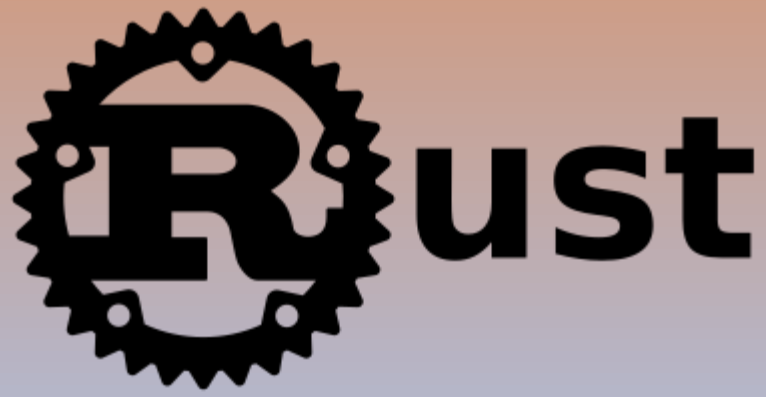

Go

Anh Dao

Jonathan Davis

CSC 4010 – 800

Dr. Martha J. Kosa

About Rust

- Rust is an open source systems programming language sponsored by the research division of Mozilla, a free-software community well known for the internet browser Firefox.
- The Rust programming language is statically typed and supports both the functional and procedural paradigms, with its procedural aspects being very similar to C++.
- The goal of the Rust Project is stated as being “To design and implement a safe, concurrent, practical systems language.”
- The creators of Rust wanted the language to have high concurrency and superior memory safety, while not sacrificing performance.

About Go

- Same as Rust, Go is an open source systems programming language, developed by Google.
- Go is strong, static typed, and it falls to imperative paradigms
- Goal of Go is to design a new programming language that would resolve common criticisms of other languages while maintaining their positive characteristics.

Rust: Hello World

```
fn main()
{
    //This will print "Hello World!" to the console.
    println!("Hello World!");

    /*
        To print a variable's string representation to the console,
        Use curly braces within your quotation marks, followed by the variable name.
    */
    let str = "Welcome to Rust!";
    println!("{}", str);
}
```

- println! Is a macro that prints text to the console
- A binary can be generated using the Rust compiler: rustc

Go: Hello World

```
package main
import "fmt"

func main() {
    fmt.Println("hello world")
}
```

- Go does has mechanisms to generate code (macro), but it’s not as easy as Rust
- A binary can be generated using the Go compiler: go
  - Build executable using: go build
  - Or directly run the compiled binary using: go run

Rust: Data Types

Scalar Primitives:

- Signed integers: i8, i16, i32, i64 and isize (pointer size)
- Unsigned integers: u8, u16, u32, u64 and usize (pointer size)
- Floating point: f32, f64
- Char: Unicode scalar values like 'a', ‘b’, or ‘c’ (4 bytes each)
- Bool: either true or false
- The Unit Type (), whose only possible value is an empty tuple: ()

Compound Primitives:

- Arrays: [1, 2, 3]
- Tuples: (1, true)

```
#[derive(Copy, Clone)]
struct Point
{
    x: f32,
    y: f32,
    z: f32,
}
```

```
#[derive(Copy, Clone)]
struct Plane
{
    p1: Point,
    p2: Point,
    p3: Point,
}
```

Go: Data Types

Scalar Primitives:

- Signed integers: int8, int16, int32, int64, uintptr (pointer size)
  - rune: alias for int32 and equivalent to int32
  - int: signed integer type that is at least 32 bits but not alias for any integer type
- Unsigned integers: uint8, uint16, uint32, uint64
- Floating point: float32, float64
  - complex64 and complex128 is set of all complex numbers with float32/64 real and imaginary part
- Bool: either true or false
- Nil: null type in Go

Compound Primitives:

- Arrays: [1, 2, 3]

