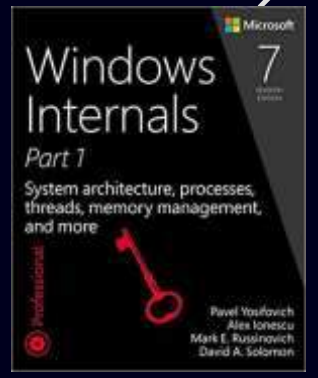
Rust Programming Fundamentals

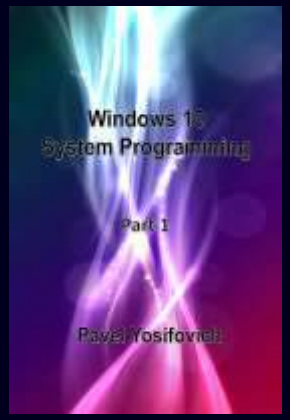
Pavel Yosifovich

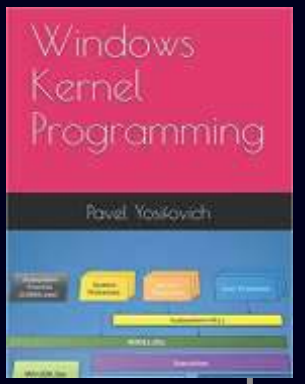
@zodiacon

About Me

- Developer, Trainer, Author and Speaker
- Co-author of “Windows Internals”, 7th edition, Part 1 (2017)
- Author
  - “Windows Kernel Programming” (2019)
  - “Windows 10 System Programming (Part 1)” (2020)
  - “WPF 4.5 Cookbook” (2012)
- Published courses on *Pluralsight* and *PentesterAcademy*
- Author of several open-source tools on *Github* (https://github.com/zodiacon)
- Website: http://scorpiosoftware.net







Agenda

- Module 1: Introduction to Rust
- Module 2: Language Fundamentals
- Module 3: Ownership
- Module 4: Compound Types
- Module 5: Common Collections
- Module 6: Managing Projects
- Module 7: Error Handling
- Module 8: Generics and Traits
- Module 9: Smart Pointers
- Module 10: Functional Programming
- Module 11: Concurrency
- Module 12: Advanced Topics

Introduction to Rust

MODULE 1

Agenda

- What is Rust?
- Installing and Updating Rust
- Hello, Rust!
- Using Cargo
- Working with Visual Studio Code
- Testing
- Summary

What is Rust?

- System programming language
  - Playing in the same playground as C/C++
  - Low level access possible
- High-level constructs
- Characteristics
  - Fast
  - Safe
  - Functional

(Some) Rust Characteristics

- Fast
  - Compiles to native code
  - No garbage collector
  - Most abstractions have zero cost
  - Most checks in compile time
- Safe
  - No uninitialized memory
  - No dangling pointers
  - No nulls
  - No double free errors
  - No manual memory management
  - All these even with multiple threads

Installing Rust

- Go to rust web site (https://www.rust-lang.org/)
- On Windows, download *Rustup-init.exe*
  - If installing on non-Windows, follow the instructions for your platform
  - Run it and select option 1 (proceed with installation)
  - Rust tools path added to the PATH environment variable
- New versions of Rust released every six weeks
- Three toolchains available at any time
  - Stable, Beta, Nightly
  - Stable installed initially by default

Updating Rust

- Open command window
- Run *rustup update* to update all installations
- Some useful *rustup* commands
  - *Rustup show*
    - Lists the installed toolchains
  - *Rustup default <toolchain>*
    - Set the default toolchain
  - *Rustup help*
    - Shows help for *rustup*
    - Append *help* to other commands to get detailed help

Useful Links

- Rust home page
- Official rust docs
- Rust playground
- The Rust book
- Rust by example
- Awesome Rust
- Rust GitHub repository

Hello, Rust!

- Open a text editor and type the following program

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

- Save the file as *hello.rs*
- Open a command window and navigate to the directory where the file is stored
- Type *rustc hello.rs*
- Type *hello* to execute the program

Some Rust Basics

- Language is case sensitive
- Functions are declared with the `fn` keyword
- Blocks of code are surrounded by curly braces { }
  - It's considered good style to put the open brace on the same line 
- Function names ending with ! are macros
  - Without the downsides of C/C++ macros!
- String literals surrounded by quotation marks " "

Hello, Cargo!

- Using the Rust compiler directly (*rustc*) is possible, but unlikely in real programs
- Rust has its own build system and package manager called *Cargo*
- Cargo is powerful and has many capabilities

New Project with Cargo

```
C:\temp>cargo new hello
     Created binary (application) `hello` package

C:\temp>cd hello

C:\temp\hello>dir

25-Feb-20  16:41    <DIR>          .
25-Feb-20  16:41    <DIR>          ..
25-Feb-20  16:41                 8 .gitignore
25-Feb-20  16:41               225 Cargo.toml
25-Feb-20  16:41    <DIR>          src
               2 File(s)            233 bytes
               3 Dir(s)  919,190,110,208 bytes free

C:\temp\hello>cd src

C:\temp\hello\src>dir

25-Feb-20  16:41    <DIR>          .
25-Feb-20  16:41    <DIR>          ..
25-Feb-20  16:41                45 main.rs
```

Cargo.toml

```
[package]
name = "hello"
version = "0.1.0"
authors = ["Pavel Yosifovich <zodiacon@live.com>"]
edition = "2018"

[dependencies]
```

main.rs

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

More Cargo Commands

- `cargo build`
  - Build the package
  - Resulting output in **`target\debug\`**
- `cargo build --release`
  - Build a release version of the package
  - Resulting output in **`target\release\`**
- `cargo run`
  - Run (build first if out of date)
- `cargo check`
  - Compiles without generating an executable
- `cargo test`
  - Run all tests in the package

Rust IDE

- Working with a text editor of some kind is fine for simple programs
- A full featured IDE is required for real work
  - Including a debugger
- There is (currently) no Rust-specific IDE
- Rust support is available in several IDEs/editors
  - Visual Studio Code, Sublime Text, IntelliJ Idea, Eclipse, Geany, …

Visual Studio Code as Rust IDE

- Rust support in Visual Studio Code is based around extensions
  - Like most things in VS Code
- Install Rust VS Code extensions
  - Rust
  - Rust rls (Rust Language Server)
  - Many others exist

Testing

- Rust supports unit testing out of the box
- Any function marked with the `#[test]` attribute is considered a test
  - Typically defined in a child module named *test*
- Run all tests with `cargo test`
  - More options available (add –h to list them)
- Integration tests are placed in their own directory named *tests*
  - Any contained file is a consumer of the crate

Summary

- What is Rust?
- Installing and Updating Rust
- Using Cargo
- VS Code as Rust IDE

Language Fundamentals

MODULE 2

Agenda

- Variables
- Mutability
- Fundamental Data Types
- Arrays
- Operators
- Functions
- Control Flow
- Scope
- Attributes
- Crates
- Summary

Variables

- Variables are named memory locations
- Variables are bound with the `let` keyword
  - Immutable by default
    - Add `mut` to make mutable
  - Bindings are implicitly typed
  - Types can be specified explicitly if needed

```
let x = 5;
let y: i32 = 10;
x += 1;     // error, x is immutable

let mut z = 2;
z += 1;     // OK
```

Variables and Mutation

- Variable bindings are immutable by default
- Helps in preventing certain kinds of errors
  - “Functional” style
- Mutation is sometimes a better choice
  - Copying large objects is expensive

Variable Shadowing

- A variable with the same name may be bound to a new value within the same scope with another `let`
  - Can be of different type
  - Hides the previous variable

```
println!("Enter a number: ");
let mut num = String::new();

std::io::stdin().read_line(&mut num).unwrap();

let num: i32 = num.trim().parse().unwrap();
```

Constants

- Constant values can be assigned with the `const` keyword
  - Type annotation is mandatory
  - Can be declared in any scope (including global scope)
  - Naming convention is ALL_CAPS
  - Must be set to constant expressions known at compile time

```
const SPEED_OF_LIGHT: f64 = 299792458.0;
```

Data Types

- Rust is a statically typed language
  - Every value has a type known at compile time
- Type annotations are in most cases optional

```
let n = "123".parse().unwrap();          // error
```

```
let n: u32 = "123".parse().unwrap();     // OK
```

Integer Data Types

Type	Description	Size (bytes)
`i8`	Signed 8 bit	1
`u8`	Unsigned 8 bit	1
`i16`	Signed 16 bit	2
`u16`	Unsigned 16 bit	2
`i32`	Signed 32 bit	4
`u32`	Unsigned 32 bit	4
`i64`	Signed 64 bit	8
`u64`	Unsigned 64 bit	8
`i128`	Signed 128 bit	16 (Rust 2018+)
`u128`	Unsigned 128 bit	16 (Rust 2018+)
`isize`	Signed platform sized integer	4 (32 bit), 8 (64 bit)
`usize`	Unsigned platform sized integer	4 (32 bit), 8 (64 bit)

More Data Types

- Floating point data types

Type	Description	Size (bytes)
`f32`	32-bit IEEE 754 floating point	4
`f64`	64-bit IEEE 754 floating point	8

- Other scalar data types

Type	Description	Size (bytes)
`bool`	Boolean (`true` or `false`)	1
`char`	Unicode Scalar Value	4

Literals

- Integer literal forms
  - Decimal (no specific prefix) e.g. `123, 45_129`
  - Hex (0x prefix) e.g. `0x2f`, `0x100_abcd`
  - Octal (0o prefix) e.g. `0o777`
  - Binary (0b prefix) e.g `0b1100_0101`
  - Byte (u8 type only) e.g. `b'a'`
- Underscores can be used within digits
- Boolean can be `true` or `false`
- Floating point values have period within a decimal number

Tuples

- Two or more values (of possibly different types) grouped together
- `let` can bind to a tuple
- Tuple destructuring
- Tuple values access

```
fn main() {
    let x = 10;
    let y = 4;
    let result = (x + y, x * y);
    let (sum, prod) = result;    // destructure

    println!("sum: {} product: {}", result.0, result.1);
    println!("sum: {} product: {}", sum, prod);
}
```

Arrays

- Array is a contiguous list of values of a specified type
  - Allocated “in situ”
  - Size cannot change (part of the type)
  - Element access with index in brackets [ ]

```
let numbers = [ 1, 3, 5, 7, 11 ];
let days = [ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
     "Friday", "Saturday" ];

println!("Today is {}", days[2]);
println!("Your number is {}", numbers[0]);
println!("There are {} days a week", days.len());

let a: [f32; 6] = [ 3.0, 5.0, 0.0, 2.6, 6.4, -1.0 ];
println!("last: {}", a[a.len() - 1]);

let zeros = [0; 10];   // [value; length]
println!("There are {} zeros!", zeros.len());
```

Array type is
`[ TYPE; SIZE ]`

Mathematical Operators

- Precedence
  - Multiplication, division, modulo
  - Addition, subtraction
- Can use parenthesis to change precedence

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)

Bitwise Operators

Operator	Description
&	AND
|	OR
^	XOR
~	NOT
>>	Shift right
<<	Shift left

OR truth table

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

XOR truth table

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

AND truth table

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Relational / Logical Operators

Operator	Description
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

- Relational operators return `true` or `false`

Operator	Description
&&	And
||	Or
!	not

- Precedence
  - Not
  - And
  - Or
  - Relational operators
- Logical operators use “short circuit” evaluation

```
let r1 = x > 7;
let r2 = y == 6 || x * 3 < 200;
let r3 = z >= y && !y < 10 || x % 2 == 0;
```

Functions

- Functions defined with the `fn` keyword
  - Naming convention is `snake_case`
  - Parameters must be typed
  - Return type is the `unit` type (“void”) unless otherwise specified
- Order of function definitions does not matter
  - There is no notion of “forward declarations”

Function Bodies

- Contain statements and end with an optional expression
- The `return` statement can be used to return prematurely (optionally with a value)
- Ending the function body with an expression (without a semicolon) is equivalent to returning that value
  - i.e. the semicolon means something: statement

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Control Flow - `if`

- `if` expressions
  - Check a Boolean value
  - Parenthesis not needed
  - `else` is optional
  - Multiple arms possible with `else if`
- If used with expressions, all arms must result in the same type
- Can be used as a way to achieve the functionality of the C/C++/C# ternary operator (? :)

Control Flow - `loop`

- `loop` expression
  - Runs a block as an infinite loop
  - Other ways to achieve this exist, but produce a warning
  - `break` statement allows exiting the block
    - Optionally returning a value

```
let mut counter = 0;
let result = loop {
    counter += 1;
    let value = do_work(counter);
    if counter == 10 {
        break value;
    }
};
println!("The result is {}", result);
```

```
fn do_work(counter: i32) -> i32 {
    counter * 2
}
```

Control Flow - `while`

- `while` statement
  - Executes the body while the condition is true

```
let mut i = 1;
while i <= 10 {
    let mut j = 1;
    while j <= 10 {
        print!("{:3} ", i * j);
        j += 1;
    }
    println!();
    i += 1;
}
```

