

CMSC 330: Organization of Programming Languages

Type-Safe, Low-level Programming with
Rust

Type Safety in Programming Languages

- In a type-safe language, the type system enforces well defined behavior. Formally, a language is type-safe *iff*

$G \vdash e : t$ and $G \vdash A$ implies

$A; e \Rightarrow v$ and $\vdash v : t$ or that e runs forever

- $A; e \Rightarrow v$ says e *evaluates* v under environment A
- $G \vdash e : t$ says e *has type* t under type environment G
- $G \vdash A$ says A *is compatible with* G
 - For all x , $A(x) = v$ implies $G(x) = t$ and $\vdash v : t$

C/C++: *Not* Type-Safe – Spatially Unsafe

$G \vdash e : t$ and $G \vdash A$ implies
 $A; e \Rightarrow v$ and $\vdash v : t$ or that e runs forever

Type safety is violated by **buffer overflows**

```
int main() {  
    int x = 1, *p = &x;  
    int y = 0, *q = &y;  
    *(q+1) = 5; // overwrites p  
    return *p; // crash  
}
```

C/C++: *Not* Type-Safe – Temporally Unsafe

and **dangling pointers** (uses of pointers to freed memory)

```
{ int *x = ...malloc();  
  free(x);  
  *x = 5; /* oops! */  
}
```

... which can happen via the stack, too:

```
int *foo(void) { int z = 5; return &z; }  
void bar(void) {  
    int *x = foo();  
    *x = 5; /* oops! */  
}
```

Automatic Memory Management

- Data may be **allocated** explicitly or implicitly. Data **reclamation** occurs **automatically**: No manual **free**
- A **garbage collector** traces pointers in use by the program, starting from the stack and global variables
 - **Retains** those objects it can **reach** (since could be used later)
 - **Reclaims** those it **cannot**
- Related technique: **Reference counting**
- Both impose **space** and **run-time costs**

Memory Management in (Type-Safe) OCaml

- Local variables live on the stack
- Tuples, closures, and constructed types live on the heap

```
let x = (3, 4) (* heap-allocated *)
```

```
let f x y = x + y in f 3
```

```
(* result heap-allocated *)
```

```
type 'a t = None | Some of 'a
```

```
None (* not on the heap—just a primitive *)
```

```
Some 37 (* heap-allocated *)
```

- Heap data reclaimed via [garbage collection](#)

In sum: What choice do programmers have?

C/C++

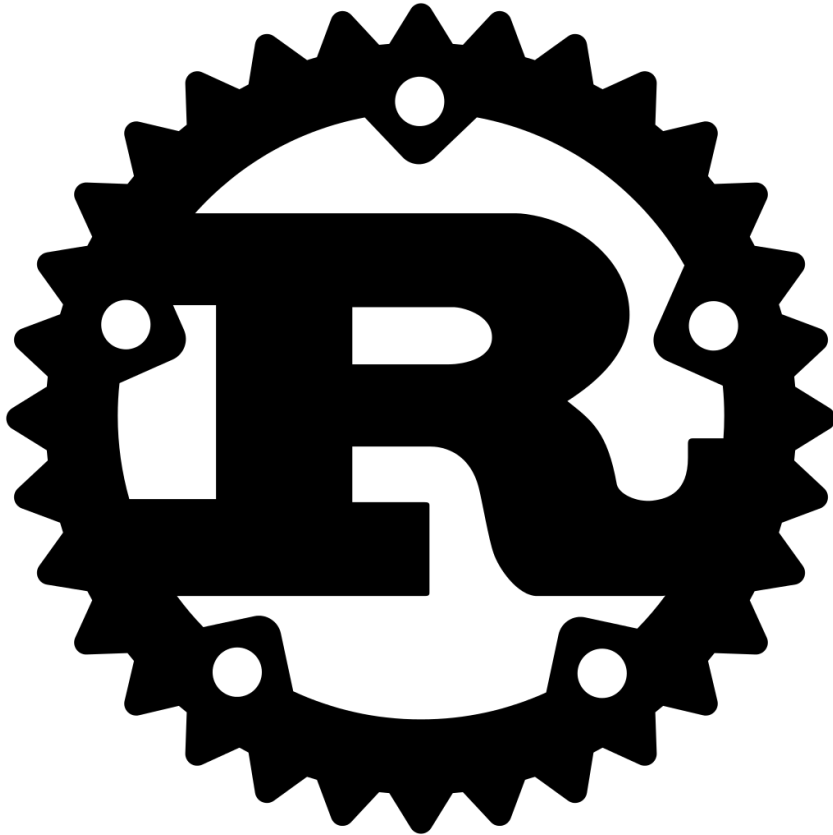
- **Type-unsafe**
- **Low level** control
- **Performance** over safety and ease of use
- **Manual** memory management, e.g., with **malloc/free**

Java, OCaml, Go, Ruby...

