures, and those that support behavioural abstraction. This includes structure types, and the ability to implement methods on data structures; enumerated types and pattern matching, to express alternative types for an object; and generics, type parameters, and traits that allow abstraction over different types and behaviours.

2.1 Structure types and methods

Structure types describe heterogeneous collections of data. A `struct` has a name and comprises a set of zero or more fields. The fields may have names, as in the example below, or can be unnamed. If the fields are unnamed, the `struct` is used similarly to a tuple, but has a type name that makes it easier to reference in some contexts. It is possible for a `struct` to have no fields, in which case it takes up no space; such empty structures can be useful as marker types to describe that some condition is met. An example of a `struct` might be:

```
struct Rectangle {
    width: u32,
    height: u32
}
```

In addition to containing data, structures can have associated methods. These are specified in an `impl` block for the `struct`, and define functions that may be called on instances of that `struct`. For example, an `area()` method can be added to the `Rectangle` structure defined above in the following way:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect = Rectangle { width: 30, height: 50 };

    println!("Area of rectangle is {}", rect.area());
}
```

Methods implemented on a `struct` take `self` as an explicit parameter. This is typically done as a reference (`&self`); we'll discuss the self type of structures in more detail in Lecture 4. Fields are accessed with an explicit `self`, as in Python. Rust is not an object oriented language, but a `struct` with associated methods can be used to fill a similar role to an object in many cases.

Read chapter 5 of the online Rust book (`https://doc.rust-lang.org/book/ch05-00-structs.html`) and work through the examples. That chapter describes how structures and methods work in Rust. You will find some mentions of ownership and borrowing in part 3 of the chapter; we will discuss how Rust manages ownership of data in lecture 4, so skip over that part for now.

2.2 Enumerated types and pattern matching

Structures represent a single type that contains multiple fields of data. By contrast, an enumeration (`enum`) can be used to represent data that has several different varieties. An `enum` type has a name, and comprises several different variants. Each variant is its own type, and can contain fields much like a `struct`. When instantiated, an `enum` has type matching *one* of the variants. The example below describes a `RoughTime` enumeration with three variant types: `InThePast`, `JustNow`, and `InTheFuture`, two of which have fields associated with them:

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}

let when = RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);
```

The concept of *pattern matching* can be used to select between values, including different variants of enumerations. In its simplest form, a `match` expression in Rust work much like a `switch` statement in C or Java, but it can generalise to work on different types of data and to bind to associated fields in that data. For example, it's possible to match against variants of the `RoughTime` enumeration define above in the following way:

```
match rt {
    RoughTime::InThePast(units, count) => format!("{} {} ago", count, units.plural()),
    RoughTime::JustNow => format!("just now"),
    RoughTime::InTheFuture(units, count) => format!("{} {} from now", count, units.plural())
}
```

Read chapter 6 of the online Rust book (`https://doc.rust-lang.org/book/ch06-00-enums.html`). That chapter describes enumerated types and pattern matching. Work through the examples. Pay particular attention to the use of `Option<T>` as a way to signal functions that return optional values, rather than passing back a potentially null pointer.

2.3 Type parameters and traits

Rust provides three ways of writing code that is generic across different types. Firstly, if the types are known, is to possible to define an `enum` that encapsulates the possible alternatives then to pattern match on the variants.

Second, if the types are not known, it is possible to write generic data structures and functions that accept type parameters, abstracting across unknown data types. For example, a library that deals with coordinates and geometry (perhaps a graphics library) might need to deal with *points* in the abstract, independent on the underlying type used to specify x- and y-coordinates. Such a library could define a `Point<T>` type, with a *type parameter* `T` that's specified when the type is instantiated to define the actual type:

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let intPoint = Point { x: 5,   y: 10  };
    let fltPoint = Point { x: 1.0, y: 4.0 };
}
```

In case of `intPoint`, the parameter `T` an integer type. In the second case, `fltPoint`, the `Point` type is parameterised by a floating point type when it is instantiated.

Finally, it is possible to define *traits* to abstract across different types that offer the same behaviour. A `trait` definition specifies the name of the trait, along with a set of prototypes for methods that instances of the trait must implement, but provides no body for those methods. For example, a trait `Area` that defines a single method, `area()`, can be expressed as follows:

```
trait Area {
    fn area(&self) -> u32;
}
```

This is similar to an interface definition in a language such as Java. It indicates that types that implement the trait must provide implementations of the specified functions. Such implementations are provided by an `impl` block specifying the trait name and type for which it's implemented. For example, the `Area` trait mentioned above could be implemented as follows:

```
impl Area for Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Traits are often used to solve the same types of problems that subclasses solve in object oriented languages, or that duck-typing solves in dynamic languages such as Python: they make it possible to implement code that works with any type that implements a particular set of methods. This is done by specifying a trait bound as a type parameter to a function. For example, the `notify()` function below can be instantiated for any type `T`, where the definition of the type parameter (given in angle brackets) indicates that `T` implements the trait `Summary`:

```
trait Summary {
    fn summarize(&self) -> String;
}

fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Read the first two parts of chapter 10 of the Rust book (`https://doc.rust-lang.org/book/ch10-00-generics.html`) and work through the examples. The final part of the chapter, on "Validating References with Lifetimes", relates to the Rust ownership and borrowing rules that we'll discuss in Lecture 4, and shouldn't be attempted at this time.

3 Next Steps

At this point you should have a beginning understanding of the basics of programming in Rust. The later parts of this course will review the use of the Rust type system to help model the problem domain and help ensure correctness of the solution (Lecture 3), consider how it handles memory allocation and data ownership (Lectures 4 and 5), how it addresses the challenges of concurrency (Lecture 6), and how it supports asynchronous programming (Lecture 7). To prepare for this material, you should begin to review the remaining material in the online Rust book, and practice programming in Rust to ensure you start to have a good understanding of the language.