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

Iterating with for

- The `for` statement can be used to iterate over a collection
  - Many ways to generate a collection

```
let numbers = [ 1, 3, 5, 7, 11 ];
for n in numbers.iter() {
    println!("{}", n);
}
```

```
// prints 0 to 9
for n in 0..10 {    // creates a std::ops::Range object
    println!("{}", n);
}
```

```
// prints 9 to 0
for n in (0..10).rev() {
    println!("{}", n);
}
```

Scope

- By default, only a few types are in scope
  - Provided by the *prelude*
- Using other types from the standard library can be done with their full name
  - The `use` statement can bring name(s) into scope

```
std::io::stdin().read_line(&mut num).unwrap();
```

```
use std::io;
…
io::stdin().read_line(&mut num).unwrap();
```

```
use std::io::stdin;
…
stdin().read_line(&mut num).unwrap();
```

Attributes

- Declarative annotations on items in Rust code
  - Mostly provided by the compiler infrastructure
- Syntax
  - `#[name]` – item-level attributes
  - `#![name]` – crate-level attributes
- Some attributes can have parameters
  - Examples
    - `#[cfg(test)]`
    - `#[derive(Copy, Clone, Debug)]`

Crates

- Rust packages are called *crates*
- Many are available through https://crates.io
- Add as dependency to *cargo.toml*

```
# cargo.toml

[dependencies]
rand = "0.7.3"
```

```
use rand::Rng;     // trait

//…
let secret = rand::thread_rng().gen_range(1, 100);
```

Summary

- Variables
- Mutability
- Fundamental Data Types
- Compound Types
- Operators
- Functions
- Control Flow
- Scope
- Crates

Ownership

MODULE 3

Agenda

- Ownership
- References
- Borrowing
- Slices
- Summary

Ownership

- Every object in Rust has a single owner (variable)
- When the owner variable goes out of scope, the object is destroyed (dropped)
- Ownership transfer is possible
  - This is the default when assigning variables
  - Unless the object implements the Copy trait

Copy vs. Move

```cpp
vector<int> v1{ 1, 2, 3 };
auto v2 = v1;
auto v3 = v1;

cout << v1.size() << " " << v2.size()
    << " " << v3.size() << endl;
```

```
3 3 3
```

```rust
let v1 = vec![1, 2, 3];
let v2 = v1;
let v3 = v1;
```

```
error[E0382]: use of moved value: `v1`
--> src\main.rs:18:14
|
16 |     let v1 = vec![1, 2, 3];
| --
whic    let v2 = v1.clone();
17 |     let v3 = v1.clone();
| --
18 |
| ^^
    println!("{} {} {}",
        v1.len(), v2.len(), v3.len());
```

Borrowing

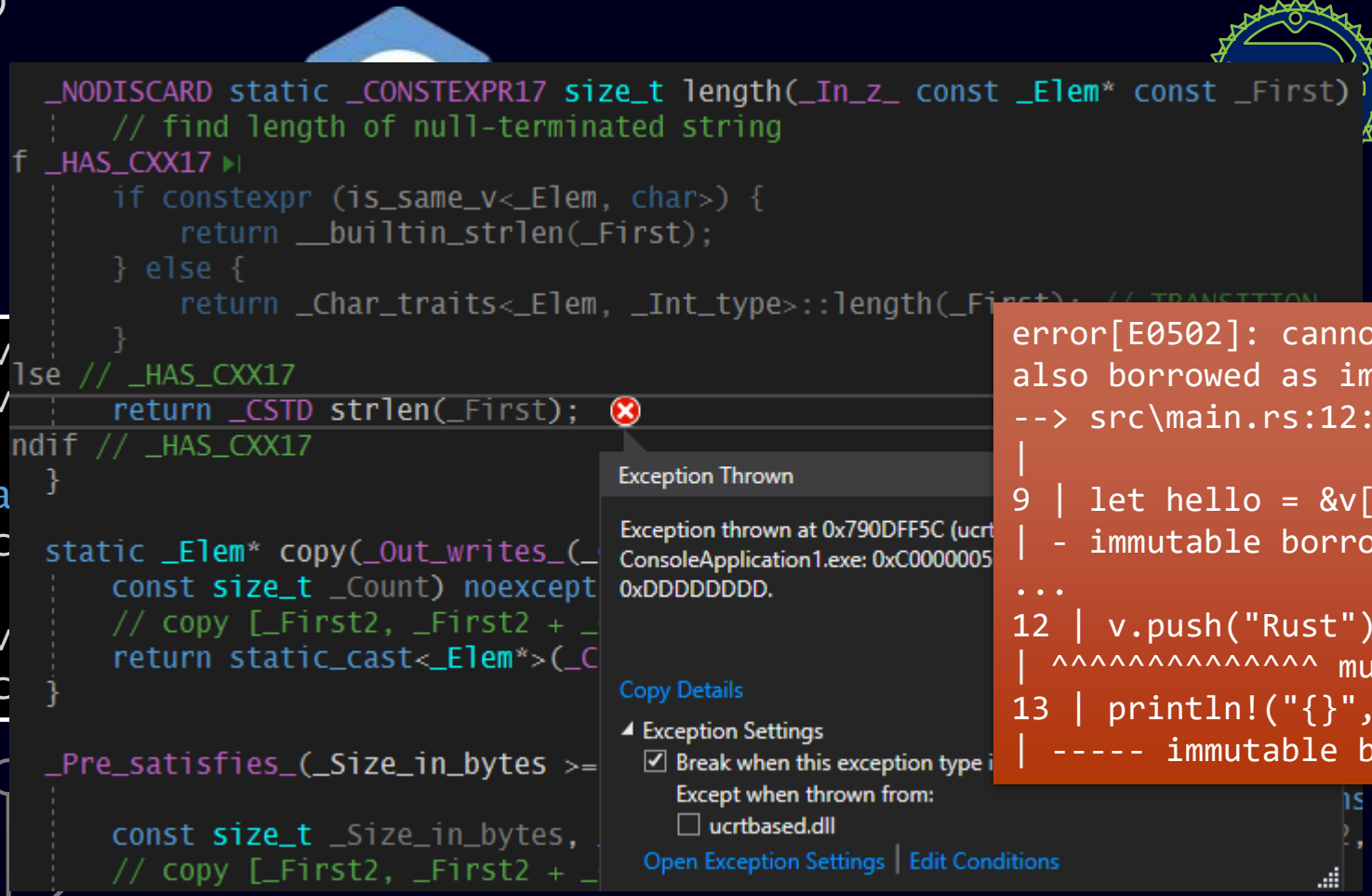
```
fn greet(s: String) {
}
```

```
fn greet(s: &String) {
    println!("Hello, {}!", s);
}
```

```
fn main() {
    let name = String::from("Pavel");
    greet(name);
    println!("Hello again, {}!", name);
}
```

```
error[E0382]: borrow of moved value: `name`
--> src\main.rs:41:31
 |
39 | let name = String::from("Pavel");
 | ---- move occurs because `name` has type `std::string::String`,
which does not implement the `Copy` trait
40 | greet(name);
 | ---- value moved here
41 | println!("Hello again, {}!", name);
 | ^^^^ value borrowed here after move
```

Ownership & Borrowing




```
error[E0502]: cannot borrow `v` as mutable because it is
also borrowed as immutable
--> src\main.rs:12:5
|
9 | let hello = &v[0];
| - immutable borrow occurs here
...
12 | v.push("Rust");
| ^^^^^^^^^^^^^^ mutable borrow occurs here
13 | println!("{}", hello);
| ----- immutable borrow later used here
```

Ownership Examples

- Simple types “implement” the Copy trait

```
let x = 7;
let y = x;
println!("{} {}", x, y);
```

- Complex types (that store data on the heap) typically do not

```
let s1 = String::from("hello");
let s2 = s1;
println!("{} {}", s1, s2);
```

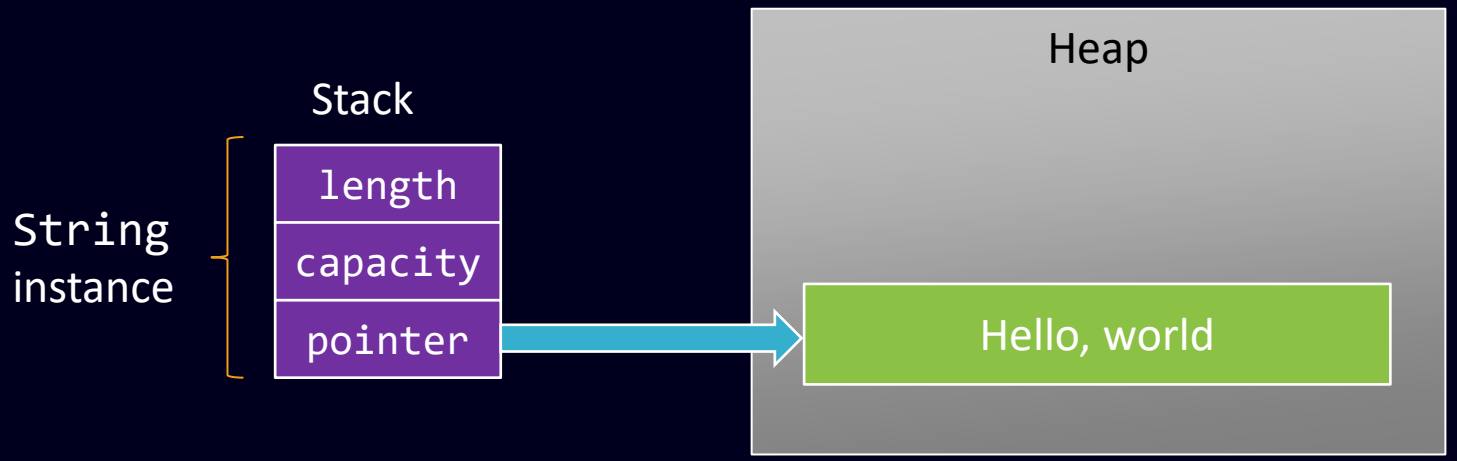
```
error[E0382]: borrow of moved value: `s1`
…
   |
9  |     let s1 = String::from("hello");
   |         -- move occurs because `s1` has type `std::string::String`, which does not
implement the `Copy` trait
10 |     let s2 = s1;
   |              -- value moved here
11 |
12 |     println!("{} {}", s1, s2);
   |                        ^^ value borrowed here after move
```

Stack vs. Heap

- Stack is a per-thread Last-In-First-Out (LIFO) memory
  - Pushing and popping operations are simple and fast
- Local variables declared on the stack
- Heap provides for dynamic memory allocation
  - Slower
  - Possibly long-lived

String Internals

- The `String` type points to its characters on the heap
- Copying would be potentially an expensive operation
- If copying is desired, call the `clone` method



The Copy Trait

- The `Copy` trait can be applied to types that do not implement the `Drop` trait
  - i.e. have nothing to clean up when the object is dropped
- Common types that have the Copy trait
  - All integer types, floating point types, char and bool
  - Tuples, if they only contain types that are also Copy
  - Arrays
- Common types that are *not* Copy
  - `String`, `Vec<>`, `HashMap<>`
- If unsure, check the documentation

Ownership and Functions

- Passing values to functions passes ownership as well
  - Unless type has the Copy trait
  - Typically, not what you want
- Returning values can transfer ownership as well

```
fn main() {
    let s = String::from("hello");
    do_something(s);

    println!("{}", s);
}

fn do_something(x: String) {
    println!("{}", x);
}
```

```
error[E0382]: borrow of moved value: `s`
 --> src\main.rs:7:20
  |
4 |     let s = String::from("hello");
  |         - move occurs because `s` has type
`std::string::String`, which does not implement the `Copy` trait
5 |     do_something(s);
  |                  - value moved here
6 |
7 |     println!("{}", s);
  |                     ^ value borrowed here after move
```

References

- The solution to the ownership transfer problem with function calls is to pass a reference to the object
  - Referred to as “borrowing”
- Use the ampersand with the parameter type and at the call site

```
fn main() {
    let s = String::from("hello");
    do_something(&s);

    println!("{}", s);
}

fn do_something(x: &String) {
    println!("{}", x);
}
```

References and Mutability

- References are immutable by default
- Just like variables, references can be mutable
- Reference mutability rule
  - If there is one mutable reference to an object, there can be no other references to the object (mutable or not)
  - There can be any number of immutable references to an object, as long as there is no mutable one

References Mutability Example

```rust
fn main() {
    let mut s = String::from("hello");
    do_something(&mut s);

    println!("{}", s);
}

fn do_something(x: &mut String) {
    x.push_str(", Rust!");
    println!("{}", x);
}
```

```rust
fn main() {
    let mut s = String::from("hello");
    let r = &s;
    do_something(&mut s);

    println!("{}", s);
    println!("{}", r);
}

fn do_something(x: &mut String) {
    x.push_str(", Rust!");
    println!("{}", x);
}
```

```
error[E0502]: cannot borrow `s` as mutable because it is also
borrowed as immutable
 --> src\main.rs:6:18
  |
5 |     let r = &s;
  |             -- immutable borrow occurs here
6 |     do_something(&mut s);
  |                  ^^^^^^ mutable borrow occurs here
...
9 |     println!("{}", r);
  |                    - immutable borrow later used here
```

