Introducing the Rust Programming Language

Advanced Systems Programming (H) 2020-2021 – Laboratory Exercise 1
Dr Colin Perkins, 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 complete the preliminaries, in Section 2, and ensure that you have access to a working version of the Rust compiler and Cargo build tool, by the end of the timetabled lab session on Monday 11 January 2021. The remainder of the material, in Section 3, should be completed prior to the discussion of Lecture 3, on 25 January 2021.

2 Preliminaries

Ensure you have access to a recent version of the `rustc` compiler and the `cargo` package manager and build tool. These are pre-installed on the student Linux servers provided by the School (`stlinux01.dcs.gla.ac.uk` to `stlinux08.dcs.gla.ac.uk`), or you can download binaries from `https://rust-lang.org/` to install on your own system. When correctly installed, you should be able to run `rustc` and `cargo` and get output like the following, depending on the exact version you have installed (> is the command prompt):

```
>rustc --version
rustc 1.48.0
>cargo --version
cargo 1.48.0
>
```

These notes assume the Rust 2018 Edition (`rustc 1.31.0` or later) is available. Examples have been tested with `rustc 1.48.0`, which is the version installed on the student Linux servers in the School, but should work with any recent version of Rust and Cargo.

You can create, compile, and run a sample application using `cargo`. The `cargo` tool is the Rust package manager and build tool. It downloads any necessary packages, invokes the compiler (`rustc`), and executes the resulting binary:

```
>cargo new --bin hello
 Created binary (application) `hello` package
>cd hello/
>cat src/main.rs
fn main() {
  println!("Hello, world!");
}
```

```
>cargo run
   Compiling hello v0.1.0 (/users/staff/csp/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 5.38s
     Running `target/debug/hello`
Hello, world!
>
```

The resulting output file (in this case, `target/debug/hello`) is a stand-alone native executable file that can be directly run from the command prompt. The `cargo` tool produces debug builds by default, use `cargo run --release` to produce optimised release builds (smaller and *much* faster, but without debug symbols). The `cargo run` command compiles and runs an executable; the `cargo build` command can be used to compile, but not run, code.

Ensure you can compile and run the "Hello, world" application.

Inspect the `src/main.rs` and `Cargo.toml` files produced by `cargo` and make sure you understand their contents. Read Chapter 1 of the online Rust book (`https://doc.rust-lang.org/book/ch01-00-getting-started.html`) for more details.

3 Rust Basics

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 18 January 2021.

3.1 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:

```
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.

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?

3.2 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.

- 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.

What are the potential risks in having types, such as `isize`, that have implementation defined size as part of the Rust programming language?

Rust commits to the Unicode representation for characters, and to the UTF-8 serialisation format for strings. Do 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 tuple = (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) = tuple;
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.

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.

3.3 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 this case, the return type annotation "`-> ()`" can be omitted from the function definition. Arguments are passed by value. The return value of a function is the value of the last expression executed.

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, 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.

Functions can return early using a `return` statement in the usual way.

Section 3.3 of the Rust book (https://doc.rust-lang.org/book/ch03-03-how-functions-work.html) 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. Do you think requiring type annotations on function definitions was the right idea, or would it be better to allow them to be omitted if the compiler can infer the type?

3.4 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 was as other languages:

```
if number < 5 {
    println!("condition was true");
} else {
    println!("condition was false");
}
```

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. As usual for Rust, since the two branches of the `if-else` are expressions, they are not terminated by semicolons (the semicolon in the above code fragment terminates the `let` statement).

What error do you get if one of the branches of the `if-else` expression is terminated by a semicolon? Why?

What happens if both branches of the `if-else` expression are terminated by a semicolon? Why?

3.5 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:

```
let a = [10, 20, 30, 40, 50];
for element in a.iter() {
    println!("the value is: {}", element);
}
```

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. Do you think this was the right choice?

3.6 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
}
```

The fields may all be named, as in the example above, or may all 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, for example when used as type parameters (see Section 3.8) to tag another `struct`.

Can you think of cases where this might 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:

```
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. 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, and does not support subclasses, but a `struct` with associated methods can be used to fill a similar role to an object in many cases (*traits*, discussed in Section 3.8, and similar to interfaces in languages such as Java, are the main abstraction mechanism).

A `struct` can be instantiated in a `let` statement by giving its name, follow by values for each of its fields in braces. Any methods defined on the `struct` can be evaluated 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());
}
```

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.

3.7 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 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 defined above, and execute different code branches depending on the type of the enumeration, 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.

In your lab groups, discuss to what extent `enum` *types and pattern matching can replace sub-classing.*

3.8 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 3.6, 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.

In your lab group, discuss 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.

4 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.

Lecture 3 will review 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.