Introducing the Rust Programming Language

Advanced Systems Programming (H) 2022-2023 – Laboratory Exercise 3
Dr Colin Perkins, Dr Anna Lito Michala, School of Computing Science, University of Glasgow

1 Introduction

The Advanced Systems Programming (H) course uses the Rust programming language (`https://rust-lang.org/`) to illustrate several topics in systems programming. You are expected to learn the basics of programming in Rust in a self-directed manner as part of this course. To assist this learning, this exercise introduces the Rust programming language and its tool set. **This is a formative exercise and is not assessed.**

The expected timeline for this laboratory exercise is that you attempt at least the material in Section 2 by the end of this timetabled lab and aim to work on the first task in Section 4 to test your understanding. The remaining tasks can be undertaken at your own time during the rest of this week and before Lecture 4.

2 Rust Basics

As we discussed in Lectures so far, Rust is a statically typed systems programming languages. It has low overheads and generates highly efficient code – comparable to that produced by C and C++ compilers. It also has a rich type system and resource/memory ownership model that guarantee memory-safety and thread-safety, and that eliminate many classes of bugs at compile-time. **Lecture 2 introduces why this might be desirable. You should watch it before proceeding with the rest of this handout.**

This laboratory exercise seeks to introduce you to the basics of Rust programming, covering concepts that should be familiar from other programming languages. Later laboratory exercises will explore more novel, and more advanced, features of Rust such as ownership and the borrow checker for deterministic memory management, safe concurrency, and using the type system to model problems and help check solutions for consistency.

This exercise starts with the basics: variables, primitive types, function, control flow, structures, and enumerations. It also contains a number of open ended questions, for discussion during the timetabled lab sessions on 30 January 2023.

2.1 Variables

Variables in Rust are declared using a `let` binding, specifying the variable name, type, and value. For example, to declare a variable named `x`, of type `i32` (a 32-bit signed integer), with value `42`, you would write, within a function such as `main()`, a statement such as:

```
let x : i32 = 42;
```

The type annotation can usually be omitted, and will be inferred based on the value:

```
let y = 42;
```

Rust is strongly typed. Variables *always* have a known and fixed type, whether or not that type is explicitly declared in the code. If the type annotation is omitted from a declaration, then it is inferred by the compiler from the context. If the compiler cannot unambiguously infer the type for a variable, the code will not compile until a type annotation is added.

Discuss with the lecturer or lab demonstrator what is the difference between a statically typed language, such as Rust, that can infer the types of variables in some cases, and a dynamically typed language, such as Python, that does not require types to be specified.

Variables in Rust are immutable by default, and cannot be changed once bound to a value. If a mutable variable is needed, it can be bound as follows:

```
let mut z = 10;
```

Section 3.1 of the Rust book (`https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html`) discusses variables and mutability. Read it to find out more about variable bindings and mutability in Rust.

Do you think it is beneficial that variables in Rust are immutable by default? Think about, and discuss with the lecturer or lab demonstrator, why it might be desirable to promote a style of programming that emphasises immutable variables.

2.2 Basic Types

Rust is a strongly typed programming language. All values have a single, well-defined, type. The primitive types are the integral, floating point, boolean, and character types:

- The **integral types** include the 8-, 16-, 32-, and 64-bit integer values, written `i8`, `i16`, `i32`, and `i64`. They correspond to the `int8_t`, `int16_t`, `int32_t` and `int64_t` types in C. There are also unsigned integral types, (`u8`, `u16`, `u32`, and `u64`), that correspond to the C types `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t`.

  The `isize` and `usize` types are native sized integers for the processor architecture; that is, they'll be 32-bit if running on a 32-bit machines, 64-bit if running on a 64-bit machines, and so on. The `isize` type corresponds to an `int` in C, and the `usize` type corresponds to an `unsigned` in C.

Discuss with the lecturer or lab demonstrator what are the potential risks and benefits in having types, such as `isize`, that have implementation defined size as part of the Rust programming language?