Go: Data Types

- string: array of runes
- rune: is a single character

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0000 *0*	SOH 0001 *1*	STX 0002 *2*	ETX 0003 *3*	EOT 0004 *4*	ENQ 0005 *5*	ACK 0006 *6*	BEL 0007 *7*	BS 0008 *8*	HT 0009 *9*	LF 000A *10*	VT 000B *11*	FF 000C *12*	CR 000D *13*	SO 000E *14*	SI 000F *15*
1_	DLE 0010 *16*	DC1 0011 *17*	DC2 0012 *18*	DC3 0013 *19*	DC4 0014 *20*	NAK 0015 *21*	SYN 0016 *22*	ETB 0017 *23*	CAN 0018 *24*	EM 0019 *25*	SUB 001A *26*	ESC 001B *27*	FS 001C *28*	GS 001D *29*	RS 001E *30*	US 001F *31*
2_	SP 0020 *32*	! 0021 *33*	" 0022 *34*	# 0023 *35*	$ 0024 *36*	% 0025 *37*	& 0026 *38*	' 0027 *39*	(0028 *40*	) 0029 *41*	* 002A *42*	+ 002B *43*	, 002C *44*	- 002D *45*	. 002E *46*	/ 002F *47*
3_	0 0030 *48*	1 0031 *49*	2 0032 *50*	3 0033 *51*	4 0034 *52*	5 0035 *53*	6 0036 *54*	7 0037 *55*	8 0038 *56*	9 0039 *57*	: 003A *58*	; 003B *59*	< 003C *60*	= 003D *61*	> 003E *62*	? 003F *63*
4_	@ 0040 *64*	A 0041 *65*	B 0042 *66*	C 0043 *67*	D 0044 *68*	E 0045 *69*	F 0046 *70*	G 0047 *71*	H 0048 *72*	I 0049 *73*	J 004A *74*	K 004B *75*	L 004C *76*	M 004D *77*	N 004E *78*	O 004F *79*
5_	P 0050 *80*	Q 0051 *81*	R 0052 *82*	S 0053 *83*	T 0054 *84*	U 0055 *85*	V 0056 *86*	W 0057 *87*	X 0058 *88*	Y 0059 *89*	Z 005A *90*	[005B *91*	\ 005C *92*	] 005D *93*	^ 005E *94*	_ 005F *95*
6_	` 0060 *96*	a 0061 *97*	b 0062 *98*	c 0063 *99*	d 0064 *100*	e 0065 *101*	f 0066 *102*	g 0067 *103*	h 0068 *104*	i 0069 *105*	j 006A *106*	k 006B *107*	l 006C *108*	m 006D *109*	n 006E *110*	o 006F *111*
7_	p 0070 *112*	q 0071 *113*	r 0072 *114*	s 0073 *115*	t 0074 *116*	u 0075 *117*	v 0076 *118*	w 0077 *119*	x 0078 *120*	y 0079 *121*	z 007A *122*	{ 007B *123*	| 007C *124*	} 007D *125*	~ 007E *126*	DEL 007F *127*

Variable Bindings

- Rust provides type safety via static typing. (Same as Go)
- In Rust, variable bindings can be type annotated when declared. However, in most cases, the compiler will be able to infer the type of the variable from the context, heavily reducing the annotation burden.
- In Rust, values (like literals) can be bound to variables, using the *let* keyword. In Go, you just need to use *:=*
- In Rust, variable bindings are immutable by default, but this can be overridden using the *mut* modifier. In Go everything is mutable
- In Rust variable bindings have a scope, and are constrained to live in a block. A block is a collection of statements enclosed by curly braces. Also, variable shadowing is allowed.