Dangling References

- References should not point to objects that no longer exist
  - Common peril in other languages
  - The Rust compiler will not allow it

```
fn main() {
    let r = do_something();
    println!("{}", r);
}

fn do_something() -> &String {
    let s = String::from("hello");
    &s
}
```

```
error[E0106]: missing lifetime specifier
 --> src\main.rs:8:22
  |
8 | fn do_something() -> &String {
  |                      ^ help: consider giving it a 'static lifetime: `&'static`
  |
  = help: this function's return type contains a borrowed value, but there is no
value for it to be borrowed from
```

Slices

- Slices are references to contiguous sequence of elements
  - The elements are part of the actual object
  - Slices never have ownership
- Rust provides special syntax for using slices

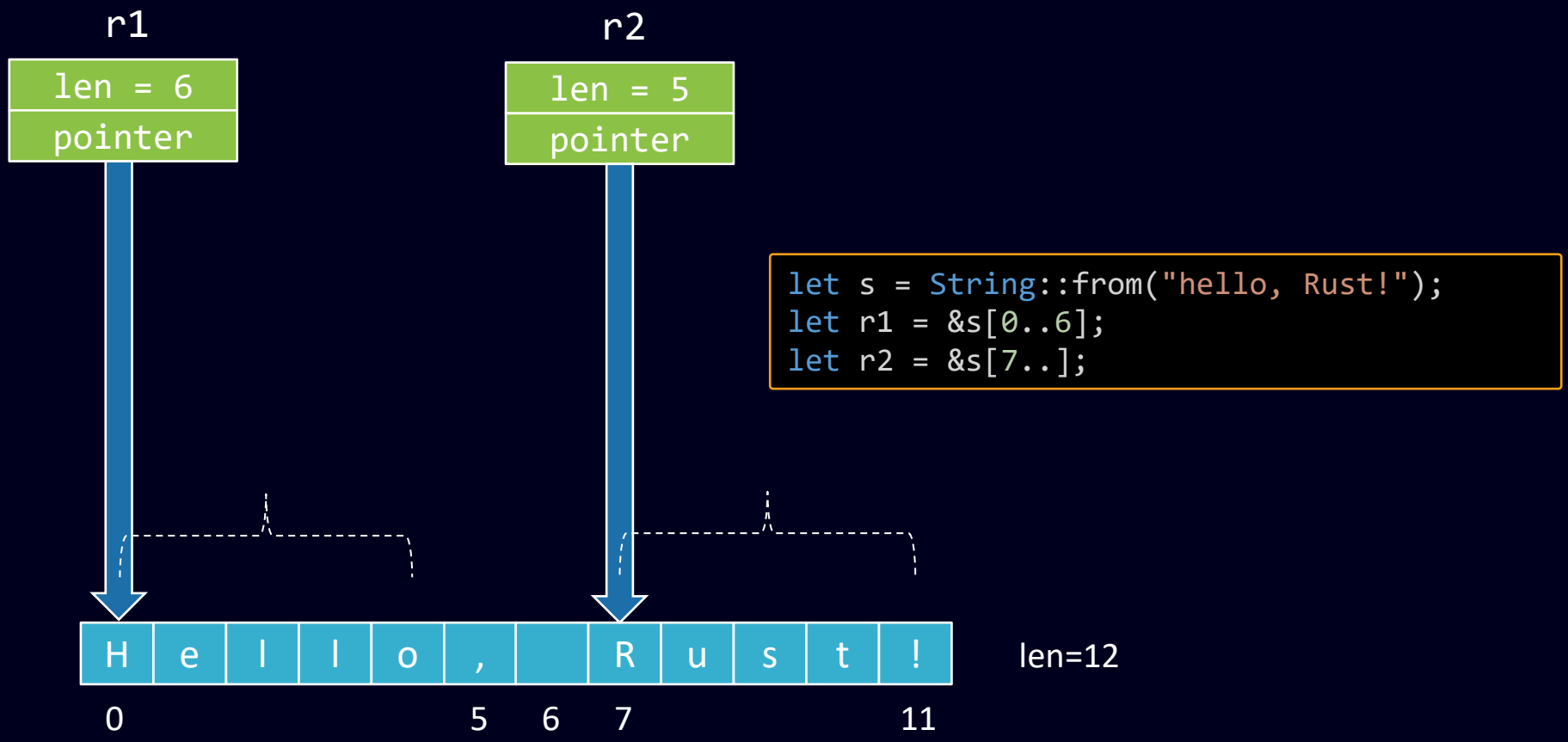
String Slices

- Reference part of a string

```
fn main() {
    let s = String::from("Hello, Rust!");
    let r1 = &s[0..6];
    let r2 = &s[7..];

    println!("{} {}", r1, r2);
}
```

- Slice syntax
  - `[start..end]` – `start` is included, `end` is not
  - Either or both indices can be omitted
- String slice type is `&str`

String Slices Internals




```
let s = String::from("hello, Rust!");
let r1 = &s[0..6];
let r2 = &s[7..];
```

String Slice Example

- Assuming ASCII string

```
fn main() {
    let s = String::from("Hello Rust!");
    let word = first_word(&s);

    println!("First word: {}", word);
}

fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
```

String Literals and Slices

- String literals are slices
- The strings themselves are part of the binary
  - Read only by definition
- String literal type is `&str`
- Read-only string parameters should be replaced with string slices

```
fn main() {
    let s = String::from("Hello Rust!");
    let word = first_word(&s[..]);
    let word2 = first_word(&s);
    let word3 = first_word("Bart Simpson");
    println!("First word: {}", word);
    println!("First word: {}", word2);
    println!("First word: {}", word3);
}

fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
```

Array Slices

- Arrays can also be referenced with slices
- Slice type is &[T]

```
fn main() {
    let data = [1, 44, 56, 78, 2, 33];
    let s1 = &data[1..3];
    let s2 = &data[2..];

    println!("{:?}", data);
    println!("{:?}", s1);
    println!("{:?}", s2);
}
```

```
[1, 44, 56, 78, 2, 33]
[44, 56]
[56, 78, 2, 33]
```

Summary

- Ownership
- References
- Borrowing
- Slices

Compound Types

MODULE 4

Agenda

- Structs
- Creating Objects
- Methods
- Enums
- Pattern Matching
- Summary

Structures

- Structures are the basic object-oriented building block in Rust
  - Naming convention is `PascalCase`
  - Defined with the `struct` keyword
  - Similar to struct/class in C++/C#/Java
- Structs contain fields (data members)

```
struct Book {
    name: String,
    author: String,
    year_published: i32,
    copies: u32
}
```

Instantiating Structs

- Instances of structs can be created
  - All fields must be initialized

```
let book1 = Book {
    name: String::from("Windows Internals 7th ed. part 1"),
    author: String::from("Pavel Yosifovich"),
    year_published: 2017,
    copies: 1000,
};
```

```
fn get_recent_book() -> Book {
    Book {
        name: String::from("Windows Kernel Programming"),
        author: String::from("Pavel Yosifovich"),
        year_published: 2019,
        copies: 1000,
    }
}
```

Shorthand Fields

- When a struct is instantiated
  - If field names and values are the same, the single name can be used

```
fn create_new_book(name: String, author: String, year: i32) -> Book {
    Book {
        name,
        author,
        year_published: year,
        copies : 1,
    }
}
```

Struct Update Syntax

- When copying values from one instance to another, fields that should have the same value can be omitted with a single syntactic addition

```
let book1 = Book {
    name: String::from("Windows Internals"),
    author: String::from("Pavel Yosifovich"),
    year_published: 2017,
    copies: 1000,
};
```

```
let book2 = Book {
    name: String::from("WPF 4.5 Cookbook"),
    author: String::from("Pavel Yosifovich"),
    year_published: book1.year_published,
    copies: book1.copies,
};
```

```
let book2 = Book {
    name: String::from("WPF 4.5 Cookbook"),
    author: String::from("Pavel Yosifovich"),
    ..book1
};
```

Tuple Structs

- Structs that look like tuples
- Have a name, but no field names
- Accessing fields with tuple syntax

```
struct Point(f32, f32, f32);
struct Color(u8, u8, u8, u8);

fn print_point(p: &Point) {
    println!("({},{},{})", p.0, p.1, p.2)
}
```

```
let black = Color(0, 0, 0, 255);
let red = Color(255, 0, 0, 255);
let origin = Point(0.0, 0.0, 0.0);
let xunit = Point(1.0, 0.0, 0.0);

print_point(&origin);
print_point(&xunit);
```

Displaying Struct Instances

- Using `println!` normally fails to print a struct's values

```
println!("{}", black);
```

```
error[E0277]: `Color` doesn't implement `std::fmt::Display`
  --> src\main.rs:33:20
   |
33 |     println!("{}", black);
   |                    ^^^^^ `Color` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Color`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
   = note: required by `std::fmt::Display::fmt`
```

- The Debug trait allows printing with {:?} format specifier

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

```
error[E0277]: `Color` doesn't implement `std::fmt::Debug`
```

- Need to add it explicitly

```
#[derive(Debug)]
struct Color(u8, u8, u8, u8);
```

Methods

- Methods are member functions that are part of compound types, such as structs
  - Method overloading is not supported in Rust
- Instance methods
  - First parameter is always `self`, representing the current instance
    - Called `this` in C++ and C#
    - Must be written explicitly
    - Can be in one of three forms: `self`, `&self`, `&mut self`
- Defined in an `impl` block
  - Multiple `impl` blocks are allowed

Instance Method Example

```rust
#[derive(Debug)]
struct Color {
    r: u8,
    g: u8,
    b: u8
}

impl Color {
    fn to_grayscale(&self) -> Color {
        let value = (0.3 * (self.r as f32) +
            0.59 * (self.g as f32) +
            0.11 * (self.b as f32)) as u8;
        Color {
            r: value,
            g: value,
            b: value
        }
    }
}
```

```rust
let red = Color {
    r: 255, g: 0, b: 0
};
println!("{:?} as grayscale: {:?}", red,
    red.to_grayscale());

let yellow = Color {
    r: 255, g: 255, b: 0
};
println!("{:?} as grayscale: {:?}", yellow,
    yellow.to_grayscale());
```

```
Color { r: 255, g: 0, b: 0 } as grayscale: Color { r: 76, g: 76, b: 76 }
Color { r: 255, g: 255, b: 0 } as grayscale: Color { r: 226, g: 226, b: 226 }
```

Associated Functions

- Called *static* methods in other OO languages
  - Not associated with any instance
- Don’t take `self` as first argument
  - Example: `String::from`
- Typically used for construction
  - The function name `new` is common for this purpose

Associated Functions Example

```
impl Color {
    fn new(r: u8, g: u8, b: u8) -> Color {
        Color { r, g, b }
    }

    fn from_grayscale(value: u8) -> Color {
        Color {
            r: value, g: value, b: value
        }
    }
}
```

```
let black = Color::new(0, 0, 0);
println!("black: {:?}", black);

let gray = Color::from_grayscale(128);
println!("gray: {:?}", gray);
```

Enumerations

- Enums are common in many programming languages
- However, they are much more powerful in Rust than in C++ or C#
  - Closer to a C/C++ union or algebraic data types in F#
- Simple enums are just a scoped list of values
  - Still very useful

```
enum Season {
    Winter,
    Spring,
    Summer,
    Fall
}
```

```
let january = Season::Winter;
```

More Enums

- Actual enum values can be set
  - Default is +1 from previous value
- Enum values can have extra data associated with them

```
enum Season {
    Winter = 1,
    Spring,
    Summer = 21,
    Fall
}
```

```
enum Command {
    RotateRight,
    RotateLeft,
    Rotate(f32),
    GoForward(f32),
    DoNothing
}
```

```
let commands = [ Command::RotateLeft, Command::GoForward(20.0) ];
for cmd in commands.iter() {
    execute_command(cmd);
}
```

Enum Implementation

- Enums can have `impl` bodies
  - Methods and associated functions

```
enum Command {
    RotateRight,
    RotateLeft,
    Rotate(f32),
    GoForward(f32),
    DoNothing
}

impl Command {
    fn is_active(&self) -> bool {
        *self != Command::DoNothing
    }
    fn rotate_to_zero(angle: f32) -> Command {
        Command::Rotate(-angle)
    }
}
```

```
let commands = [
    Command::RotateLeft,
    Command::DoNothing,
    Command::GoForward(20.0)
];

for cmd in commands.iter() {
    execute_command(cmd);
    if cmd.is_active() {
        println!("Active!");
    }
}
```

The Option<T> Enum

```
enum Option<T> {
    Some(T),
    None,
}
```

- Generic enum, included in the prelude
- Represents one of two options
  - Some value of type T - `Some(T)`
  - No value - `None`

```
let age = Some(10);
let name = Some("Bart Simpson");
let ageless: Option<i32> = None;
```

Extracting Values from `Option<>`

- Given an `Option<T>` object, the current value must be extracted somehow and used
- `Option<T>` has methods to help out
  - The `unwrap` method returns T if value is `Some(T)`
    - Otherwise – panics
  - `expect` returns T if value is `Some(T)`, otherwise displays the text provided and panics
- Pattern matching provides a convenient way to handle different values or patterns

The `Result<T,E>` Enum

- Recoverable errors are typically handled with the `Result` type
  - Brought into scope by the prelude

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- A function that may fail returns a `Result`
  - If successful, its value is `Ok(T)`
  - Otherwise, it's `Err(E)`
- Offers similar methods to `Option<T>`