- The **floating point** types in Rust are `f32` and `f64`. These are IEEE-754 single- and double-precision floating point values, analogous to `float` and `double` in C (although C does not guarantee that floating point arithmetic conforms to IEEE-754).
- The **boolean type** is `bool` and has values `true` and `false`.
- The **character type** represents a 21-bit Unicode Scalar Value. A character literal is written in single quotes (`'é'`), or as a quoted numeric Unicode value (`'\u00e9'`). Note that the definition of a Unicode Scalar Value includes values, such as combining accents, that don't necessarily fit the typical intuition of a character.

Rust commits to the Unicode representation for characters, and to the UTF-8 serialisation format for strings. Discuss with the lecturer or lab demonstrator whether you think this is a good idea.

In addition to these basic types, Rust supports compound types. These include tuples, where each value is unnamed but may be of a different type. For example, a tuple holding an integral value and a character might be defined as follows:

```
let t = (4, 'a');
```

Given a tuple variable, it's possible to de-structure it to access its individual elements. For example, given the above definition of the variable `tuple`, one could write:

```
let (a, b) = t;
println!("a = {}", a);
```

which would print the value of the first element of the tuple (it would print "`a = 4`").

The empty tuple type, `()`, is equivalent to `void` in C and is used to represent no value.

In addition to tuples, where each element can be a different type, Rust also has arrays where all the elements need to be of the same type:

```
let a = [10, 11, 12, 13, 14, 15];
let x = a[2];
println!("{}", x);
```

Rust stores the length along with the array, and checks array bounds.

Discuss with the lecturer or lab demonstrator how this representation of arrays differs from that used in the C programming language. Think about what are the advantages and disadvantages of the two approaches?

Read Section 3.2 of the Rust book (`https://doc.rust-lang.org/book/ch03-02-data-types.html`) for more details about the basic types.

2.3 Functions

Functions in Rust are declared using the keyword `fn`. They take arguments in parenthesis, and declare a return type. For example, a function named `area()` that takes two arguments, `w` and `h`, both of type `i32` and returns a value of type `i32`, can be defined as:

```
fn area(w: i32, h: i32) -> i32 {
  w * h
}
```

Function arguments have a name and a type annotation. The function return type is given after the `->` symbol. The types of function arguments and the return type must be specified; the compiler will not try to infer them. A function that returns nothing has return type `()`. In such cases, the return type annotation "`-> ()`" can be omitted from the function definition. Arguments are passed by value.

Rust has a `return` statement that can be used to return a value from a function. Alternatively, if the function ends without a return statement, the return value is the value of the last expression. In the example above, the function contains only a single expression (`w * h`) and omits the return statement, so the value returned is that of the expression.

Note that Rust distinguishes *expressions* and *statements*. A statement is terminated by a semicolon and evaluates to type `()`, representing no value. An expression does not end in a semicolon, and evaluates to some value. In the example function shown above, `w * h` is an expression that evaluates to type `i32`. This matches the return type of the function, so the function compiles and returns the expected value. If the expression were terminated by a semicolon, however, it would be interpreted as a statement with value `()`. Since `()` does not match the declared return type, the function would no longer compile.

Try compiling the `area` function above after adding a semicolon to the end of the `w * h` line to demonstrate this. Discuss with the lecturer or lab demonstrator to make sure you understand the difference between expressions and statements. Do you think Rust's approach of implicitly returning the value of the final expression in a function is a good idea?

Read Section 3.3 of the Rust book (`https://doc.rust-lang.org/book/ch03-03-how-functions-work.html`) that describes functions in more detail.

The designers of Rust made the choice not to allow the compiler to try to infer the types of function arguments or the function return type, and to require explicit type annotations on function definitions. In most cases, the compiler could infer the types, meaning the type annotations are unnecessary, except perhaps as documentation. Discuss with the lecturer or lab demonstrate whether you think requiring type annotations on function definitions was the right idea, or if it would be better to allow them to be omitted if the compiler can infer the type?

2.4 Control Flow: Conditional Expressions

Rust includes the usual features for basic control flow. Conditions can be expressed using `if-else` expressions. In basic use, these work in the same way as other languages:

```
if number < 5 {
    println!("number was less than five");
} else {
    println!("number was not less than five");
}
```

A key difference from many languages, however, is that `if-else` expressions evaluate to a value and can be used anywhere an expression is needed, for example:

```
let y = if function_returning_boolean() {
            42
        } else {
            7
        };
```

Since `if-else` is an expression, the `if` and `else` branches of the condition must evaluate to a value of the same type. In this example, since the value of the expression is used, the two branches are not terminated by semicolons and the branches both evaluate to integral values.