- **Type safe**
- **High level**, less control
- **Ease-of-use** and **safety** over performance
- **Automatic** memory management via **garbage collection**
 - No explicit malloc/free

Something in between ... ?

Rust: Type-safe (and Thread-safe), and Fast



- A Mozilla-sponsored, public project since 2010
 - Started in 2006 by Graydon Hoare while at Mozilla
- Most loved programming language in Stack Overflow annual surveys every year from 2016 through 2021
- Key properties: Type safety, and no data races, despite use of concurrency and manual memory management

Rust in the Real World

- Firefox Quantum and Servo components
 - <https://servo.org>
- REmacs port of Emacs to Rust
 - <https://github.com/Wilfred/remacs>
- Amethyst game engine
 - <https://www.amethyst.rs/>
- Magic Pocket filesystem from Dropbox
 - <https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/>
- OpenDNS malware detection components
- <https://www.rust-lang.org/en-US/friends.html>

Features of Rust

- Lifetimes and Ownership
 - Key feature for ensuring safety
- Traits as core of object(-like) system
- Variable default is immutability
- Data types and pattern matching
- Type inference
 - No need to write types for local variables
- Generics (aka parametric polymorphism)
- First-class functions
- Efficient C bindings

Takes ideas from
functional and OO
languages, and
recent research

Installing Rust

- Instructions, and stable installers, here:

<https://www.rust-lang.org/en-US/install.html>

- On a Mac or Linux (VM), open a terminal and run

`curl https://sh.rustup.rs -sSf | sh`

- On Windows, download+run `rustup-init.exe`

<https://static.rust-lang.org/rustup/dist/i686-pc-windows-gnu/rustup-init.exe>

Rust Compiler, Build System

- Rust programs can be compiled using `rustc`
 - Source files end in suffix `.rs`
 - Compilation, by default, produces an executable
 - No `-c` option
- Preferred: Use the `cargo` package manager
 - Will invoke `rustc` as needed to build files
 - Will download and build dependencies
 - Based on a `.toml` file and `.lock` file
 - You won't have to mess with these for this class
 - Like `ocamlbuild` or `dune`

Using cargo

- Make a project, build it, run it

```
% cargo new hello_cargo --bin
```

```
% cd hello_cargo
```

```
% ls
```

```
Cargo.toml  src/
```

```
% ls src
```

```
main.rs
```

```
% cargo build
```

```
Compiling hello_cargo v0.1.0 (file:///...)
```

```
Finished dev [unoptimized + debuginfo] ...
```

```
% ./target/debug/hello_cargo
```

```
Hello, world!
```

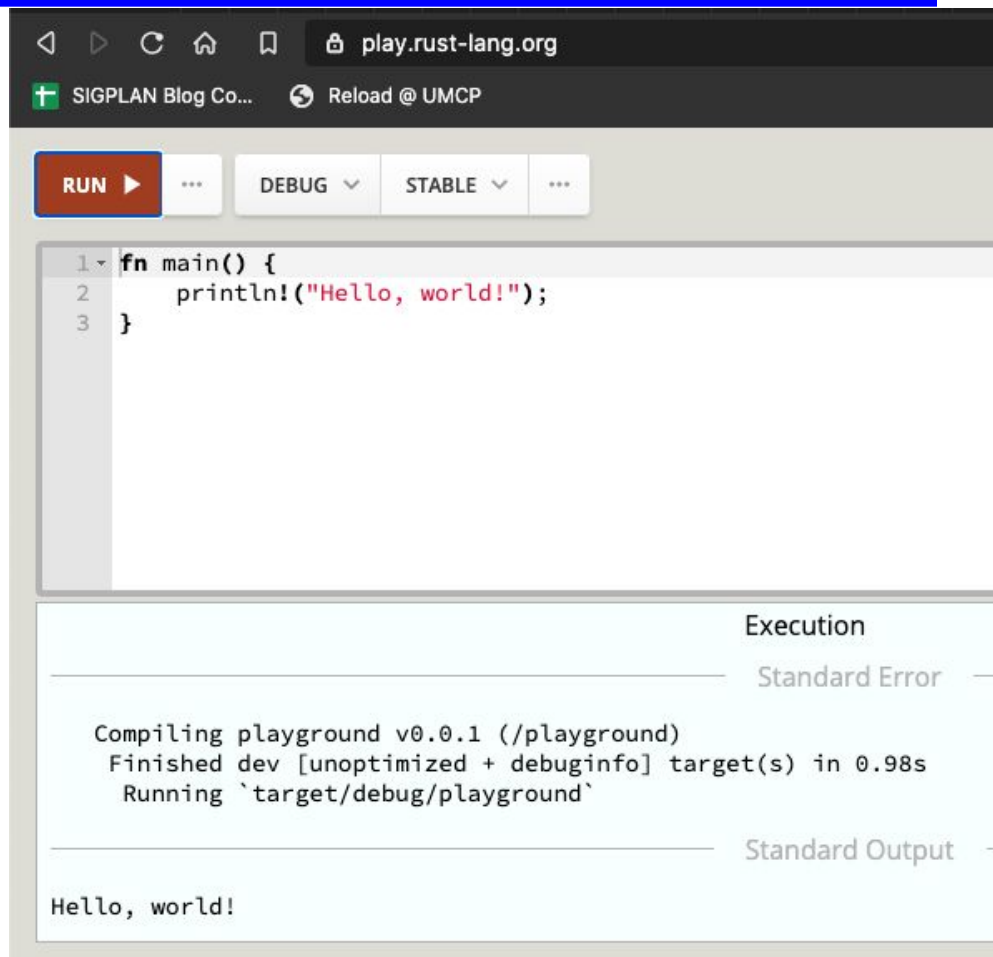
Use `cargo` to run tests, too; will discuss later