Rust: Types & Conversion

- Rust provides no implicit type conversion (coercion) between primitive types. But, explicit type conversion (casting) can be performed using the “as” keyword.
- Rules for converting between integral types follow C conventions generally, except in cases where C has undefined behavior. The behavior of all casts between integral types is well defined in Rust.

```
let decimal: f32 = 65.4321;

// Error! No implicit conversion
let integer: u8 = decimal;

// Explicit conversion
let integer = decimal as u8;
let character = integer as char;
```

Go: Types & Conversion

- Same as Rust, Go provides no implicit type conversion (coercion) between primitive types.
- Go assignment between items of different types requires an explicit conversion.

```
package main
import (
    "fmt"
    "math"
)

func main() {
    var x, y int = 3, 5
    var f float64 = math.Sqrt(float64(x*x + y*y))
    var z uint = uint(f)
    fmt.Println(x, y, z)
}

--> 3, 5, 5
```

Rust: Literals and Type Inferencing

- Numeric literals can be type annotated by adding the type as a suffix.
- The type inference engine does more than just looking at the type of the r-value during an initialization; it also looks at how the variable is used afterwards to infer its type.

```
// element has type "u8" due to the suffixed literal
let element = 5u8;

// Compiler does not know the exact type of vec.
let mut vec = Vec::new();

// Compiler will now know vec is a vector of u8's.
vec.push(element);
```

Go: Literals and Type Inferencing

- When declaring a variable without specifying an explicit type (either by using the := syntax or var = expression syntax), the variable's type is inferred from the value on the right hand side. The type inference engine does more than just looking at the type of the r-value during an initialization; it also looks at how the variable is used afterwards to infer its type.
- When the right hand side of the declaration is typed, the new variable is of that same type:

```
var i int
j := i // j is an int
```

- When the right hand side contains an untyped numeric constant, the new variable may be an int, float64, or complex128 depending on the precision of the constant:

```
i := 42           // int
f := 3.142        // float64
g := 0.867 + 0.5i // complex128
```

Rust: Flow Control

- Branching with if-else is similar to other languages. Unlike many of them, the boolean condition doesn't need to be surrounded by parentheses, and each condition is followed by a block. if-else conditionals are expressions, and, all branches must return the same type.
- Rust provides a loop keyword to indicate an infinite loop. The break statement can be used to exit a loop at anytime, whereas the continue statement can be used to skip the rest of the iteration and start a new one.
- Rust also supports the use of for and while loops, with the former being compatible with iterators.

```
let mut n = 5;

while n > -6 {

    if n < 0 {
        println!("{} is negative", n);
        n = n - 1;
    }
    else if n > 0 {
        println!("{} is positive", n);
        n = n - 1;
    }
    else {
        println!("{} is zero", n);
        n = n - 1;
    }

}

// `n` will take the values: 1, 2, ..., 100 in each iteration
for n in 1..101 {
    println!("Loop number {}", n);
}
```

Go: Flow Control

- Go if-else syntax is similar to Rust.
- Also support switch cases
- Go only supports one looping construct *for* loop
- If omit the condition, it will be infinite loop

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

```
x := 1
if x < 3 {
    x += 1
}
```

Matching

- Rust provides pattern matching via the match keyword, which can be used like a C switch.
- Matching can be used with tuples, enums, pointers, structs, and more.
- Go doesn’t support pattern matching.

```
let number = 13;

match number
{
    // Match a single value
    1 => println!("One!"),

    // Match several values
    2 | 3 | 5 | 7 | 11 => println!("Prime Number!"),

    // Match an inclusive range
    13...19 => println!("Teen Number!"),

    // Handle the rest of cases
    _ => println!("Not a Special Number..."),
}
```

Rust: Functions

- Functions are declared using the “fn” keyword. Its arguments are type annotated, just like variables, and, if the function returns a value, the return type must be specified after an arrow ->.

