Rust Basics

- Rust is a statically and strongly typed systems programming language – along the lines of C:
  - *statically* means that all types are known at compile-time,
  - *strongly* means that these types are designed to make it harder to write incorrect programs. A successful compilation means you have a much better guarantee of correctness than with a language like C.
  - *systems* means generating the best possible machine code with full control of memory use.
- So the uses are pretty hardcore: operating systems, device drivers and embedded systems that might not even have an operating system.
- However, it's actually a very pleasant language to write normal application code in as well.

Assignments

- Explore the links in the github readme
- Do Assignment #1 – see BrightSpace

Unifying Principles

- The big difference from C and C++ is that Rust is *safe by default*; all memory accesses are checked. It is not possible to corrupt memory by accident.
- The unifying principles behind Rust are:
  - strictly enforcing *safe borrowing/references* of data
  - functions, methods and closures to operate on data
  - tuples, structs and enums to aggregate data
  - pattern matching to select and destructure data
  - traits to define *behavior* on data
- We will look at some of these features in detail!

Hello World

- The original purpose of "hello world", ever since the first C version was written, was to test the compiler and run an actual program.

Hello World

- Let’s try it:
  - Log onto the ”CSC301 Machine” – see BrightSpace
  - Use one of the Linux editors (atom, vim, emacs, nano) to create the Rust version of ‘hello world’:

```rust
fn main() {
    println!("Hello, World!");
}
```

  - Save it as ‘hello.rs’
  - Compile it:

```
$ rustc hello.rs
```

  - Run it

```
$ ./hello
Hello, World!
```

Loops and Ifs

- As expected, Rust supports loops and if statements.

```
fn main() {
    for i in 0..5 {
        if i % 2 == 0 {
            println!("even {}",
        } else {
            println!("odd {}",
        }
    }
}
```

Primitive Data Types

- What is a surprise perhaps as you look at programs is that contrary to C you don't have to declare the data type of a variable – Rust figures it out similar to Python with one big difference:
  - It figures it out at compile time!

Necessary keyword!

```
fn main() {
    let mut sum = 0;
    for i in 0..5 {
        sum += i;
    }
    println!("sum is {}", sum);
}
```

The 'mut' keyword for mutable is Necessary if you want to write To a variable. By default, Rust Treats all variables as constants!

Primitive Data Types

- We can of course declare data types for variables:

```
fn main() {
    let mut sum : i16 = 0;
    for i in 0i16..5i16 {
        sum = sum + i;
    }
    println!("sum is {}", sum);
}
```

- Try declaring ‘sum’ as i32 and see what happens…

Primitive Data Types

- Technically we say that Rust does not support ‘type hierarchies’, that is, types such as i32 and i16 are completely unrelated and can only be made compatible via ‘type casting’.

```
fn main() {
    let mut sum : i32 = 0;
    for i in 0i16..5i16 {
        sum = sum + i as i32;
    }
    println!("sum is {}", sum);
}
```

Type cast

Functions

- The type or signature of a function has to be completely specified in Rust – contrary to variable declarations.

```
fn sqr(x: f64) -> f64 {
    return x * x;
}
```

- Each parameter is declared similar to variables and the return value is declared with the ‘->’ symbol.
- If no ‘->’ symbol appears in the function declaration then it is a ‘void’ function returning the value of type ‘()’.

Functions

- You actually do not have to specify the 'return' statement:

```
fn sqr(x: f64) -> f64 {
    x * x
}
```

- The return value is the last value computed - NO semi-colon!