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

Uses `rustc`, the Rust compiler

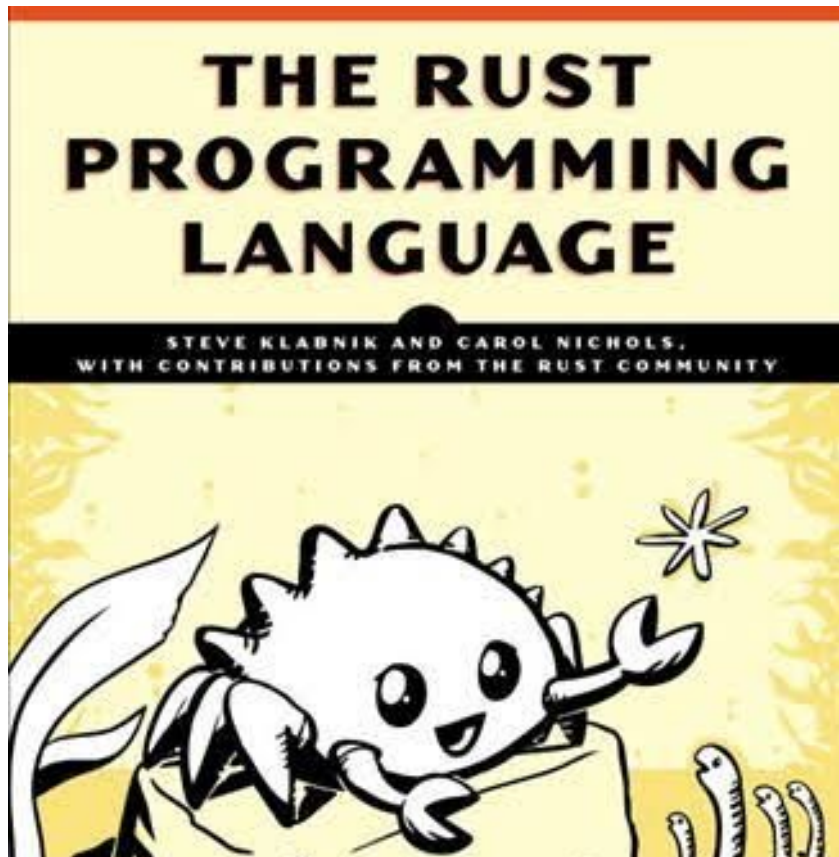
Rust, Interactively

- Rust has no top-level *a la* OCaml or Ruby
- There is an in-browser execution environment
 - <https://play.rust-lang.org/>



Rust Documentation

- Rust documentation is a good reference, and way to learn
 - <https://doc.rust-lang.org/stable/>
- This contains links to
 - the Rust Book (on which most of our slides are based)
 - the reference manual, and
 - short manuals on the compiler, cargo, and more



Rust Basics

Functions

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

Hello, world!