- The final expression in the function will be used as return value. Alternatively, the return statement can be used to return a value earlier from within the function.

```
fn point_point_subtraction(p1: Point, p2: Point) -> Point
{
    let x = p1.x - p2.x;
    let y = p1.y - p2.y;
    let z = p1.z - p2.z;
    return Point {x: x, y: y, z: z};
}
```

Rust: Functions

- Methods are functions attached to objects. These methods have access to the data of the object and its other methods via the self keyword.
- Methods are defined under an impl block.

```
struct Point {
    x: f64,
    y: f64,
}

// Implementation block, all `Point` methods go in here
impl Point {
    // This is a static method
    // Static methods don't need to be called by an instance
    // These methods are generally used as constructors
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    // Another static method, taking two arguments:
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
}

fn main() {
    let rectangle = Rectangle {
        // Static methods are called using double colons
        p1: Point::origin(),
        p2: Point::new(3.0, 4.0),
    };
}
```

Rust: Functions

- Closures in Rust, also called lambdas, are functions that can capture the enclosing environment.
- The syntax and capabilities of closures make them very convenient for on the fly usage.
- Calling a closure is exactly like calling a function. However, both input and return types can be inferred and input variable names must be specified.
- Rust also supports the use of Higher Order Functions.
- Generics are also supported to generalize types and functionalities to broader cases.

```
fn is_triple_bool<F>(int: i32, some_closure: F) -> bool
    where F: Fn(i32) -> bool
{
        some_closure(int * 3)
}

fn is_even(n: i32) -> bool
{
    return n % 2 == 1;
}

fn main()
{
    fn function (i: i32) -> i32 { i + 1 };
    let closure_annotated = |i: i32| -> i32 { i + 1 };
    let closure_inferred  = |i     |           i + 1  ;

    //answer will be a bool with value true, because 2*3 is even.
    let answer = is_triple_bool(2, is_even);

    //answer will be a bool with value false, because 3*3 is odd.
    let answer = is_triple_bool(3, is_even);
}
```

Go: Functions

- Functions are declared using the *func* keyword. Its arguments are type annotated, just like variables, and, if the function returns a value, the return type must be specified after function name and variables
- The final expression in the function will be used as return value. Alternatively, the return statement can be used to return a value earlier from within the function.
- Functions in Go can return multiple values.

```
func point_point_subtraction(p1 Point, p2 Point) Point {
    x := p1.X - p2.X
    y := p1.Y - p2.Y
    z := p1.Z - p2.Z
    return Point {X: x,Y: y,Z: z}
}
```

Go: Methods

- Methods in Go are just Function with special receiver argument
- Function declare its arguments after function name, but method receiver argument list between the *func* keyword and the method name.

```
type Vertex struct {
    X, Y float64
}

// Abs method has a receiver of type Vertex named v.
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs())
}
```

Scoping Rules

- Scopes play an important part in ownership, borrowing, and lifetimes. That is, they indicate to the compiler when borrows are valid, when resources can be freed, and when variables are created or destroyed.
- Variables in Rust do more than just hold data in the stack: they also own resources, or memory in the heap.
- Rust enforces RAII (Resource Acquisition Is Initialization), so whenever an object goes out of scope, its destructor is called and its owned resources are freed.
- This behavior shields against resource leak bugs, so the user never has to manually free memory or worry about memory leaks.

Ownership and Moves

- Because variables are in charge of freeing their own resources, resources can only have one owner. This also prevents resources from being freed more than once.
- When doing assignments or passing function arguments by value, the ownership of the resources is transferred. In Rust-speak, this is known as a move.
- After moving resources, the previous owner can no longer be used. This avoids creating dangling pointers.

- #[derive(Copy, Clone] allows any Point to be copied if needed.

```
#[derive(Copy, Clone)]
struct Point
{
    x: f32,
    y: f32,
    z: f32,
}
```

- When computing the normal, two point-point subtractions must be performed; both subtractions need to use the value of the first point on the plane (plane.p1). How can we ensure that the normal function will still have ownership of plane.p1 after the first subtraction is done?

```
fn normal(plane: Plane) -> Point
{
    //n = (PB - PA) x (PC - PA)
    let pbpa: Point = point_point_subtraction(plane.p2, plane.p1);
    let pcpa: Point = point_point_subtraction(plane.p3, plane.p1);
    return cross_product(pbpa, pcpa);

}
```

Borrowing and Lifetimes

- Most of the time, we'd like to access data without taking ownership over it. To accomplish this, Rust uses a borrowing mechanism. Instead of passing objects by-value, objects can be passed by reference using the ampersand “&”.
- The compiler statically guarantees (via its borrow checker) that references always point to valid objects. That is, while references to an object exist, the object cannot be destroyed.
- A lifetime is a construct the compiler (also called the borrow checker) uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed.