The `match` Keyword

- Pattern matching expression
- All `match` *arms* must be covered
  - Use an underscore to indicate all other values

```
fn forecast(s: Season) -> String {
    let result = match s {
        Season::Winter => "Cold!!",
        Season::Summer => "Hot!!",
        _ => "Comfortable"
    };
    result.to_string()
}
```

```
fn execute_command(cmd : &Command) -> bool {
    match cmd {
        Command::RotateLeft | Command::RotateRight => {
            println!("Rotating for fun");
            true
        },
        Command::Rotate(angle) => {
            println!("Rotating {} degrees", angle);
            true
        },
        _ => {
            println!("Just hanging around...");
            false
        }
    }
}
```

`match` with `Option<T>`

- Any enum is a natural candidate for `match`
  - `Option<T>` happens to be a common case

```
fn birthday(age: Option<i32>) -> i32 {
    match age {
        Some(x) => x + 1,
        None => 0
    }
}
```

```
let age = Some(10);
let ageless: Option<i32> = None;

println!("Birthday {}", birthday(age));
println!("Birthday {}", birthday(ageless));
```

The `if let` Keywords

- In some cases, only one option from a match expression is of interest
  - `Option<T>` being a canonical example
- The `if let` keyword combination allows matching on a single pattern
  - else can be added as well (optional), synonymous with the "match all" underscore
- Syntax: `if let` <pattern> = <expression> `{...}`

if let Examples

```
let current_season = find_season("Australia");

if let Season::Winter = current_season {
    println!("Cold!!!");
}
```

```
fn birthday(age: Option<i32>) -> i32 {
    if let Some(x) = age {
        x + 1
    }
    else {
        0
    }
}
```

Summary

- Structs
- Creating Objects
- Methods
- Enums
- Pattern Matching

Compound Collections

MODULE 5

Agenda

- Vectors
- Strings
- Hash Maps
- Other Collections
- Summary

Vectors

- Dynamic array of objects from the same type (or subtype)
  - Elements contiguous in memory
- Type is `Vec<T>`
- Actual data stored on the heap
- New empty vector can be created with the `new` associated function

```
let data: Vec<i32> = Vec::new();      // type annotation might be needed
```

Working with Vectors

- The `vec!` Macro can create a vector and initialize it with values
  - Now type inference kicks in

```
let data = vec![3, 4, 5];
```

- Adding items at the end with the `push` method
- Accessing elements with indexing or the `get` method
  - Reference to the element is returned
  - Index is zero-based
  - `get` returns `Option<T>`

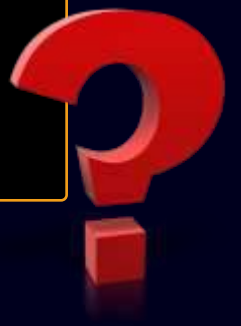
```
let mut data = vec![3, 4, 5];
data.push(6);
data.push(7);
data[3] += 2;

println!("{} {}", data[3], data.get(3).expect("error!"));
```

Accessing Vector Elements

- Any attempt to access non-existing elements causes the program to panic
  - `get` is convenient if this situation can be handled
- References must be explicitly requested when binding to variables (if the Copy trait is implemented by T)

```
let mut data = vec![3, 4, 5];
data.push(6);
data.push(7);

let mut m = data[3];
m += 1;
let mut n = &mut data[3];
*n = *n + 1;
```

```
let mut data = vec![3, 4, 5];
let n = &data[0];

data.push(6);
println!("{}", n);
```



Iterating Vectors

- The `for` statement can be used for easy iteration of vectors without using indices

```
let commands = [
    Command::RotateLeft, Command::DoNothing, Command::GoForward(20.0)
];
for cmd in &commands {
    execute_command(cmd);
}
```

```
let mut data = vec![3, 4, 5];
for n in &data {
    println!("{}", n);
}

for n in &mut data {
    *n += 1;
    println!("{}", n);
}
```

```
for n in data.iter() {
    println!("{}", n);
}

for n in data.iter_mut() {
    *n += 1;
    println!("{}", n);
}
```

Strings

- Every language/platform must have its own version of string ☺
- Rust strings are more complicated than most string types in other languages/platforms
- The core string type is `str` (string slices and string literals)
- The `String` type provided by the standard library is a mutable, owned, UTF-8 encoded string
- Other string types provided by the standard library include `OsString` (and `OsStr`), `CString` (and `CStr`), all from `std::ffi`
  - Needed for working with other languages/platforms

Creating Strings

- Empty string with `String::new`
- Initial string value with `String::from`
- Calling `to_string` on string literals or slices (`&str`) returns a true `String`
  - Available as part of the `Display` trait

```
let h1 = String::from("hello");
let h2 = String::from("שלום");

println!("{} = {}", h1, h2);
```

```
let mut s1 = String::new();
s1.push_str("Something");
s1 += " interesting";
```

Manipulating Strings

- `push` method appends a single character
- `push_str` method appends a string slice
- The `+` operator works on strings as well
  - Requires a `String` added to a slice
- Indexing of a `String` is not supported

```
let s1 = String::from("hello");
let s2 = " Rust!".to_string();
let s3 = s1 + &s2;

println!("{}", s3);
```

```
fn add(self, s: &str) -> String
```

String Formatting

- The `format!` Macro can be used to format a string in a similar manner to `println!`
  - Just returns the resulting string without printing anything
  - Does not take ownership of its arguments

```
let month = String::from("March");
let day = 10;
let year = 2020;
let date = format!("Today is {}, {}, {}", month, day, year);

println!("{}", date);
```

Some String Internals

- `String` is a wrapper over `Vec<u8>`
  - The `len` method returns the number of bytes (*not* characters)
  - Slicing with byte indices is allowed
    - Slice must be on a character boundary (otherwise panics)
    - Can check with `is_char_boundary`

```
let s1 = String::from("hello");
let s2 = String::from("שלום");
println!("s1: {}, s2: {}", s1.len(), s2.len());
```

```
s1: 5, s2: 8
```

String Iteration

- The `chars` method returns each `char` in the string
- The `bytes` method returns each byte in the string
  - May be appropriate for some scenarios

```
let s2 = String::from("שלום");
for ch in s2.chars() {
    println!("{}", ch);
}
```

```
ש
ל
ו
ם
```

```
let s2 = String::from("שלום");
for b in s2.bytes() {
    println!("{} 0x{:x}", b, b);
}
```

```
215 0xd7
169 0xa9
215 0xd7
156 0x9c
215 0xd7
149 0x95
215 0xd7
157 0x9d
```

Hash Maps

- Associative collection, mapping a key to a value
  - Keys must be unique
  - Hashing algorithm can be replaced on an instance basis
- `HashMap<Key, Value, S = RandomState>`
  - Not included in the prelude
  - Add `use std::collections::HashMap;`
  - No macro for creating and populating a `HashMap`

Initializing Hash Maps

- `HashMap::new` creates an empty HashMap
- The `insert` method adds a key/value pair
  - If the key already exists, it replaces the value
- Can be initialized from two vectors holding keys and values

```
let mut cities = HashMap::new();
cities.insert(String::from("United States"), String::from("Washington D.C."));
cities.insert(String::from("Israel"), String::from("Jerusalem"));
```

```
let countries = vec![
    String::from("United States"),
    String::from("Israel"),
];
let capitals = vec![
    String::from("Washington D.C."),
    String::from("Jerusalem"),
];

let cities: HashMap<_, _> = countries.iter().zip(capitals.iter()).collect();
```

Working with HashMap

```
let mut cities = HashMap::new();
cities.insert(String::from("United States"), String::from("Washington D.C."));
cities.insert(String::from("Israel"), String::from("Jerusalem"));
cities.insert(String::from("France"), String::from("Paris"));
cities.insert(String::from("Spain"), String::from("Madrid"));

for key in cities.keys() {
    println!("{}", key);
}

for value in cities.values() {
    println!("{}", value);
}

for (key, value) in &cities {
    println!("({},{})", key, value);
}

let countries = [ "France", "Germany", "Israel", "Portugal", "Spain" ];
for country in &countries {
    let capital = match cities.get(&country.to_string()) {
        Some(name) => name,
        None => "<unknown>"
    };

    println!("Capital of {}: {}", country, capital);
}
```

```
Israel
United States
Spain
France

Jerusalem
Washington D.C.
Madrid
Paris

(Israel,Jerusalem)
(United States,Washington D.C.)
(Spain,Madrid)
(France,Paris)
Capital of France: Paris
Capital of Germany: <unknown>
Capital of Israel: Jerusalem
Capital of Portugal: <unknown>
Capital of Spain: Madrid
```

More HashMap

- Keys and values are owned by the HashMap
- Accessing a value by key is possible with indexing
  - Panics if the key does not exist
- The entry API allows accessing an entry directly and manipulating/reading the value

```
cities.entry("Portugal".to_owned()).or_insert("Lisbon".to_owned());
```

```
let text = "hello rust world rust text in rust";
let mut map = HashMap::new();
for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}
println!("{:?}", map);
```

```
{"world": 1, "rust": 3, "in": 1, "hello": 1, "text": 1}
```

Customizing HashMap

- The default hashing algorithm can be replaced
  - Implement the `BuildHasher` trait
  - Some custom ones exist in https://crates.io/
- Custom key types must implement the traits `Eq`, `Hash` and `PartialEq`
  - In simple cases, a straight `#[derive]` is all it takes

```
#[derive(Hash, Eq, PartialEq, Debug)]
enum CardSuit {
    Spades, Clubs, Hearts, Diamonds,
}

#[derive(Hash, Eq, PartialEq, Debug)]
struct Card {
    value: i32,
    suit: CardSuit,
}
```

```
impl Card {
    fn new(value: i32, suit: CardSuit) -> Card {
        Card { value, suit }
    }
}
```

```
let mut card_map = HashMap::new();
card_map.insert(Card::new(6, CardSuit::Hearts), "6 of Hearts");
card_map.insert(Card::new(1, CardSuit::Spades), "Ace of Spades");
println!("{:?}", card_map);
```

```
{Card { value: 1, suit: Spades }: "Ace of Spades", Card { value: 6, suit: Hearts }: "6 of Hearts"}
```

Other Collections

- `VecDequeue<>`
  - Similar to `Vec<>`, but allows quick insertions and removal from the head of the vector
  - Uses a circular array internally
- `LinkedList<>`
  - Doubly-linked list
- `HashSet<>`
- `BTreeMap<,>`
- `BTreeSet<>`

Summary

- Vectors
- Strings
- Hash Maps

Managing Projects

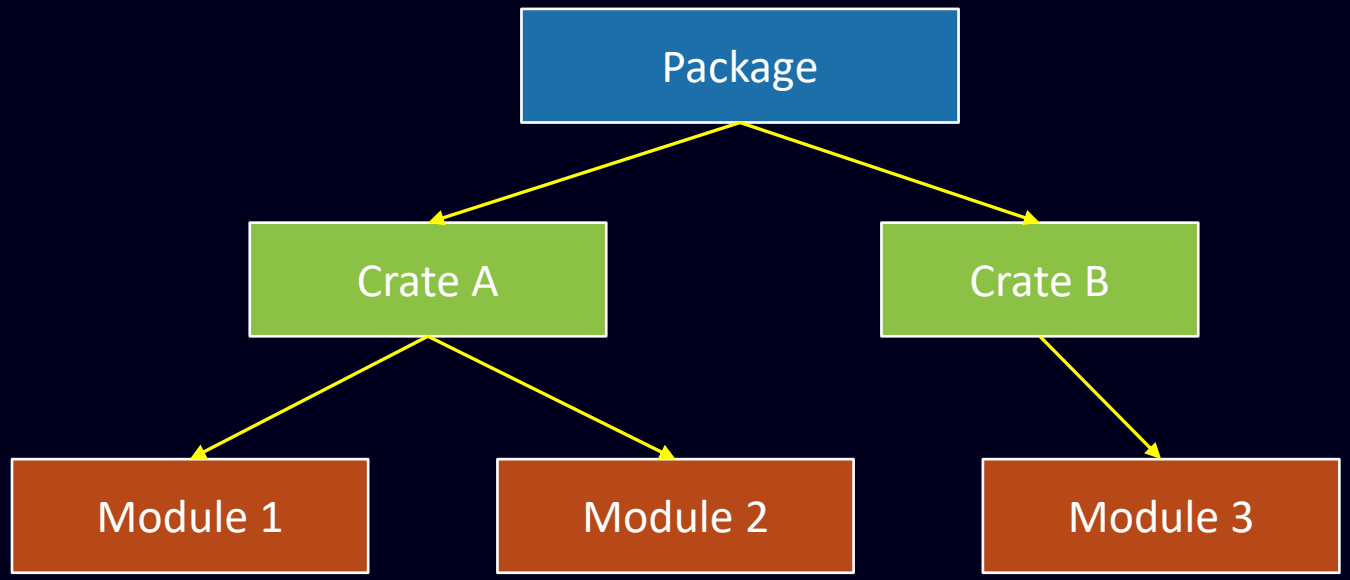
MODULE 6

Agenda

- Module System
- Packages and Crates
- Modules
- The `use` Statement
- Files as Modules
- Summary

Module System

- Rust uses a module system to manage projects



Packages and Crates

- A crate is a binary (application) or library
- A package is one or more crates
  - Contains a *Cargo.toml* file
  - Can contain at most one library crate (*`cargo new --lib <name>`*)
  - Can contain any number of binary crates
- Crate root is *src/main.rs* (binary crate) or *src/lib.rs* (library crate)
- Other binary crates in the package can be placed in the *src/bin* directory
  - Each file is a separate binary crate

General Package Layout

- Cargo is aware of this layout
- Default executable is *src/main.rs*
  - Other executables under *src/bin/*
- Run an example
  - `cargo run --example simple`
- Run a binary
  - `cargo run --bin app1`
- Many other options/configs available with cargo (read the *Cargo Book*)

```
.
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── app1.rs
│       ├── app2.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
├── benches/
│   ├── large-input.rs
│   └── multi-file-bench/
│       ├── main.rs
│       └── bench_module.rs
├── examples/
│   ├── simple.rs
│   └── multi-file-example/
│       ├── main.rs
│       └── ex_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.rs
```

Modules

- Organizational unit within a crate
- Files are modules by definition
- Modules can be public or private (private is the default)
- The `mod` keyword introduces a module
- Modules can be nested arbitrarily
- Modules can contain functions, constants, structs, enums, etc.

