# Introducing the Rust Programming Language (1/2)

Advanced Systems Programming (M) 2018-2019 – Laboratory Exercise 1

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 introduces the Rust programming language and its tool set. **It is a formative exercise and is not assessed.**

Install the `rustc` compiler and the `cargo` package manager and build tool. These should both be pre-installed on the Linux-based lab machines, or if you wish to install on your own system you can download binaries from `https://rust-lang.org`. When correctly installed, you should be able to run `rustc` and `cargo` and get output like the following (> is the command prompt):

```
>rustc --version
rustc 1.31.1 (b6c32da9b 2018-12-18)
>cargo --version
cargo 1.31.0 (339d9f9c8 2018-11-16)
>
```

These notes assume the Rust 2018 Edition (`rustc 1.31.0` or later) is available, although the Rust 2015 Edition (`rustc 1.0` or later) can be used in most cases.

You can create, compile, and run a sample application using `cargo`. The `cargo` tool is the Rust package manager and build tool. It can download any necessary packages, run the compiler (`rustc`), and run 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/csp/tmp/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 2.13s
     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, pass the `--release` flag to produce optimised release builds (smaller and *much* faster, but without debug symbols).

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.

## 2 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.

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.

As with most programming languages, some essential features of Rust are how it supports variables, functions, and control flow; and what are the primitive types on which Rust programs operate.

## 2.1 Variables

Variables are declared using a `let` binding, specifying the variable name, type, and value:

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

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

```
let x = 42;
```

As in most functional languages, variables 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 y = 10;
```

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

## 2.2 Functions

Functions in Rust are declared using the keyword `fn`. They take arguments in parenthesis, and declare a return type:

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

Arguments are written in the form `name: type` and the return type is given after the `->` symbol. Arguments are passed by value. A function returns the value of its last expression. Beware: terminating an expression with a semicolon turns it into a statement that returns nothing (a value `()` of type unit). 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. Read Section 3.3 of the Rust book (`https://doc.rust-lang.org/book/ch03-03-how-functions-work.html`) for more information about functions.

## 2.3 Control Flow

Conditions can be expressed using `if-else` expressions. These work much the same as in other languages:

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

A key difference from some 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. Rust also include the usual control flow statements for looping. The simplest of these is the infinite loop:

```
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 some code for each element of an iterator:

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

The `for` loop works a lot like the equivalent in Java. There is no C-style for loop. For more details of control flow, read Section 3.5 of the Rust book: `https://doc.rust-lang.org/book/ch03-05-control-flow.html`.

### 2.4 Primitive Types

Rust is a strongly typed programming language. All values have a single known type. The types must specified in function definitions, and can be specified (but are more commonly inferred) in variable bindings. The basic types are integral, floating point, boolean, and character types. Integral types include 8-, 16-, 32-, and 64-bit integers (`i8`, `i16`, `i32`, and `i64`) and the corresponding unsigned types (`u8`, `u16`, `u32`, and `u64`). These correspond to the `int8_t`, `int16_t`, etc. types in C. 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. These correspond to the `int` and `unsigned` types 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` (unlike C, integers cannot be used as boolean values). Characters literals are specified in single quotes (`'x'`) and represent Unicode Scalar Values.

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:

```
let t = (4, 3);
```

and arrays where values must all 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 primitive types.

## 3 Exercises

You should read one of the recommended books about Rust, to begin to get an understanding of the language. In parallel, complete the following basic exercises:

1. On your local machine, create and run the "Hello, world!" project described in the Introduction of this handout. The goal here is to show that you have a working Rust installation.

2. Review the Rustlings website (`https://github.com/rustlings/rustlings`) and complete the exercises in the Variable Bindings, Functions, Primitive Types, and If sections. These exercises should run in your browser, or you can download and run the code locally. The goal here is to show that you understand the basics of Rust control flow, function, and types.

The material covered next week, in Lecture 2 and Lab 2, will begin to look at more advanced types and features of Rust. Once you have completed these exercises, you might want to read ahead about the Rust type system, including `struct` and `enum` types, and the basics of references.