Discuss with the lecturer or demonstrator what error you get if one of the branches of the `if-else` expression is terminated by a semicolon and why this error occurs? Similarly, discuss what happens if both branches of the `if-else` expression are terminated by a semicolon.

2.5 Control Flow: Looping

Rust also includes the usual control flow statements for looping. The simplest of these is the `loop` statement that will loop forever:

```
loop {
    println!("forever");
}
```

There is also a `while` statement that allows iteration until some condition is met:

```
while x > 0 {
    println!("{}", x);
    x = x - 1;
}
```

There is no equivalent of the C `do...while` loop in Rust. It is possible to escape early from a `loop` or `while` loop using a `break` statement, or to abandon the current iteration and move onto the next using `continue`, in the usual way.

Rust also has the concept of a `for` loop that executes for each element of an iterator, much like the `for` loop in languages such as Java:

```
for e in [10, 20, 30, 40, 50] {
    println!("the value is: {}", e);
}
```

Read Section 3.5 of the Rust book `https://doc.rust-lang.org/book/ch03-05-control-flow.html` to learn more about control flow.

Rust does not support a C-style `for` loop. Discuss with the lecturer or lab demonstrator whether you think this was the right choice.

2.6 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, each of which can be of a different type. The fields names are optional, but their types must be specified. An example of a `struct` might be:

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

It is common for all of the fields in a `struct` to have names, as in the example above. Alternatively, it's possible to define what's known as a *tuple struct* where none of the fields are named. Tuple structs are used similarly to tuples, but have a type name that makes them easier to reference in some contexts. It is also possible for a `struct` to have no fields, in which case it takes up no space.

Discuss with the lecturer or lab demonstrator in what cases might a `struct` with no fields be useful?

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:

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

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

An `impl` block can contain more than one method definition. Methods implemented on a `struct` take `self` as an explicit parameter, and fields are accessed with an explicit `self`, as in Python. We'll discuss the self type of structures in more detail in Lecture 4.

A `struct` can be instantiated in a `let` statement by giving its type name, follow by values for each of its fields in braces. Methods defined on the `struct` can be called using dot notation, in the style of object oriented languages. The following example shows how to instantiate an instance of a `struct` and call a method on the resulting instance:

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

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

Despite the initial similarity to the way objects are used, Rust is not an object oriented language and it is not possible to create subclasses of a `struct`. However, a `struct` with associated methods can be used to fill a similar role to an object in many cases (*traits*, discussed in Section 2.8, and similar to interfaces in languages such as Java, are the main abstraction mechanism).

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.

2.7 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 a type matching that of *one* of the variants.

The example below defines two enumerated types, `TimeUnit` and `RoughTime`. The `TimeUnit` type has three possible variants, representing years, months, and days. The `RoughTime` type can describes approximate times and has three variant types: `InThePast`, `JustNow`, and `InTheFuture`. These variants are tuple structs, one of which has no fields, while the others each have two associated fields:

```
enum TimeUnit {
    Years, Months, Days
}

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

fn main() {
  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 defined above, and execute different code branches depending on the type of the enumeration, in the following way:

```
enum TimeUnit {
    Years, Months, Days
}

impl TimeUnit {
    fn plural(self) -> String {
      match self {
        TimeUnit::Years  => String::from("years"), // Lecture 3d talks about strings
        TimeUnit::Months => String::from("months"),
        TimeUnit::Days   => String::from("days")
      }
    }
}

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

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

  match when {
      RoughTime::InThePast(units, count) => {
        println!("{} {} ago", count, units.plural())
      },
      RoughTime::JustNow => {
        println!("just now")
      },
      RoughTime::InTheFuture(units, count) => {
        println!("{} {} 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.

Discuss with the lecturer or lab demonstrator to make sure you understand how enumerated types and pattern matching work. There is more flexibility here than the equivalent feature in many other programming languages.