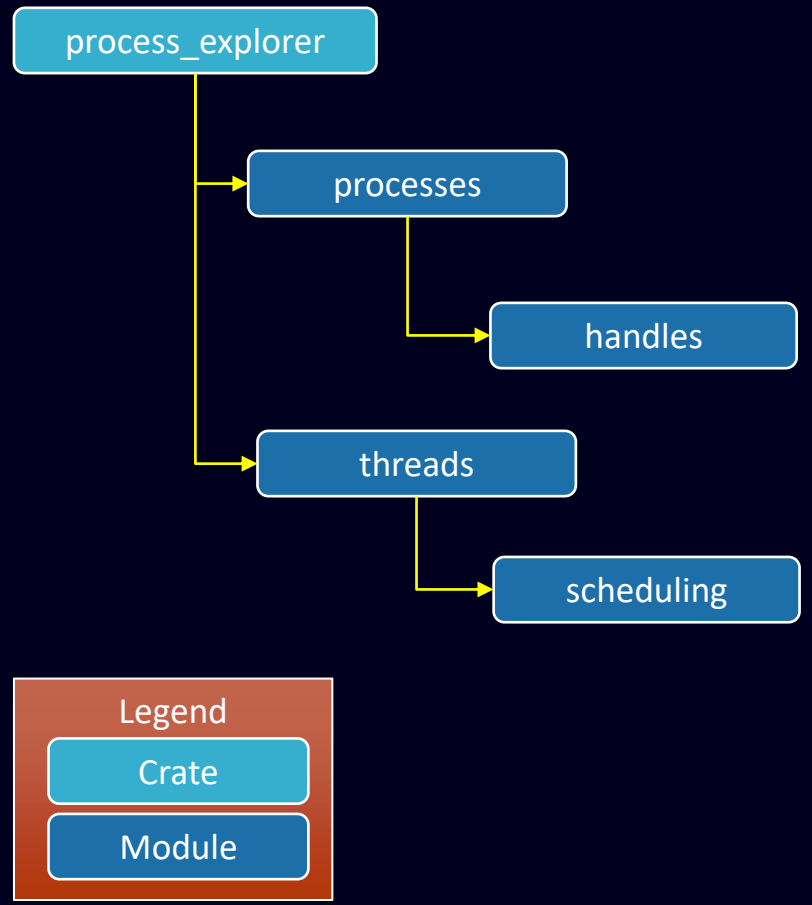
Files / Folders and Modules

- A binary crate is expected to have a *main.rs* file
- A library crate is expected to have a *lib.rs* file
- A directory that contains modules must have a *mod.rs* file
- Files are modules, but must be “included” in the relevant base files above
  - Using the `mod` keyword
- Modules have visibility, too

Library Crate Example

```
c:\> cargo new --lib process_explorer
```

```
pub mod processes {
    struct ProcessInfo {
    }
    pub fn enum_processes() -> Vec<ProcessInfo> {
        unimplemented!()
    }

    mod handles {
        use crate::processes::ProcessInfo;
        fn handle_count(process: &ProcessInfo) -> u32 {
            unimplemented!()
        }
    }
}

mod threads {
    struct ThreadInfo {
    }

    pub fn enum_threads(pid: u32) -> Vec<ThreadInfo> {
    }

    mod scheduling {

    }
}
```



Paths to Modules

- Absolute path
  - Starts with the crate's name or the keyword `crate`
- Relative path from the current module
  - Starts with the keyword `self` (current module), `super` (parent module) or an identifier in the current module
- Path parts separated by `::`

Elements Visibility

- Accessing identifiers from sibling modules or other crates requires them to be marked public (`pub`)
  - Private is the default (no keyword)
- Rust 2018 adds `pub(crate)`
  - Public to the crate, private outside the crate
- Modules themselves can be made public
  - Nested modules not accessible from parent module by default
- Anything that is an implementation detail should be hidden
- Child modules have access to private elements from parent modules

Type Visibility

- Structs and enums have visibility, too
  - Methods must be marked public, so they are accessible from parent modules
- Structs
  - Each function and field is private by default
    - Must be marked public to be accessible
- Enums
  - Fields are always public (accessible if the enum itself is public)

Bringing Paths into Scope

- Working with absolute paths for every access is verbose, inconvenient, and reduces readability
- The `use` keyword can bring modules into scope
  - Note that it's not the same behavior as the `using` keyword in C++ or C#!

Using use

- `use` can bring any public item into scope, at any level
- For functions it's better to bring the module rather than the full function name
  - Makes it clear the function is not local to the current module
- For structs and enums, sometimes it's OK to bring the full name into scope

```
use crate::processes::handles;

pub fn get_handles(pid: u32) -> u32 {
    handles::handle_count_pid(pid)
}
```

```
use crate::processes::handles::handle_count_pid;

pub fn get_handles(pid: u32) -> u32 {
    handle_count_pid(pid)
}
```

Naming Collisions with use

- Using `use` can bring potential name collisions
  - Same name from different modules
- One solution is to use parent module to differentiate
- Another is to alias names with the `as` keyword

```
use std::io;
use std::fmt;

fn do_something() -> fmt::Result {
    fmt::Result::Ok(())
}

fn do_something_else() -> io::Result<()> {
    io::Result::Ok(())
}
```

```
use std::io::Result as IoResult;
use std::fmt::Result;

fn do_something() -> Result {
    Result::Ok(())
}

fn do_something_else() -> IoResult<()> {
    IoResult::Ok(())
}
```

Re-exporting Names

- With the `use` statement, the name available in the current scope is private
- Adding `pub` to the `use` statement re-exports the name for external callers
- Useful when the internal structure of modules is not necessarily the best way to expose the functionality to outside callers

Using External Packages

- The https://crates.io site has many packages available for use
  - Add the dependency to *Cargo.toml*
  - Add a `use` keyword to bring names into scope
  - Package name is the root of the name hierarchy
- The standard library can be considered external as well
  - But no need to add it as a dependency

```
# cargo.toml

[dependencies]
rand = "0.7.3"
```

```
use rand::Rng;

//…
let secret = rand::thread_rng().gen_range(1, 100);
```

Semantic Versioning

- The package directory has a *Cargo.lock* file for maintaining the current version of external packages
- Semantic versioning
  - A version consists of 3 numbers (major, minor, build)
  - Related rules for upgrades
- By default, cargo uses the values from *Cargo.lock*
  - Run `cargo update` to update packages to the latest build
- To move to a new major or minor build, update *Cargo.toml*
  - Cargo will update *Cargo.lock* with the new version

More Options for use

- Nested paths can be used to avoid a long list of use statements
- The *Glob* operator can bring all public items into scope
  - Use with caution
  - Useful with testing

```
use std::io::*;
```

```
use std::io;
use std::cmp::Ordering;
```

```
use std::{io, cmp::Ordering};
```

```
use std::io;
use std::io::Write;
```

```
use std::io::{self, Write};
```

Separating Modules into Files

- The crate root (*src/lib.rs* or *src/main.rs*) is the only file examined by default
- Separating modules
  - Declare the module with the `mod` keyword
  - Move the module's code to a file with the same name
    - Remove the `mod` statement in the new file (if any)
    - A file is a module by definition

Refactored Modules

Processes.rs

```
pub struct ProcessInfo {
}

pub fn enum_processes() -> Vec<ProcessInfo> {
    Vec::new()
}

pub mod handles {
    use super::ProcessInfo;

    fn handle_count(process: &ProcessInfo) -> u32 {
        …
    }
    pub fn handle_count_pid(pid: u32) -> u32 {
        …
    }
}
```

lib.rs

```
mod processes;

mod threads {
    struct ThreadInfo {
    }

    use super::processes::ProcessInfo;

    fn enum_threads(process: &ProcessInfo) -> Vec<ThreadInfo> {
        Vec::new()
    }

    mod scheduling {
        use super::super::processes;

        fn do_work() {
            processes::enum_processes();
        }
    }
}
```

More About Cargo and Crates

- Cargo supports publishing to *crates.io*
- *crates.io* is the default *registry* for packages
  - Can add other registries through configuration
  - Files located by default in user's folder/*.cargo*
- Pre-compilation execution is possible with a "build" file
  - Named *build.rs* by default, located in the folder of *cargo.toml*
  - Technically, can do anything

Summary

- Module System
- Packages and Crates
- Modules
- The `use` Statement
- Files as Modules

Error Handling

MODULE 7

Agenda

- Types of Errors
- Panicking
- The `Result` Type
- Handling Result Objects
- When to Panic
- Summary

Types of Errors

- Recoverable errors
  - Failure to open a file
  - Failure to connect to a network resource
  - Bad input from the user
- Unrecoverable errors
  - Accessing an array element out of bounds
  - Failure to allocate object

Error Handling Options

- Return error code from functions
  - Common in C
- Exceptions
  - Common in C++, C#, Java, Python, …
- Rust's way
  - Unrecoverable errors crash the process
  - Recoverable errors can be handled by using a "special" return type

Panicking

- Panicking causes the process to report an error and crash
  - The `panic!` macro
  - Good for unrecoverable errors
- By default, unwinds the call stack, destroying objects in the process
- The full stack trace can be reported as well

Panicking Example

```rust
fn main() {
    let data = vec![ 12, 3, 4];

    println!("{}", data[12]);
}
```

```
C:\temp\hello> cargo run
Running `target\debug\hello.exe`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 12',
/rustc/...\src\libcore\slice\mod.rs:2791:10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\hello.exe` (exit code: 101)
```

```
stack backtrace:
   0: backtrace::backtrace::trace_unsynchronized
             at C:\Users\VssAdministrator\.cargo\registry\src\github.com-1ecc6299db9ec823\backtrace-
0.3.40\src\backtrace\mod.rs:66
   1: std::sys_common::backtrace::_print_fmt
             at /rustc/...\/src\libstd\sys_common\backtrace.rs:77
…
  15: alloc::vec::{{impl}}::index<i32,usize>
             at /rustc/...\src\liballoc\vec.rs:1883
  16: hello::main
             at .\src\main.rs:4
…
  24: std::rt::lang_start<()>
             at /rustc/...\src\libstd\rt.rs:67
  25: main
  26: invoke_main
             at d:\agent\_work\4\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:78
…
  28: BaseThreadInitThunk
  29: RtlUserThreadStart
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
error: process didn't exit successfully: `target\debug\hello.exe` (exit code: 101)
```

The Result Type

- Recoverable errors are typically handled with the `Result` type
  - Brought into scope by the prelude

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- A function that may fail returns a `Result`
  - If successful, its value is `Ok(T)`
  - Otherwise, it's `Err(E)`
- Can use any standard mechanism to handle

Handling a Result Examples(1)

```
use std::fs;
use std::io::Read;

fn main() {
    let file = fs::File::open("c:/temp/somedata.txt");

    let mut file = match file {
        Ok(file) => file,
        Err(error) => panic!("Error opening file: {:?}", error)
    };

    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();

    println!("{}", contents);
}
```

```
let mut file = fs::File::open("c:/temp/somedata.txt")
    .expect("error opening file");
```

Handling a Result Examples(2)

```
use std::fs;
use std::io::{ Read, ErrorKind };

fn main() {
    let filename = "c:/temp/somedata1.txt";
    let file = fs::File::open(filename);

    let mut file = match file {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match fs::File::create(filename) {
                Ok(file) => file,
                Err(error) => panic!("Failed to create file {:?}", error)
            },
            e => panic!("Other error: {:?}", e)
        }
    };
```

Ways of Handling Result

- The `expect` method
  - If error, prints a string and panics
- The `unwrap` method
  - Returns the `Ok` value if successful, panics otherwise
- The `unwrap_or_else` method
  - Provided a function to handle an error, otherwise returns the Ok value

Propagating Errors

- Code is typically constructed in layers
- Lower layers may encounter errors
- Higher layers know how to handle them
- Handling errors in lower layers is not recommended
  - How to handle such an error?
- It’s better for lower layers that identify an error to propagate it to higher layers that have a better chance of handling the error

