Introducing Rust

- What is Rust?
  - **Basic operations and types**
  - Abstraction: Traits, Enumerated types and pattern matching
  - Memory allocation and boxes
- Why is it interesting?

The Rust Programming Language

- A modern systems programming language with a strong static type system
- Initially developed by Graydon Hoare as a side project, starting 2006
- Sponsored by Mozilla since 2009
- Rust v1.0 released in 2015
- Rust v1.31 “Rust 2018 Edition” released December 2018
  - Backwards compatible – but tidies up the language
  - https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html
- New releases made every six weeks – strong backwards compatibility policy

Basic Features and Types (1/10)

```
fn main() {
    println!("Hello, world!”);
}
```

Function definition: **`main()`** takes no arguments, returns noting

Macro expansion: **`println!()`** with string literal as argument

Basic Features and Types (2/10)

```
use std::env;

fn main() {
    for arg in env::args() {
        println!("{:?}", arg);
    }
}
```

Import **`env`** module from standard library

Use **`for`** loop to iterate over command line arguments

```
$ rustc args.rs
$ ./args 1 2 3
"./args"
"1"
"2"
"3"
$
```

Basic Features and Types (3/10)

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m!=0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}

fn main() {
  let m = 12;
  let n = 16;
  let r = gcd(m, n);
  println!("gcd({}, {}) = {}", m, n, r);
}
```

```
$ rustc gcd.rs
$ ./gcd
gcd(12, 16) = 4
$
```

Function arguments and return type; mutable vs immutable

Control flow: **`while`** and **`if`** statements

Local variable definition (**`let`** binding); type is inferred

Function implicitly returns value of final expression
(**`return`** statement allows early return)

Function call, assigning result to local variable

Basic Features and Types (4/10)

C	Rust
`unsigned`	`usize`
`uint8_t, unsigned char`	`u8`
`uint16_t`	`u16`
`uint32_t`	`u32`
`uint64_t`	`u64`

C	Rust
`int`	`isize`
`int8_t, signed char`	`i8`
`int16_t`	`i16`
`int32_t`	`i32`
`int64_t`	`i64`
`float`	`f32`
`double`	`f64`
`int`	`bool`
	`char`

- Primitive types map closely to those in C
  - Rust has native **`bool`**, C uses **`int`** to represent boolean
  - In C, a **`char`** is one byte, implementation defined if signed, character set unspecified
  - In Rust, a **`char`** is a 32-bit Unicode scalar value:
    - Unicode scalar value ≠ code point ≠ grapheme cluster ≠ “character”
    - e.g., ü is two scalar values “Latin small letter U (U+0075)” + “combining diaeresis (U+0308)”, but one grapheme cluster (https://crates.io/crates/unicode-segmentation – text is hard)

https://doc.rust-lang.org/book/ch03-02-data-types.html

Basic Features and Types (5/10)

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let b = a[2];
    println!("b={}", b);
}
```

Arrays work as expected; elements are all the same type; number of elements fixed
Type of the array is inferred from type of elements

Array types are written `[T]`, where `T` is the type of the elements

```
$ rustc array.rs
$ ./args
b=3
$
```

Basic Features and Types (6/10)

```
fn main() {
    let mut v : Vec<u32> = Vec::new();
    v.push(1);
    v.push(2);
    v.push(3);
    v.push(4);
    v.push(5);
}
```

Vectors, **`Vec<T>`**, are the variable sized equivalent of arrays
The type parameter **`<T>`** specifies element

The **`: Vec<u32>`** modifier specifies the type for variable **`v`**
and can usually be omitted, allowing compiler to infer type:

```
let mut v = Vec::<u32>::new();
```

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
}
```

The **`vec![...]`** macro is a shortcut to create vector literals

Vectors implemented as the equivalent of a C program that uses **`malloc()`** to allocate space for an array, then **`realloc()`** to grow the space when it gets close to full.

A vector, **`Vec<T>`**, can be passed to a function that expects a reference to an array **`[T]`**

*(**`Vec<T>`** implements the trait **`Deref<Target=&[T]>`** that defines the conversion)*

Basic Features and Types (7/10)

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
    println!("The 2nd element is {}”, tup.1)
}
```

Tuples are collections of unnamed values; each element can be a different type

`let` bindings can de-structure tuples

Tuple elements can be accessed by index

```
()
```

An empty tuple is the unit type (like **`void`** in C)

Basic Features and Types (8/10)

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

fn area(rectangle: Rectangle) -> u32 {
    rectangle.width * rectangle.height
}

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

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

Structs are collections of named values; each element can have a different type

https://doc.rust-lang.org/book/ch05-00-structs.html

Access fields in struct using dot notation

Creates a struct, specifying field values

```
$ rustc struct.rs
$ ./struct
Area of rectangle is 1500
$
```

Basic Features and Types (9/10)

```
struct Point(i32, i32, i32);

let origin = Point(0, 0, 0);
```

Elements of a struct can be unnamed: known as tuple structs
useful as type aliases

```
struct Marker;
```

Unit-like structs have no elements and take up no space
useful as markers or type parameters

Basic Features and Types (10/10)

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

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 defined in **`impl`** block

Explicit **`self`** references, like Python

Method call uses dot notation

Methods can be implemented on structs – somewhat similar to objects, but Rust does not support inheritance or sub-types

Introducing Rust

- What is Rust?
  - **Basic operations and types**
  - Abstraction: Traits, Enumerated types and pattern matching
  - Memory allocation and boxes
- Why is it interesting?