2.8 Type Parameters and Traits

Rust provides three ways of writing code that is generic across different types. The first, if the types are known, is 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::<i32>{ x: 5,   y: 10  };
    let dblpoint = Point::<f64>{ x: 1.0, y: 4.0 };
}
```

In case of `intpoint`, the parameter `T` is replaced by the 32-bit integer type, `i32`, when the `Point` is instantiated. In the case of `dblpoint`, the type parameter `T` is replaced by the double precision floating point type when the `Point` 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.

Implementations of a trait are provided by an `impl` block that gives the trait name in addition to the name of the `struct` that implements that trait. For example, the `Area` trait mentioned above could be implemented for the `Rectangle` type, defined as in Section 2.6, as follows:

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

A type can implement more than one trait, if more than one `impl` block is provided, and can implement methods that are not part of any trait. Note that traits can define methods but not instance variables: they describe operations that can be performed on a `struct` but cannot add data to a structure.

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` that implements the trait `Summary`:

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

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

In this case, the type T is resolved to a concrete type that implements the particular trait at compile time. It is also possible to specify that a function returns some type that implements a trait, without specifying the exact type:

```
fn summarise() -> impl Summary {
  // ...
}
```

This syntax indicates that the `summarise()` function will return some type that implements the trait `Summary`, but does not specify what that type will be. Code that uses the return value can only call the methods defined on the trait, since it doesn't know the actual type.

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.

Discuss with the lecturer or lab demonstrator the differences in using traits as an abstraction mechanisms, as compared to using sub-classing in an object oriented language, and as compared to duck-typing in a dynamic language such as Python. What are the advantages and disadvantages of the different approaches?

3 Formative Exercises

To demonstrate your understanding of the basics of programming in Rust, complete the following exercises. These exercises are not assessed, and you do not need to submit your solutions.

1. Write a Rust program to represent shapes. Your program should define two structures, `Point` that represents a point on a 2d plane identified by x- and y-coordinates, and `Rectangle` that represents a rectangle defined by two points indicating its corners.
2. Extend your program to give each `Rectangle` an additional parameter to indicate whether it is opaque or transparent. This should be done by defining an `enum` with two variants to specify the degree of transparency, then adding a field to the `Rectangle` structure to hold a value of the enumerated type.
3. Implement a method on `Rectangle` to determine if a `Point` is covered by the rectangle. A point is covered by a rectangle if it falls within the boundaries of the rectangle and if the rectangle is opaque. Your code for this part should define an `impl` block for `Rectangle` containing a function `covers()` that takes a `Point` as a parameter and returns a `bool`. Your implementation of the `covers()` function should use pattern matching to determine if the rectangle is opaque.
4. Write a `main()` function that creates a array of values representing rectangles and a single point, then uses a `for` loop to iterate over the array to check whether the point is covered by any of the rectangles by calling the `covers()` method.
5. Extend the `impl` block of your `Rectangle` to add a method `width()` that returns the width of the rectangle.
6. Write a function `enlarge()` that takes a rectangle as a parameter and returns a new rectangle that has its top-left corner in the same position but is twice as wide. This should be defined as follows:

   ```
   fn enlarge{r: Rectangle) -> Rectangle {
     // ...body goes here
   }
   ```

7. Extend your main function so, after the `for` loop, it creates a new rectangle, enlarges it, and then prints the width of the enlarged rectangle. This should be structured as follows (where "..." indicates that text has been elided):

```
...
let r1 = Rectangle { ... };
let r2 = enlarge(r1);