Propagating Errors Example

```rust
use std::fs;
use std::io::Read;

fn read_contents(filename: &str) -> Result<String, std::io::Error> {
    let file = fs::File::open(filename);

    let mut file = match file {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut contents = String::new();
    match file.read_to_string(&mut contents) {
        Ok(_) => Ok(contents),
        Err(e) => Err(e)
    }
}
```

```rust
fn read_contents(filename: &str) -> Result<String, std::io::Error> {
    fs::read_to_string(filename)
}
```

```rust
fn read_contents(filename: &str) -> Result<String, std::io::Error> {
    let mut file = fs::File::open(filename)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

```rust
fn read_contents(filename: &str) -> Result<String, std::io::Error> {
    let mut contents = String::new();
    fs::File::open(filename)?.read_to_string(&mut contents)?;
    Ok(contents)
}
```

Using the ? Operator

- Can only be used in functions that return `Result`
- The `main` function can also return a `Result`

```
use std::io;
use std::fs;
fn main() -> Result<(), io::Error> {
    let text = read_contents("myfile.txt")?;
    println!("{}", text);
    Ok(())
}
```

```
use std::io;
use std::fs;
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
    let text = read_contents("myfile.txt")?;
    println!("{}", text);
    Ok(())
}
```

To Panic or Not to Panic

- When code panics, the process crashes
  - No way to handle the error
- If the error is unrecoverable anyway, let the process die
- Recoverable errors should always be handled in some meaningful way
  - In some cases, a specific `Err` may be too severe to handle and the code should panic
- Calling `unwrap` or `expect` for `Result` types is acceptable in tests and prototyping
  - And in some other cases that cannot possibly fail

Summary

- Types of Errors
- Panicking
- The `Result<>` Type
- Handling Result Objects
- When to Panic

Generics and Traits

MODULE 8

Agenda

- Generics
- Traits
- Working with Traits
- Basic Polymorphism
- Summary

Generics

- Generic types have already been used in the course
  - `Vec<>`, `HashMap<>`, `Result<>`, `Option<>`
- Helps with type safety and reduces code duplication
- Rust allows creating our own generic types and functions

Generic Functions

- Functions that could work on many potential types may be made generic

```rust
fn max(data: &[i32]) -> i32 {
    let mut largest = data[0];

    for &n in data {
        if n > largest {
            largest = n
        }
    }
    largest
}
```

```rust
fn max<T>(data: &[T]) -> T {
    let mut largest = data[0];

    for &n in data {
        if n > largest {
            largest = n
        }
    }
    largest
}
```


```
rror[E0369]: binary operation `>` cannot be applied to type `T`
36 |         if n > largest {
   |            - ^ ------- T
   |            |
   |            T
   |
   = note: `T` might need a bound for `std::cmp::PartialOrd`
```

Generic Structs

- Use angle brackets and identifier placeholder(s) for the generic type(s)
  - “T” is a common name (short for “Type”)

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
```

```
let p1 = Point { x: 3, y: 10};       // Point<i32>
let p2 = Point { x: 4.2, y: 5.0 };   // Point<f64>

println!("{:?}", p1);
println!("{:?}", p2);
```

```
let p3: Point<f32> = Point { x: 4.2, y: 5.0 };     // Point<f32>
```

Generic Enums

- Common examples: `Result<T,E>`, `Option<T>`

```
enum Option<T> {
    Some(T),
    None,
}
```

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
enum Command<T> {
    RotateRight,
    RotateLeft,
    Rotate(T),
    GoForward(T),
    DoNothing
}
```

Generic Struct Methods

- The `impl` block must specify the generic parameter(s)

```
impl<T> Point<T> {
    fn item(&self, i: i32) -> &T {
        match i {
            0 => &self.x,
            1 => &self.y,
            _ => panic!("no such element!"),
        }
    }
}
```

```
println!("({},{})", p1.item(0), p1.item(1));
```

Method Specialization

- It’s possible to specialize methods for specific types

```
impl Point<f64> {
    fn distance(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }
}
```

```
let p1 = Point { x: 3, y: 10};
let p2 = Point { x: 4.2, y: 5.0 };

println!("Distance: {}", p2.distance());
//println!("Distance: {}", p1.distance());
```

How Generics Work

- Generics code generation in Rust works similarly to C++ templates
  - Compiler generates code based on the concrete types used
  - May cause a lot of generated code, depending on the types and usage
- No extra cost at runtime

Traits

- Traits allow specification of shared functionality
  - Many similarities to interfaces in other languages
- Form the basis of attributes (`Derive`) and polymorphism
- May be implemented by structs or enums
- Some do not require any implementation, just `Derive` specification

Trait Definition

- Use the `trait` keyword
- Add method signatures without implementation
- Can add methods with a default implementation

```
trait Shape {
    fn x(&self) -> f64 {
        0.0
    }
    fn y(&self) -> f64 {
        0.0
    }

    fn area(&self) -> f64;
}
```

Trait Implementation

```
struct Rectangle {
    width: f64,
    height: f64,
}

impl Shape for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}
```

```
struct Circle {
    x: f64,
    y: f64,
    radius : f64,
}

impl Shape for Circle {
    fn x(&self) -> f64 {
        self.x
    }

    fn y(&self) -> f64 {
        self.y
    }

    fn area(&self) -> f64 {
        self.radius * self.radius * std::f64::consts::PI
    }
}
```

- Cannot call default implementation if overridden

Traits as Function Parameters (1)

- Traits can be passed to functions/methods
- Accept any type that implements the trait

```
fn print_shape(s: &impl Shape) {
    println!("Shape at ({},{}). Area: {}", s.x(), s.y(), s.area());
}
```

```
let r = Rectangle { width: 10.0, height: 6.0 };
let c = Circle::new(3.0, 8.0, 4.5);
print_shape(&r);
print_shape(&c);
```

```
Shape at (0,0). Area: 60
Shape at (3,8). Area: 63.61725123519331
```

Traits as Function Parameters (2)

- The `impl` prefix from the previous slide is a shorthand for a generic function
  - Convenient to use in many cases
  - However, sometimes it's too repetitive

```
fn print_shape(s: &impl Shape) {
    println!("Shape at ({},{}). Area: {}", s.x(), s.y(), s.area());
}
```

```
fn print_shape<T: Shape>(s: &T) {
    println!("Shape at ({},{}). Area: {}", s.x(), s.y(), s.area());
}
```

```
fn compare_shapes(s1: &impl Shape, s2: &impl Shape) {}
```

```
fn compare_shapes<T: Shape>(s1: &T, s2: &T) {}
```

Multiple Traits Constraints

- If a variable requires implementing more than one trait, concatenate with +
  - Also works with generic type constraints
- Alternatively, the `where` clause can be used

```
fn compare_shapes<T: Shape + Display>(s1: T, s2: T) { }
```

```
fn compare_shapes<T>(s1: T, s2: T) where T: Shape + Display { }
```

Returning Traits

- Functions can return a trait
  - Allows returning any concrete implementation of the trait
  - However, currently only a single type implementation must be returned

```
fn return_great_shape(i: i32) -> impl Shape {
    if i > 0 {
        Circle::new(0.0, 0.0, (i as f64) * 10.0)
    }
    else {
        Circle::new(0.0, 0.0, -i as f64)
    }
}
```

```
fn return_great_shape(i: i32) -> impl Shape {
    if i > 0 {
        Circle::new(0.0, 0.0, (i as f64) * 10.0)
    }
    else {
        Rectangle { width: -i, height: -i }
    }
}
```

The max Function Again

- One fix is to add a constraint on `PartialOrd` and `Copy`
- Other options
  - Constraint on `Clone` instead of Copy
  - Return a reference, so that neither Copy nor `Clone` are needed

```
fn max<T : PartialOrd + Copy>(data: &[T]) -> T {
    let mut largest = data[0];

    for &n in data {
        if n > largest {
            largest = n
        }
    }
    largest
}
```

Conditionally Implemented Methods

- With traits, it's possible to implement methods on types that adhere to certain traits

```
impl<T: PartialOrd + Copy> Point<T> {
    fn largest(&self) -> T {
        if self.x > self.y {
            self.x
        }
        else {
            self.y
        }
    }
}
```

```
let p1 = Point { x: 3, y: 10};
let p2 = Point { x: 4.0, y: 3.5 };
let p4 = Point { x : "hello", y: "rust" };
let p5 = Point { x: vec![1, 2, 3], y: vec![ 3, 4, 5 ]};

println!("{}", p1.largest());    // 10
println!("{}", p2.largest());    // 4.0
println!("{}", p4.largest());    // rust
// println!("{:?}", p5.largest());
```

Common Traits

Trait	Description
`Debug`	Debug formatting of objects with {:?}
`PartialEq`	Allows checking objects for equality and use the == and != operators
`Eq`	Has no methods. Indicates every value is equal to itself Can only be applied where `PartialEq` is applied
`PartialOrd`	Allows comparing instances with the <, >, <=, >= operators. Requires `PartialEq` as well Has `partial_cmp` method returning `Option<Ordering>`
`Ord`	Specifies that a valid order exist between any two values. The `cmp` method returns `Ordering`
`Clone`	Provides a `clone` method to deep copy an object
`Copy`	Shallow copies the value
`Hash`	Defines the `hash` method that allows hashing a value that can be used with (e.g.) `HashMap`
`Default`	Defines the `default` method that allows returning a default value for a type

Polymorphism

- Traits act as interfaces in classic OO
- Example

```
let r = Rectangle { width: 10.0, height: 6.0 };
let c = Circle::new(3.0, 8.0, 4.5);

let mut shapes: Vec<&dyn Shape> = vec![ &c, &r ];
let c2 = Circle::new(3.0, 5.6, 10.0);
shapes.push(&c2);

for s in shapes {
    println!("{}", s.area());    // polymorphic call
}
```

- Other options discussed in the next module

Implementing Traits on Non-Local Types

- Normally, traits implementation rule require the trait or the type be part of the local crate
- It's possible to circumvent this requirement by wrapping the type with a tuple struct
  - Called the *newtype pattern*
  - Wrapper removed by the compiler

```
H e l l o ,   R u s t !
```

```
use std::fmt;

struct StringWrapper(String);

impl fmt::Display for StringWrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        self.0.chars().for_each(|ch| write!(f, "{} ", ch).unwrap());
        Ok(())
    }
}
```

```
fn main() {
    let s = StringWrapper(String::from("Hello, Rust!"));
    println!("{}", s);
}
```

Summary

- Generics
- Traits
- Working with Traits
- Basic Polymorphism

Smart Pointers

MODULE 9

Agenda

- What is a Smart Pointer?
- The Box<> Type
- The Drop Trait
- Reference Counting
- Summary

What is a Smart Pointer?

- A pointer just points to a piece of memory or object
  - References are pointers
- Smart pointers provide management over some data
  - Common in other languages such as C++
  - Examples: `String`, `Vec<>`
- Smart pointers are structs that typically implement the `Deref` and `Drop` traits
  - `Deref` allows treating the smart pointer as a simple reference
  - `Drop` allows the smart pointer to do whatever cleanup is needed when it's destroyed

Box<T>

- `Box<>` is the simplest smart pointer, holding some data on the heap
  - Essentially the same as the C++ `std::unique_ptr<>`
- Common uses for `Box<>`
  - Recursive data structures
  - Transferring ownership of a big chunk of data without copying
  - Polymorphism

Recursive Data Structures

- Example: singly linked list

```
enum LinkedList {
    Value(i32, LinkedList),
    Null
}
```

- Cannot compile – recursive definition is unbounded

```
enum LinkedList {
    Value(i32, Box<LinkedList>),
    Null
}
```

```
use crate::LinkedList::{ Value, Null };

fn main() {
    let list = Value(3, Box::new(Value(3, Box::new(Value(7, Box::new(Null))))));
}
```

Using a Box<>

```
let b1 = Box::new(5);
println!("{}", b1);

//assert_eq!(5, b1);
assert_eq!(5, *b1);
```

```
struct Rectangle {
    width: i32,
    height: i32,
}

impl Rectangle {
    fn new(width: i32, height: i32) -> Self {
        Rectangle { width, height }
    }

    fn area(&self) -> i32 {
        self.width * self.height
    }
}
```

```
let r1 = Box::new(Rectangle::new(4, 8));
println!("{}", r1.area());
println!("{}", (*r1).area());

assert_eq!(32, r1.area())
```

Example: Our Own Box

- For demonstration purposes, here is a simple custom Box type

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(value: T) -> MyBox<T> {
        MyBox(value)
    }
}
```

- However, accessing the data is problematic

```
let b2 = MyBox::new(6);

println!("{}", *b2);
assert_eq!(6, *b2);
```

The Deref Trait

- The `std::ops::Deref` trait is implemented by smart pointers
  - Allows access to the underlying data using the dereference operator *
  - Has a single method: `deref`
- A related feature, *deref-coercion*, allows using the smart pointer without explicitly specifying the dereference operator for arguments to functions or methods

Implementing Deref

- For our custom Box

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.0
    }
}
```

```
let b2 = MyBox::new(6);
println!("{}", *b2);
assert_eq!(6, *b2);
```

- Without the `Deref` trait, only & references can be dereferenced
  - `*b2` is translated to `*(b2.deref())`

Deref Coercion

- Provides dereferencing transparency when passing smart pointers to functions or methods