Function Types

- Just integer variables have type notations such as i16 and i64
- Functions also have type notations.
- Take our sqr function:

```
fn sqr(x: f64) -> f64 {
    return x * x;
}
```

- It has a type of ‘fn(f64)->f64’

Function Types

- We can now do the following:

```
fn sqr(x:f64) -> f64 {
    x*x
}

fn main() {
    let y: fn(f64) -> f64 = sqr;
    let x = y(3.0);
    println!("{}", x)
}
```

Functions and Borrowing

- By default all function arguments are passed ‘by value’ in Rust – that is, the value of each argument is copied into the function – we learn more about this later in the course….
- ‘Borrowing’ allows you to override this behavior

Borrowing

- Borrowing is Rust’s way of talking about references
- Here is an example:

```
fn by_ref(x: &i32) -> i32{
    *x + 1
}

fn main() {
    let i = 10;
    let res1 = by_ref(&i);
    let res2 = by_ref(&41);
    println!("{} {}", res1,res2);
}
```

- Notice the asterisk needed to access the actual value stored in the reference ‘x’.
- This example is kind of silly but it makes sense if ‘x’ were a large object

Borrowing

- If the function wants to modify the passed object you need to insert a **mutable reference**

```
fn modifies(x: &mut f64) {
    *x = 1.0;
}

fn main() {
    let mut res = 0.0;
    modifies(&mut res);
    println!("res is {}", res);
}
```

Arrays

- All statically-typed languages have *arrays*, which are values packed nose to tail in memory.
- Arrays are *indexed* from zero.

```
fn main() {
    let arr = [10, 20, 30, 40];
    let first = arr[0];
    println!("first {}", first);

    for i in 0..4 {
        println!("[{}] = {}", i,arr[i]);
    }
    println!("length {}", arr.len());
}
```

```
first 10
[0] = 10
[1] = 20
[2] = 30
[3] = 40
length 4
```

Array Types

- Array type notation is: [<type>; <size>]

```
fn main() {
    let x = [10, 20, 30, 40];
    let y: [i64;4] = x;

    for i in 0..y.len() {
        println!("[{}] = {}", i,y[i]);
    }
}
```

Array Slices

- Array slices allows us to access (parts of) arrays without having to give size info.
- Example:

```
fn main() {
    let x = [10, 20, 30, 40];
    let y: &[i64] = &x[1..3];

    for i in 0..y.len() {
        println!("[{}] = {}", i,y[i]);
    }
}
```

- Turns out that that is very convenient for function calls.

Functions & Array Slices

- Array slices allow us to write functions that work on arrays of any lengths

```rust
fn sum(values: &[i32]) -> i32 {
    let mut res = 0;
    for i in 0..values.len() {
        res += values[i]
    }
    res
}

fn main() {
    let x = [10,20,30,40];
    let xsum = sum(&x);
    println!("xsum = {}", xsum);

    let y = [5,2];
    let ysum = sum(&y);
    println!("ysum = {}", ysum);
}
```

Structs

- Structs in Rust works similar to ‘class’ in languages like Java and Python
  - We’ll see later that the similarity is much deeper than suggested as first glance
- Example:

```

struct Rectangle {
  xdim : i32,
  ydim : i32
}

fn main() {
  let r = Rectangle {xdim:10, ydim:15};
  println!("r = xdim:{}, ydim:{}", r.xdim, r.ydim);
}

```

Constructor

Tuples

- Similar to structs, tuples are simple data structures that can hold values of different types.
- The main difference is that you construct tuples on the fly without prior declarations.

```
// let rust figure out how to print out
// a tuple using the {:?} formatter.
#[derive(Debug)]

struct Point {
  x : f64,
  y : f64
}

// using the 'type' command to create a
// type alias for f64
type Radius = f64;

fn main() {
  let circle = (10.5, Point {x:3.0, y:5.5});
  // explicitly typed tuple
  let mycircle : (Radius, Point) = circle;

  println!("{:?}", mycircle);
}
```

Enums

- If you are familiar with C then enums in Rust are an interesting combination of C-enums and C-unions.
- The example uses Rust enums in a typical Rust use-case:
  - creating a type that allows for different variants which themselves have different types.
  - Note that Rust keeps track of the type of the actual return value from the function.

```rust
#[derive(Debug)]
enum Value {
    Number(f64),
    Bool(bool)
}

fn different_values(x: f64) -> Value {
    if x > 0.0 {
        Value::Number(x)
    } else {
        Value::Bool(false)
    }
}
fn main() {
    let y1 = different_values(3.5);
    let y2 = different_values(-0.1);

    println!("y1={:?} y2={:?}", y1, y2);
}
// output:
// y1=Number(3.5) y2=Bool(false)
```