Macros

- provides a powerful macro system that allows metaprogramming.
- Instead of generating a function call, macros are expanded into source code that gets compiled with the rest of the program.
- However, unlike macros in C and other languages, Rust macros are expanded into abstract syntax trees, rather than string preprocessing, so you don't get unexpected precedence bugs.

```
macro_rules! print_point
{
    ($point: ident) =>
    (
        print!("\n({}, {}, {})\n", $point.x, $point.y, $point.z);
    )
}
```

Error Handling

- simplest error handling mechanism in Rust is panic. It prints an error message, starts unwinding the task, and usually exits the program.

- An enum called Option<T> in the std library is used when absence is a possibility. It manifests itself as one of two "options":
    - Some(T): An element of type T was found
    - None: No element was found

- These cases can either be explicitly handled via match or implicitly with unwrap. Implicit handling will either return the inner element or panic.

```
// The commoner has seen it all, and can handle any gift well.
// All gifts are handled explicitly using `match`.
fn give_commoner(gift: Option<&str>) {
    // Specify a course of action for each case.
    match gift {
        Some("snake") => println!("Yuck! I'm throwing that snake in a fire."),
        Some(inner)   => println!("{}? How nice.", inner),
        None          => println!("No gift? Oh well."),
    }
}


// Our sheltered princess will `panic` at the sight of snakes.
// All gifts are handled implicitly using `unwrap`.
fn give_princess(gift: Option<&str>) {
    // `unwrap` returns a `panic` when it receives a `None`.
    let inside = gift.unwrap();
    if inside == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", inside);
}
```

```rust

#[derive(Copy, Clone)]
struct Line
{
    p1: Point,
    p2: Point,
}


fn point_point_subtraction(p1: Point, p2: Point) -> Point
{
    let x = p1.x - p2.x;
    let y = p1.y - p2.y;
    let z = p1.z - p2.z;
    return Point {x: x, y: y, z: z};
}


fn cross_product(p1: Point, p2: Point) -> Point
{
    //a x b (a cross b) =
    //a_y*b_z - a_z*b_y (the first component of a x b)
    //a_z*b_x - a_x*b_z (the second component of a x b)
    //a_x*b_y - a_y*b_x (the third component of a x b)

    let x = (p1.y * p2.z) - (p1.z * p2.y);
    let y = (p1.z * p2.x) - (p1.x * p2.z);
    let z = (p1.x * p2.y) - (p1.y * p2.x);

    return Point {x: x, y: y, z: z};
}

```

```go

type Line struct {
    p1 Point
    p2 Point
}

func point_point_subtraction(p1 Point, p2 Point) Point {
    x := p1.X - p2.X
    y := p1.Y - p2.Y
    z := p1.Z - p2.Z
    return Point {X: x,Y: y,Z: z}
}

func cross_product(p1 Point, p2 Point) Point {
    x := (p1.Y * p2.Z) - (p1.Z * p2.Y)
    y := (p1.Z * p2.X) - (p1.X * p2.Z)
    z := (p1.X * p2.Y) - (p1.Y * p2.X)

    return Point {X: x, Y: y, Z: z}
}

func dot_product(p1 Point, p2 Point) float32 {
    return (p1.X * p2.X) + (p1.Y * p2.Y) + (p1.Z * p2.Z)
}

func normal(plane Plane) Point {
    pbpa := point_point_subtraction(plane.p2, plane.p1)
    pcpa := point_point_subtraction(plane.p3, plane.p1)
    return cross_product(pbpa, pcpa)
}

func multiply_point(scalar float32, point Point) Point {
```


A Great Tutorial/Introduction to Rust and Go

- Rust: https://rustbyexample.com/index.html
- Go: https://tour.golang.org/