```
fn print_greet(s: &str) {
    println!("Hi, {}!", s);
}
```

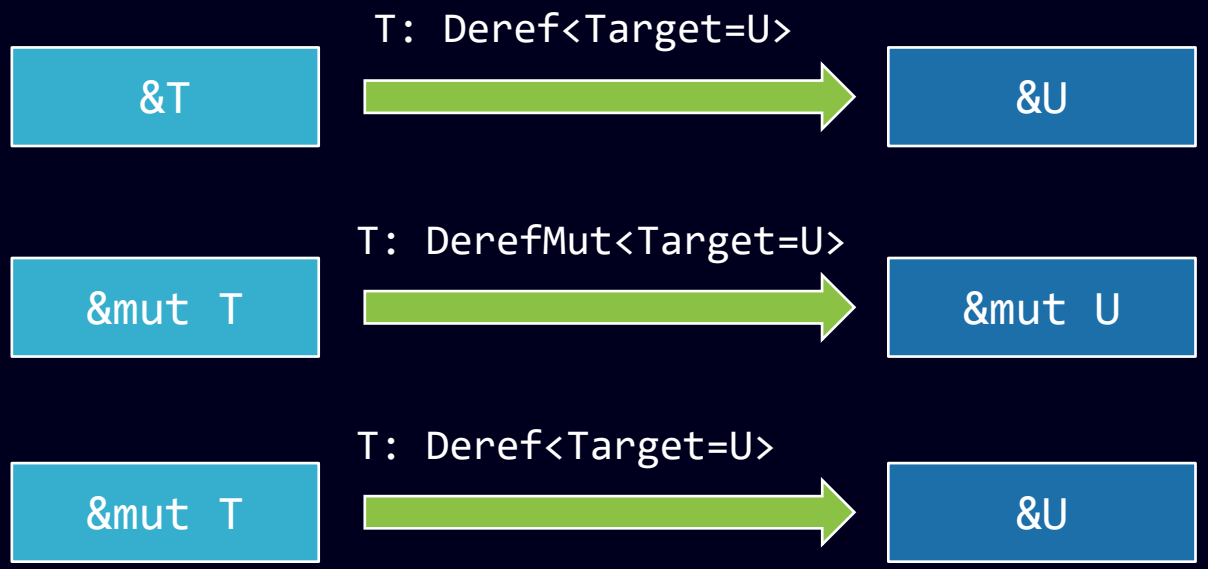
```
let b3 = MyBox::new(String::from("Pavel"));
print_greet(&b3);
```

- String also supports the Deref trait
  - Returns a string slice

```
print_greet(&(*b3)[..]);
```

Dereferencing and Mutability

- For mutable reference, the `DeferMut` trait is defined
- Compiler dereference rules



The Drop Trait

- The `Drop` trait allows code to be executed before an object is dropped (destroyed)
  - Same as a destructor in C++
- Has one method (`drop`), accepting `&mut self`
- For Box<T>, it's the chance to free the memory allocated on the heap when the Box was constructed
- Calling drop manually is not allowed
  - If needed, call `std::mem::drop` on the object
    - Imported by the prelude

Reference Counting

- The `Box<>` type is a single owner of its data
  - The normal case in Rust
- Sometimes multiple owners are required
  - Example: multiple clients sharing an object
    - Object should die when all clients are done using it
- Rust provides the `Rc<T>` and `Arc<T>` types
  - `Arc<>` is thread safe (atomic)
  - Similar to the C++ `std::shared_ptr<>` type

Using Rc<T>

- Create with the `new` associated function
- When sharing, call `clone`
  - Does not clone the actual data, just increments the reference count
  - Count decremented when that particular `Rc<>` goes out of scope

```
use std::rc::Rc;
fn main() {
    let s = Rc::new(String::from("hello"));
    println!("RC: {}", Rc::strong_count(&s));
    let r = s.clone();
    println!("RC: {} {}", Rc::strong_count(&s), Rc::strong_count(&r));

    let data = Data { name: s.clone() };
    println!("RC: {}", Rc::strong_count(&s));
    drop(r);
    println!("RC: {}", Rc::strong_count(&s));
    println!("{}", data.name);
}

struct Data {
    name: Rc<String>,
}
```

```
RC: 1
RC: 2 2
RC: 3
RC: 2
```

Weak References

- The `Weak<T>` type can be used to hold a “weak” reference to an object
  - Does not prevent the object from dying
  - Used mostly to prevent cycles
  - Similar to the C++ `std::weak_ptr<>` type
- Call `Rc<T>::downgrade` to get a `Weak<T>`
- When access is required from a `Weak<T>`, call `Weak<T>::upgrade`
  - Returns `Option<Rc<T>>`

Summary

- The Box<> Type
- The Drop Trait
- Shared Ownership

Functional Programming

MODULE 10

Agenda

- Function Programming Style
- Anonymous Functions
- Closures
- Iterators
- Summary

Functional Programming Style

- Functional programming style has a few principles
  - Computation as evaluation of functions
  - Functions are first class citizens (functions as data)
    - Can be passed to other functions or returned from functions
  - Pure functions
  - Immutability
  - Declarative style
  - Abstract over operations, rather than types

Anonymous Functions

- Functions with no name
  - Arguments between two bars/pipes
  - Curly braces optional if single expression/statement
- Can be stored in variables
- Invoked when needed
- Can be passed to functions

```
fn main() {
    let f = |x| x * 2;

    println!("{}*2={}", 5, f(5));
    println!("{}*2={}", 10, f(10));
    // println!("{}*2={}", 3.5, f(3.5));
}
```

Function Traits

- All anonymous functions implement at least one of three traits: `Fn`, `FnMut` or `FnOnce`
- Function types are generic types with trait(s)

```
fn transform<T : Fn(i32)->i32>(x: i32, g: T) -> i32 {
    println!("Applying transform on {}...", x);
    g(x)
}
```

```
let f = |x| x * 2 ;
let f2 = |x| x * x ;

println!("{} transformed is {}", 7, transform(7, f));
println!("{} transformed2 is {}", 7, transform(7, f2));
```

```
fn half(x: i32) -> i32 {
    x / 2
}
```

```
println!("{} transformed3 is {}", 7, transform(7, half));
```

Structs with Anonymous Functions

- Structs can have fields that are functions as well

```
struct Calculator<T : Fn(i32, i32)->i32> {
    fun : T,
}

impl<T> Calculator<T> where T : Fn(i32, i32) -> i32 {
    fn new(f: T) -> Self {
        Calculator { fun: f }
    }

    fn do_calc(&self, x: i32, y: i32) -> i32 {
        (self.fun)(x, y)
    }
}
```

```
let mul = |x, y| x * y;
let c = Calculator::new(mul);
for n in 1..10 {
    println!("calc with {},{}: {}", n, n + 1, c.do_calc(n, n + 1));
}
```

```
calc with 1,2: 2
calc with 2,3: 6
calc with 3,4: 12
calc with 4,5: 20
calc with 5,6: 30
calc with 6,7: 42
calc with 7,8: 56
calc with 8,9: 72
calc with 9,10: 90
```

Closures

- A closure is an anonymous function that has access to the environment
  - “captures” the environment

```
fn main() {
    let z = 3;

    let f = |x| x + z;

    println!("{}", f(3));
    println!("{}", f(7));
    println!("{}", transform(4, f));
}

fn transform<T : Fn(i32)->i32>(x: i32, g: T) -> i32 {
    println!("Applying transform on {}...", x);
    g(x)
}
```

```
fn main() {
    let z = 3;

    fn f2(x: i32) -> i32 {
        x + z
    }
```

Capturing with Closures

- How does a closure capture the environment?
  - `FnOnce` (always implemented)
    - Captures the environment once
    - Can’t take ownership of the variables
    - Can be called only once
  - `Fn`
    - Borrows variables immutably (no move)
  - `FnMut`
    - Borrows variables mutably (no move)
- Normally inferred automatically
- Add `move` to move ownership to the closure

Moving Ownership

```
let mut z = 3;

let f = |x| x + z;

println!("{}", f(3));
z += 1;
println!("{}", f(3));
```

```
let mut z = 3;

let f = move |x| x + z;

println!("{}", f(3));
z += 1;
println!("{}", f(3));
```

```
6
6
```

```
let z = vec![1, 2, 3];

let f = move |x| x == z;

println!("{}", f(vec![2, 3, 4]));

//println!("{}", z[0]);
```

Iterators

- The iterator pattern allows performing operations on sequences of items
- Iterators are lazily evaluated
  - “dereferenced” only when operation invoked
- Iterators implement the `Iterator` trait

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

     // more methods with default implementation
}
```

Iterating

- Sequences provide the `iter` method that returns an iterator
  - Examples: `Vec<>`, `String`
  - The classic `for` loop calls the `next` method on the iterator to get the next item
- The `iter_mut` allows iterating over mutable references
- The `into_iter` method returns an iterator that returns owned values, rather than references
- Some types implement the `IntoIterator` trait, converting to an iterator without calling any method explicitly
  - `into_iter` method
  - Canonical example: `Vec<>`

Consuming Iterators

- The `Iterator` trait defines many methods that have default implementations
  - Call `next` to consume values as needed

```
let z = vec![1, 33, 5, 22, 80, 23, 44, 3];
let sum: i32 = z.iter().sum();

println!("sum: {}", sum);
```

Iterator Adaptors

- Some methods on `Iterator` return new iterators

```
let z = vec![1, 33, 5, 22, 80, 23, 44, 3];

let double = z.iter().map(|x| x * 2);
let even = z.iter().filter(|x| x % 2 == 0);

for n in double {
    print!("{} ", n);
}
println!();

for n in even {
    print!("{} ", n);
}
println!();
```

```
2 66 10 44 160 46 88 6
22 80 44
```

(Some) Iterator Adapters

Adapter method	Description
`map`	Projection
`filter`	Filter items
`for_each`	Do something with each item (does not return anything)
`filter_map`	Combines `filter` and `map`
`enumerate`	returns tuples of index and the item
`take`	Returns the number of items specified
`skip`	Skips the given number of items
`skip_while`	Skip items while a condition is true
`take_while`	Take items while a condition is true
`map_while`	Project while a condition is true
`collect`	Builds a collection

More on Iterators

- Iterators are consumed when used
- The `collect` method on an iterator can return a collection (such as `Vec<>`)
- It's possible to create new iterators by implementing the `Iterator` trait
  - The `next` method will suffice

Summary

- Anonymous Functions
- Closures
- Iterators

Concurrency

MODULE 11

Agenda

- Multithreading Basic
- Rust and Concurrency
- Threads
- Message Communication
- Sharing Data
- The Send and Sync traits
- Summary

Multithreading Basics

- Running multiple threads concurrently can have performance benefits
  - Practically all systems today are multi-core
- A *thread* is an instance of a function executing independently
- A multithreaded application can result in errors related to concurrency, such as data races
- Data race
  - Data accessed by multiple threads concurrently, where at least one thread is writing

Rust and Concurrency

- It turns out that the ownership model also protects from data races
- Rust attempts to be as close to the platform as possible
  - Rust threads are mapped 1:1 to OS threads
- Higher level abstractions are available as external crates
  - Example: the *rayon* crate

Creating threads

- The `std::thread::spawn` function creates a new thread of execution
  - Returns `std::thread::JoinHandle<>`
  - Call `Join` to wait for a thread to finish
    - Returns `Err` if the thread panics
    - Otherwise, returns a result (can be unit)
- When the main (first) thread is finished, it automatically kills all other threads

Thread Example

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let t = thread::spawn(|| {
        println!("thread ID: {:?}", thread::current().id());
        for i in 1..=10 {
            println!("thread says {}", i);
            thread::sleep(Duration::from_millis(1));
        }
        println!("thread done!");
    });

    println!("main thread ID: {:?}", thread::current().id());
    for i in 1..=10 {
        println!("Main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }

    t.join().unwrap();
}
```

```
main thread ID: ThreadId(1)
Main thread: 1
thread ID: ThreadId(2)
thread says 1
Main thread: 2
thread says 2
thread says 3
Main thread: 3
Main thread: 4
thread says 4
Main thread: 5
thread says 5
thread says 6
Main thread: 6
thread says 7
Main thread: 7
thread says 8
Main thread: 8
Main thread: 9
thread says 9
thread says 10
Main thread: 10
thread done!
```

Thread Panic

- If a thread other than the main thread panics, `JoinHandle<>::join` returns `Err<>`
- If handled, does *not* cause the process to crash

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let t = thread::spawn(|| {
        println!("new thread!!! with ID {:?}", thread::current().id());
        thread::sleep(Duration::from_secs(1));
        panic!("panicking!!!");
    });
    let tid = t.thread().id();

    println!("main thread: waiting for work to complete");

    if let Err(_) = t.join() {
        println!("thread {:?} panicked!", tid);
    }
    println!("Don't panic!");
}
```

```
main thread: waiting for work to complete
new thread!!! with ID ThreadId(2)
thread '<unnamed>' panicked at 'panicking!!!', src\main.rs:8:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
thread ThreadId(2) panicked!
Don't panic!
```

