# DS-210: PROGRAMMING FOR DATA SCIENCE

## LECTURE 15

## RUST: COMPILING. BASIC TYPES AND VARIABLES. PROJECT MANAGER (`cargo`).

# REMINDER: MIDTERM IS ON MONDAY

- Same time as always
- Arrive early or on time

# REMINDER: MIDTERM IS ON MONDAY

- Same time as always
- Arrive early or on time

# FINAL PROJECT DISCUSSION

# WRITE AND COMPILE SIMPLE RUST PROGRAM

```
fn main() {
    let x = 9;
    let y = 16;
    println!("Hello, snow!");
    println!("{} plus {} is {}", x, y, x+y);
}
```

**A few facts:**

- function `main`: the code that is executed
- `println!` is a macro:
    - first parameter is a format string
    - `{}` are replaced by the following parameters

# WRITE AND COMPILE SIMPLE RUST PROGRAM

```rust
fn main() {
    let x = 9;
    let y = 16;
    println!("Hello, snow!");
    println!("{} plus {} is {}", x, y, x+y);
}
```

**A few facts:**

- function `main`: the code that is executed
- `println!` is a macro:
  - first parameter is a format string
  - `{}` are replaced by the following parameters

**Simplest way to compile:**

- put the content in file `hello.rs`
- command line:
  - navigate to this folder
  - `rustc hello.rs`
  - run `./hello` or `hello.exe`

# VARIABLE DEFINITIONS

- By default immutable!

```
In [ ]: let x = 3;
        x = x + 1; // <== error here
```

# VARIABLE DEFINITIONS

- By default immutable!

```
In [2]: let x = 3;
        x = x + 1; // <== error here

        let x = 3;
            ^ first assignment to `x`
        x = x + 1; // <== error here
        ^^^^^^^^^ cannot assign twice to immutable variable
        cannot assign twice to immutable variable `x`
        help: consider making this binding mutable

        mut x
```

# VARIABLE DEFINITIONS

- By default immutable!

```
In [2]: let x = 3;
        x = x + 1; // <== error here

let x = 3;
    ^ first assignment to `x`
x = x + 1; // <== error here
^^^^^^^^^ cannot assign twice to immutable variable
cannot assign twice to immutable variable `x`
help: consider making this binding mutable

mut x
```

- Use `mut` to make them mutable

```
In [3]: // mutable variable
        let mut x = 3;
        x = x + 1;
        x

Out[3]: 4
```

# VARIABLE DEFINITIONS

- By default immutable!

In [2]:
```
let x = 3;
x = x + 1; // <== error here
```

```
let x = 3;
    ^ first assignment to `x`
x = x + 1; // <== error here
^^^^^^^^^ cannot assign twice to immutable variable
cannot assign twice to immutable variable `x`
help: consider making this binding mutable

mut x
```

- Use `mut` to make them mutable

In [3]:
```
// mutable variable
let mut x = 3;
x = x + 1;
x
```

Out[3]: 4

- Variable shadowing: new variable with the same name

In [4]:
```
let solution = "4";
let solution : i32 = solution.parse()
                     .expect("Not a number!");
let solution = solution * (solution - 1) / 2;
println!("solution = {}",solution);
```

```
solution = 6
```

# BASIC TYPES: INTEGERS AND FLOATS

- unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize` (architecture specific size)
  - from $0$ to $2^n - 1$
- signed integers: `i8`, `i16`, `i32` (default), `i64`, `i128`, `isize` (architecture specific size)
  - from $-2^{n-1}$ to $2^{n-1} - 1$

(if you need to convert, use the `as` operator)

# BASIC TYPES: INTEGERS AND FLOATS

- unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize` (architecture specific size)
  - from $0$ to $2^n - 1$
- signed integers: `i8`, `i16`, `i32` (default), `i64`, `i128`, `isize` (architecture specific size)
  - from $-2^{n-1}$ to $2^{n-1} - 1$

(if you need to convert, use the `as` operator)

In [5]:
```
let x : i16 = 13;
let y : i32 = -17;
// won't work without the conversion
(x as i32) * y
```

Out[5]: -221

# BASIC TYPES: INTEGERS AND FLOATS

- unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize` (architecture specific size)
  - from $0$ to $2^n - 1$
- signed integers: `i8`, `i16`, `i32` (default), `i64`, `i128`, `isize` (architecture specific size)
  - from $-2^{n-1}$ to $2^{n-1} - 1$

(if you need to convert, use the `as` operator)

In [5]:
```
let x : i16 = 13;
let y : i32 = -17;
// won't work without the conversion
(x as i32) * y
```

Out[5]: -221

- floats: `f32` and `f64` (default)

In [6]:
```
let x = 4;
let z = 1.25; // default float type: f64
// won't work without the conversion
(x as f64) * z
```

Out[6]: 5.0

# BASIC TYPES: BOOLEANS, CHARACTERS, AND STRINGS

- `bool` uses one byte of memory

In [7]:
```
let x = true;
let y: bool = false;

// x and (not y)
x && !y
```

Out[7]:
```
true
```

# BASIC TYPES: BOOLEANS, CHARACTERS, AND STRINGS

- `bool` uses one byte of memory

In [7]:
```
let x = true;
let y: bool = false;

// x and (not y)
x && !y
```

Out[7]: true

- `char` defined via single quote, uses four bytes of memory (Unicode scalar value)

In [8]:
```
let x = 'a';
let y = '🚦';
let z = '🦕';
```

# BASIC TYPES: BOOLEANS, CHARACTERS, AND STRINGS

- `bool` uses one byte of memory

In [7]:
```
let x = true;
let y: bool = false;

// x and (not y)
x && !y
```

Out[7]: true

- `char` defined via single quote, uses four bytes of memory (Unicode scalar value)

In [8]:
```
let x = 'a';
let y = '🚦';
let z = '🦕';
```

- string slice defined via double quotes (not so basic actually!)

In [9]:
```
let s1 = "Hello! How are you, 🦕?";
let s2 : &str = "Zażółć gęślą jaźń.";
```

# PROJECT MANAGER: `cargo`

- create a project: `cargo new PROJECT-NAME`
- main file will be `PROJECT-NAME/src/main.rs`

# PROJECT MANAGER: `cargo`

- create a project: `cargo new PROJECT-NAME`
- main file will be `PROJECT-NAME/src/main.rs`

- to run: `cargo run`
- to just build: `cargo build`

# PROJECT MANAGER: `cargo`

- create a project: `cargo new PROJECT-NAME`
- main file will be `PROJECT-NAME/src/main.rs`

- to run: `cargo run`
- to just build: `cargo build`

Add `--release` to create a "fully optimized" version:

- longer compilation
- faster execution
- some runtime checks not included (e.g., integer overflow)
- debuging information not included
- the executable in a different folder

# PROJECT MANAGER: `cargo`

If you just want to **check** if your current version compiles: `cargo check`

- Much faster for big projects