println!("r2.width() = {}", r2.width());
...
```

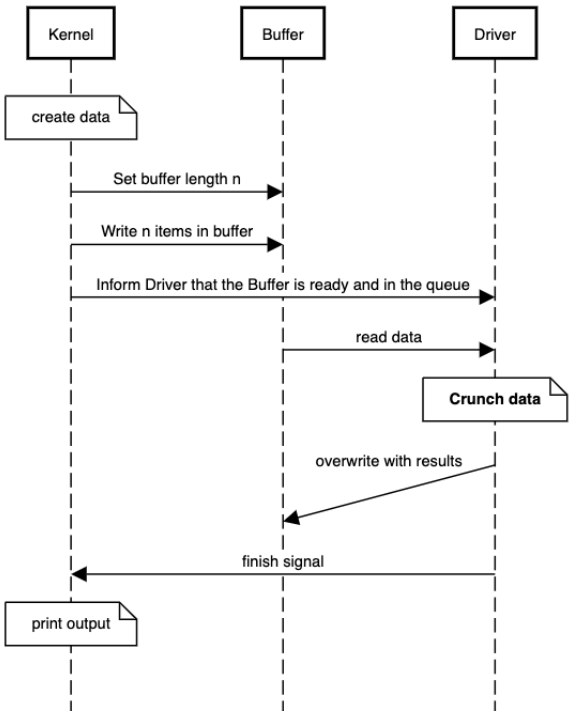
8. Extend your code to also print the width of `r1`. Lecture 4 will discuss why this doesn't work.

Discuss your solutions with the lecturer or lab demonstrators to make sure you understand how Rust works. As you can see, Rust mostly follows typical practice and behaves like a standard programming language. But, as point 8 above begins to demonstrate, there are some areas where it behaves differently to the norm.

4 Formative Exercises - Advanced

Now that you have coved the basics of programming in Rust, complete the following exercise to test the limits of these basic concepts and apply them in a systems programming example. This exercise is not assessed, and you do not need to submit your solutions. Consider the following sequence diagram of the Kernel talking to a GPU Driver. There are three entities. The `Driver` that represents the entity which will consume data, the `Buffer` which represents the entity which holds `n` data or results, and the `Kernel` that represents the provider of data and the consumer of results. The diagram represents point addition (Crunch data) using a GPU where a point is described by a two dimensional data point.

Traditionally you would write this in something like C and OpenCL but we will try to implement it in Rust. If it looks familiar it is only because it is inspired by the Operating Systems (H) assessment but without the API specifics! For this exercise we will simulate the process as you need advanced Rust features to actually implement a driver but you could do this as an added exercise if you want later in the course.



1. Write a Rust program to represent this communication. Your program should define the `Buffer` as `Vec<>` which represents the entity which holds `n` data or results in an array, and the structure `Point` used in the previous exercises to represents a point with two dimensions. For simplicity your `main` function can be your `kernel`.

2. Extend your program to represent the `Driver` as an `enum` with five states, `waiting, reading, processing, writing, done`. The driver always starts at waiting, moves to reading when promoted by the kernel, them moves to processing while performing the point addition. When the addition is complete it moves to writing mode and finally when it is ready to send the "finished" signal to the kernel it moves to done. Think about the methods that you need to write to move between these states of the `Driver` and how to update them.

3. Implement a method on `Buffer` to write a `Point` and update its size. Your code for this part should define a function `write()` that takes a `Point` as a parameter and returns a `bool`. Your implementation of the `write()` function should use pattern matching to determine if the buffer is available and in the waiting state.

4. Write a `main()` function that takes a command line parameter `n`, creates `n` `Points`, and adds them to the array using the `write()` method.

5. Extend the program to add a method `read()` that returns the contents of the buffer in order.

6. Write a function `crunch()` that performs point addition between two read points and continues until all the points in the `Buffer` are added. This should be defined as follows:

```
fn crunch{buffer: &Vec<Point>, d: Driver) -> Point {
  // ...body goes here
}
```

7. Extend your main function so, after the `crunch()` to print the result of the added points:

```
...
let sum = crunch(&buffer, d);

println!("sum.x() = {} and sum.y() ={}", sum.x(), sum.y());
...
```

   Readthis paper if you are curious about the difference between OpenCL and Rust implementations of GPU drivers.

5 Next Steps

At this point you should have a beginning understanding of the basics of programming in Rust – *now practice!* **You are expected to learn the basics of programming in Rust in a self-directed manner**. The laboratory exercises and lectures will direct you to appropriate resources, but on onus is on you to study the material.

In Lecture 3 we reviewed the material in this handout. The later parts of the course will then consider the use of the Rust type system to help model the problem domain and help ensure correctness of the solution (Lecture 4), consider how it handles memory allocation and data ownership (Lectures 5 and 6), how it addresses the challenges of concurrency (Lecture 7), and how it supports asynchronous programming (Lecture 8). 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.