Let Statements

```
{  
  let x = 37;  
  let y = x + 5;  
  y  
} //42
```

```
{  
  let x = 37;  
  x = x + 5; //err  
  x  
}
```

```
{ //err:  
  let x:u32 = -1;  
  let y = x + 5;  
  y  
}
```

```
{  
  let x = 37;  
  let x = x + 5;  
  x  
} //42
```

```
{  
  let mut x = 37;  
  x = x + 5;  
  x  
} //42
```

```
{  
  let x:i16 = -1;  
  let y:i16 = x+5;  
  y  
} //4
```

Redefining a variable *shadows* it (like OCaml);
aim to avoid

Variables immutable by default; use **mut** to allow updates

Types inferred by default; optional annotations must be consistent (may override defaults)

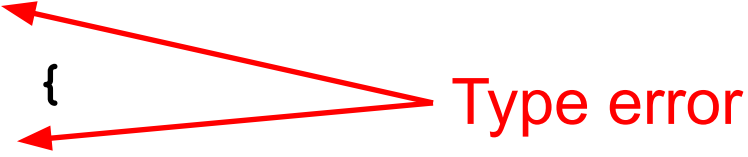
Conditionals

```
fn main() {  
    let n = 5;  
    if n < 0 {  
        print!("{}", is negative", n);  
    } else if n > 0 {  
        print!("{}", is positive", n);  
    } else {  
        print!("{}", is zero", n);  
    }  
}
```

5 is positive

Conditionals are *Expressions* (like OCaml)

```
fn main() {  
  let n = 5;  
  let x = if n < 0 {  
    10  
  } else {  
    "a"  
  };  
  print! ("{:?} | ", x);  
}
```



Type error

Factorial in Rust (recursively)

```
fn fact(n:i32) -> i32
{
    if n == 0 { 1 }
    else {
        let x = fact(n-1);
        n * x
    }
}

fn main() {
    let res = fact(6);
    println!("fact(6) = {}",res);
}
```

fact(6) = 720

Quiz: What does this evaluate to?

```
{ let x = 6;  
  let y = "hi";  
  if x == 5 { y } else { 5 };  
  7  
}
```

- A. 6
- B. 7
- C. 5
- D. Error

Quiz: What does this evaluate to?

```
{ let x = 6;  
  let y = "hi";  
  if x == 5 { y } else { 5 };  
  7  
}
```

- A. 6
- B. 7
- C. 5
- D. Error – if and else have incompatible types**

Quiz: What does this evaluate to?

```
{ let x = 6;  
  let y = 4;  
  y = x;  
  x == y  
}
```

- A. 6
- B. true
- C. false
- D. error

Quiz: What does this evaluate to?

```
{ let x = 6;  
  let y = 4;  
  y = x;  
  x == y  
}
```

- A. 6
- B. true
- C. false
- D. error – y is immutable**

Using Mutation

- Mutation is useful when performing iteration
 - As in C and Java

```
fn fact(n: u32) -> u32 {  
    let mut x = n;  
    let mut a = 1;  
    loop {  
        if x <= 1 { break; }  
        a = a * x;  
        x = x - 1;  
    }  
    a  
}
```

infinite loop
(break out)

Other Looping Constructs

- While loops
 - `while e block`
- For loops
 - `for pat in e block`
 - More later – e.g., for iterating through collections

```
for x in 0..10 {  
    println! ("{}", x); // x: i32  
}
```

Other Looping Constructs

- These (and `loop`) are *expressions*
 - They return the final computed value
 - `unit`, if none
 - **`break`** may take an expression, which is the loop's final value

```
let mut x = 5;
let y = loop {
    x += x - 3;
    println!("{}", x) ;// 7 11 19 35
    x % 5 == 0 { break x; }
};
print!("{}", y) ; //35
```

Quiz: What does this evaluate to?

```
let mut x = 1;  
for i in 1..6 {  
    let x = x + 1;  
}  
x
```

- A. 1
- B. 6
- C. 0
- D. error

Quiz: What does this evaluate to?

```
let mut x = 1;  
for i in 1..6 {  
    let x = x + 1;  
}  
x
```

- A. 1
- B. 6
- C. 0
- D. error

Data: Scalar Types

- Integers

- `i8`, `i16`, `i32`, `i64`, `isize`
 - `u8`, `u16`, `u32`, `u64`, `usize`

Machine word size

- Characters (unicode)

- `char`

- Booleans

- `bool` = { `true`, `false` }

Defaults (from inference)

- Floating point numbers

- `f32`, `f64`

- Note: arithmetic operators (+, -, etc.) *overloaded*

Compound Data: Tuples

- Tuples
 - n-tuple **type** (*t1*, ..., *tn*)
 - `unit ()` is just the 0-tuple
 - n-tuple **expression** (*e1*, ..., *en*)
 - Accessed by pattern matching or like a record field

```
let tuple = ("hello", 5, 'c');  
assert_eq! (tuple.0, "hello");  
let (x,y,z) = tuple;
```


Compound Data: Tuples

Distance between two points s and e

```
fn dist(s: (f64, f64), e: (f64, f64)) -> f64 {  
    let (sx, sy) = s;  
    let ex = e.0;  
    let ey = e.1;  
    let dx = ex - sx;  
    let dy = ey - sy;  
    (dx*dx + dy*dy).sqrt()  
}
```

Compound Data: Tuples

Can include patterns in parameters directly, too

```
fn dist2( (sx,sy) : (f64,f64) , (ex,ey) : (f64,f64) ) -> f64 {  
    let dx = ex - sx;  
    let dy = ey - sy;  
    (dx*dx + dy*dy).sqrt()  
}
```

We'll see Rust `structs` later. They generalize tuples.

Arrays: Standard Operations

- **Creating** an array (can be mutable or not)
 - But must be of fixed length
- **Indexing** an array
- **Assigning** at an array index

```
let nums = [1,2,3]; // type is [i32;3]
let strs = ["Monday", "Tuesday", "Wednesday"]; //[&str;3]
let x = nums[0]; // 1
let s = strs[1]; // "Tuesday"
let mut xs = [1,2,3];
xs[0] = 1; // OK, since xs mutable
let i = 4;
let y = nums[i]; //fails (panics) at run-time
```

Arrays: Iteration

- Rust provides a way to **iterate over a collection**
 - Including arrays

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

- `a.iter()` produces an **iterator**, like a Java iterator
 - This is a **method call**, a /a Java. More about these later
- The special **for** syntax issues the `.next()` call until no elements are left
 - No possibility of running out of bounds

Quiz: Will this function type check?

```
fn f(n: [u32]) -> u32 {  
    n[0]  
}
```

- A. Yes
- B. No

Quiz: Will this function type check?

```
fn f(n: [u32; len]) -> u32 {  
    n[0]  
}
```

- A. Yes
- B. No – because
array length not
known. Need to
fill in len**

Testing

- In any language, there is the need to test code
- In most languages, testing requires extra libraries:
 - Minitest in Ruby
 - Ounit in Ocaml
 - Junit in Java
- Testing in **Rust** is a first-class citizen!
 - The **testing framework** is built into **cargo**

Unit Testing In Rust

- Unit testing is for local or private functions
 - Put such tests [in the same file as your code](#)
- Use `assert!` to test that something is true
- Use `assert_eq!` to test that two things that implement the `PartialEq` trait are equal
 - E.g., integers, booleans, etc.
 - We'll explain [traits](#) later on

Unit Testing In Rust

This is a
module,
tests

```
fn bad_add(a: i32, b: i32) -> i32 {  
    a - b  
}  
  
#[cfg(test)]  
mod tests {  
    #[test]  
    fn test_bad_add() {  
        assert_eq!(bad_add(1,2), 3);  
    }  
}
```

Indicates that
this module
contains tests

Indicates
that this
function is
a test

Integration Testing In Rust

- **Integration testing** is for APIs and whole programs
- Create a **tests** directory
- Create different files for testing major functionality
- Files don't need **`#[cfg(test)]`** or a special module
 - But they do still need **`#[test]`** around each function
- Tests refer to code as if it were an external library
 - Declare it as an external library using **`extern crate`**
 - Include the functionality you want to test with **`use`**

Integration Testing In Rust

src/lib.rs

```
pub fn add(a: i32, b: i32) -> i32 {  
    a + b  
}
```

tests/test_add.rs

```
extern crate my-project-name;  
use my-project-name::add;  
#[test]  
pub fn test_add() {  
    assert_eq!(add(1,2), 3);  
}  
#[test]  
pub fn test_negative_add() {  
    assert_eq!(add(1,-2), -1);  
}
```

Running Tests

- `cargo test` runs all of your tests
- `cargo test s` runs all tests that contain *s* in the name
- By default, console output is hidden
 - Use `cargo test -- --nocapture` to un-hide it

Fun Fact

- The original Rust compiler was written in OCaml
 - Betrays the sentiments of the language's designers!
- Now the Rust compiler is written in ... Rust
 - How is this possible? Through a process called **bootstrapping**:
 - The first Rust compiler written in Rust is compiled by the Rust compiler written in OCaml
 - Now we can use the binary from the Rust compiler to compile itself
 - We discard the OCaml compiler and just keep updating the binary through self-compilation
 - So don't lose that binary! ☺