Thread Customization

- A thread abstraction is somewhat limited in terms of customization
- Some aspects can be customized using the `std::thread::Builder` type
  - Stack size (current default is 2 MB)
  - Name (String, has no runtime effect, useful for debugging)
- No control on other platform-specific aspects unless unsafe calls are made to the underlying platform API
  - E.g. thread priority

Threads and Ownership

- Using closures to capture outer scope variables cannot work
  - Fails compilation
  - Thread lifetime not tied to the outer scope's lifetime
- Solution: use `move` closure to transfer ownership

```
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        println!("vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

```
error[E0373]: closure may outlive the current function, but it borrows
`v`, which is owned by the current function
 --> src\main.rs:5:32
  |
5 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
6 |         println !("vector: {:?}", v);
  |                                   - `v` is borrowed here
  |
```

```
let v = vec![1, 2, 3];
let handle = thread::spawn(move || {
    println!("vector: {:?}", v);
});
```

Example: Parallel Primes Counter (1)

```rust
fn is_prime(n: u32) -> bool {
    let limit = f32::sqrt(n as f32) as u32;
    for i in 2..=limit {
        if n % i == 0 {
            return false;
        }
    }
    true
}
```

```rust
fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() < 3 {
        println!("Usage: count_primes <from> <to> [threads]");
        return;
    }
    let from: u32 = args[1].parse().unwrap();
    let to: u32 = args[2].parse().unwrap();
    let threads: u32 = if args.len() > 3 { args[3].parse().unwrap() } else { 1 };
    let now = std::time::Instant::now();
    let count = count_primes(threads, from, to);
    let duration = std::time::Instant::now() - now;
    println!("Count: {} Elapsed: {} msec", count, duration.as_millis());
}
```

Example: Parallel Primes Counter (2)

```
fn count_primes(threads: u32, from: u32, to: u32) -> u32 {
    let mut handles = Vec::new();
    let chunk = (to - from + 1) / threads;
    for i in 0..threads {
        let start = from + chunk * i;
        let end = if i == threads - 1 { to } else { start + chunk - 1 };
        let handle = thread::spawn(move || {
            let mut count = 0;
            for n in start..=end {
                if is_prime(n) {
                    count += 1;
                }
            }
            count
        });
        handles.push(handle);
    }
    let mut total = 0;
    for h in handles {
        total += h.join().unwrap();
    }
    total
}
```

Thread Communication

- Rust's standard library provides a message passing mechanism in the `std::sync::mpsc` module

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let text = "Hello, Message!".to_string();
        thread::sleep(Duration::from_secs(2));
        tx.send(text).unwrap();
    });

    let result = rx.recv().unwrap();
    println!("Received: {}", result);
}
```

Message Passing with mpsc

- "Multiple Producers, Single Receiver"
- Add producer by cloning the sender (transmitter)
- Send message with `Sender<T>::send`
  - Non-blocking
- Receive with `Receiver<T>::recv` (blocking) or `try_recv` (non-blocking)

try_recv Example

```rust
use std::thread;
use std::sync::mpsc;
use std::time::Duration;
use std::io::Write;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let messages = vec![ "Hello!", "This", "is", "a", "long", "message" ];
        for msg in messages {
            thread::sleep(Duration::from_secs(1));
            tx.send(msg).unwrap();
        }
        tx.send("q").unwrap();
    });
    loop {
        match rx.try_recv() {
            Ok("q") => break,
            Ok(value) => println!("Received: {}", value),
            _ => {
                print!(".");
                std::io::stdout().flush().unwrap();
                thread::sleep(Duration::from_millis(200));
            }
        }
    }
}
```

```
.....Received: Hello!
.....Received: This
.....Received: is
.....Received: a
.....Received: long
....Received: message
```

Atomic Operations

- One form of thread safety is with atomic operations
  - Variable value is seen before or after the change, no way to "interfere" in between
- The `std::sync::atomic` module provides simple atomic types
  - `AtomicBool`, `AtomicIsize`, `AtomicUsize`, `AtomicI32`, …
- Mostly work using compiler intrinsics
- Always prefer these to "heavier" synchronization if possible

Atomics Example

```
fn count_primes(threads: u32, from: u32, to: u32) -> u32 {
    let mut handles = Vec::new();
    let chunk = (to - from + 1) / threads;
    let count = Arc::new(AtomicU32::new(0));
    for i in 0..threads {
        let start = from + chunk * i;
        let end = if i == threads - 1 { to } else { start + chunk - 1 };
        let count = count.clone();
        let handle = thread::spawn(move || {
            for n in start..=end {
                if is_prime(n) {
                    count.fetch_add(1, Ordering::SeqCst);
                }
            }
        });
        handles.push(handle);
    }
    for h in handles {
        h.join().unwrap();
    }
    count.load(Ordering::SeqCst)
}
```

The Mutex

- Classic synchronization primitive
- Owns the data it protects
- Data cannot be accessed without locking on the mutex
- Wrap in `Arc<>` and clone before passing (moving) to a thread

Mutex Example

```rust
use std::thread;
use std::sync::{ Arc, Mutex };

fn list_primes(threads: u32, from: u32, to: u32) -> Vec<u32> {
    let mut handles = Vec::new();
    let chunk = (to - from + 1) / threads;
    let mutex = Arc::new(Mutex::new(Vec::<u32>::new()));

    for i in 0..threads {
        let start = from + chunk * i;
        let end = if i == threads - 1 { to } else { start + chunk - 1 };
        let mutex = mutex.clone();
        let handle = thread::spawn(move || {
            for n in start..=end {
                if is_prime(n) {
                    let mut vec = mutex.lock().unwrap();
                    vec.push(n);
                }
            }
        });
        handles.push(handle);
    }
    for h in handles {
        h.join().unwrap();
    }
    let result = mutex.lock().unwrap().to_owned();
    result
}
```

The Send and Sync Traits

- Marker (and unsafe) traits (no methods)
- Send
  - Owner of the type can be transferred to another thread
  - Most types are Send (e.g. Arc<>)
    - Counter example: `Rc<>`
- Sync
  - Objects of the type can be shared by multiple threads
    - i.e. type T is Sync if &T is Send
  - Example: `Mutex<>`
    - Counter examples: `Rc<>`, `Cell<>`, `RefCell<>`
- A type is Send/Sync if all its fields are Send/Sync

Other Synchronization Primitives

- `Barrier`
  - Threads using the barrier must wait at the barrier until all threads make it to the barrier
- `Condvar`
  - Condition variable
- `Once`
  - One time thread safe initialization
- `RwLock`
  - Single writer (exclusive), multiple reader (shared) locking

Summary

- Rust and Concurrency
- Threads
- Message Communication
- Sharing Data
- The Send and Sync traits

Advanced Topics

MODULE 12

Agenda

- Unsafe Rust
- Foreign Function Interface (FFI)
- Macros
- Interior Mutability
- Lifetimes
- Summary

Unsafe Rust

- Rust is a safe language (memory, concurrency)
- Many of its internals, however, are built with *unsafe* code
  - No compiler guarantees
- Example: many parts of `Vec<>`
- Any calls to other languages are unsafe by definition
- Unsafe code must be in an `unsafe` keyword block

```
pub fn push(&mut self, value: T) {
    if self.len == self.buf.capacity() {
        self.reserve(1);
    }
    unsafe {
        let end = self.as_mut_ptr().add(self.len);
        ptr::write(end, value);
        self.len += 1;
    }
}
```

Safe vs. Unsafe

- Unsafe code can do anything
- Safe code must trust unsafe code implicitly
- If unsafe operations are required, wrap them in a safe function
  - Never require the caller to specify an `unsafe` block explicitly

Interfacing with Other Languages

- Calling functions/APIs written in a different language is a common requirement
- Foreign Function Interface (FFI)
  - Rust support for working with non-Rust environments
- Classic example: calling OS APIs
- The `std::ffi` module

Calling C Functions Example

```rust
extern "stdcall" {
    fn Beep(freq: u32, druration: u32) -> bool;
    fn GetLastError() -> u32;
}
```

```rust
fn beep(freq: u32, duration: u32) -> Result<(), u32> {
    unsafe {
        match Beep(freq, duration) {
            false => Err(GetLastError()),
            _ => Ok(())
        }
    }
}
```

```rust
fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() < 3 {
        println!("Usage: beep <frequency> <duration_in_msec>");
    }
    else {
        let freq: u32 = args[1].parse().unwrap();
        let duration: u32 = args[2].parse().unwrap();
        beep(freq, duration).unwrap();
    }
}
```

Macros

- In C/C++, macros consist of text substitutions
- Rust macros are context aware and have access to parts of the Abstract Syntax Tree (AST)
- Macro types
  - Declarative macros
    - The most common
  - Procedural macros

Declarative Macro Example

- Defined with the `macro_rules!` macro

```
macro_rules! readline {
    ($a:expr) => ({
        std::io::stdin().read_line(&mut $a).unwrap();
        $a = $a.trim().to_string();
    });
    () => ({
        let mut s = String::new();
        std::io::stdin().read_line(&mut s).unwrap();
        s = s.trim().to_string();
        s
    });
}
```

```
fn main() {
    println!("What is your first name?");
    let name = readline!();
    println!("What is your last name?");
    let mut last_name = String::new();
    readline!(last_name);

    println!("Your full name is {} {}", name, last_name);
}
```

AST Elements in Macros

Element Name	Description
`expr`	Any expression
`ident`	Identifier
`item`	Module level item (functions, use declarations, etc.)
`block`	Sequence of statements
`meta`	Meta item (parameters inside attributes)
`pat`	Pattern
`Path`	Qualified names
`stmt`	Statement
`tt`	Token tree
`ty`	Type (e.g. u32, String, i64)
`vis`	Visibility (e.g. pub)
`lifetime`	Lifetime (e.g. ‘a)
`Literal`	Literal that can be any token (e.g. “abc”, some_var)

Common Built-in Macros

- `dbg!`
  - Prints the expression and its value
  - Pass references to avoid moving
- `compile_error!`
  - Reports an error and terminates compilation
- `env!`
  - Returns an environment variable's value at compile time
  - Panics if does not exist
- `option_env!`
  - Similar to `env!`, but does not panic if variable does not exist
- `eprint!` and `eprintln!`
  - Similar to `print(ln)!`, but prints to standard error
- `concat!`
  - Concatenates any number of literals and returns a single literal of type `&'static str`

Interior Mutability

- Common pattern that allows mutation even with immutable references
  - Wrapper object over the data to mutate
- `RefCell<T>`
  - Enforces borrowing rules at runtime instead of compile time
  - If violated, causes a panic
- Example: managing a cache

Working with `RefCell<T>`

- Borrowing must be explicitly requested
  - `borrow` method borrows immutably
    - Returns a `Ref<T>` smart pointer
  - `borrow_mut` method borrows mutably
    - Returns a `RefMut<T>` smart pointer
- Works internally by counting the number of outstanding immutable and mutable borrows

Lifetimes

- In most situations in Rust, the compiler infers the lifetime of objects and references, preventing dangling references
  - Typically tied to scope

```
fn main() {
    let x = 6;

    let r = &x;

    println!("r: {}", r);
}
```


```
fn main() {
    let r;
    {
        let x = 6;
        r = &x;
    }

    println!("r: {}", r);
}
```

The Need for Lifetime Annotations (1)

```
fn longest_string(s1: &str, s2: &str) -> &str {
    if s1.len() > s2.len() {
        s1
    }
    else {
        s2
    }
}
```

```
Error[E0106]: missing lifetime specifier
  |
  | fn longest_string(s1: &str, s2: &str) -> &str {
  |                       ----      ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say
whether it is borrowed from `s1` or `s2`
help: consider introducing a named lifetime parameter
  |
  | fn longest_string<'a>(s1: &'a str, s2: &'a str) -> &'a str {
  |                   ^^^^     ^^^^^^^      ^^^^^^^     ^^^
```

- What is wrong here?

The Need for Lifetime Annotations (2)

```
fn main() {
    let x = String::from("Hello, Rust!");
    let y = "Hello, world!";
    let s = longest_string(x.as_str(), y);
    println!("{}", s);
}
```

```
fn main() {
    let x = String::from("Hello, Rust!");
    let s;
    {
        let y = "Hello, world!";
        s = longest_string(x.as_str(), y);
    }
    println!("{}", s);
}
```

- Solution

```
fn longest_string<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    }
    else {
        s2
    }
}
```

Lifetime Annotations

- No need for the programmer to specify in most cases
- Some cases do require such *lifetime annotations*
  - Specified similarly to generic parameters
  - Must start with a ‘ (apostrophe), followed by an identifier
    - The letter *a* is typically used, followed by *b* (if needed), etc.
- Lifetime annotations can be specified for
  - Functions
  - `Structs`
  - `Impl` blocks

The `'static` Lifetime

- Special lifetime that indicates the entire duration of the program
- This is the lifetime of string literals
- Sometimes compiler error suggests using such a lifetime
  - Usually because of a dangling reference is attempted to be returned from a function

Lifetime Elision

- The compiler has three rules for lifetime annotations
  - If any of the rules apply, no annotations are needed
- Rule 1: each input parameter that is a reference, gets its own lifetime
- Rule 2: if there is exactly one input lifetime parameter, that lifetime is assigned to the output lifetime parameter
- Rule 3: if there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because it's a method, the lifetime of `self` is assigned to all output lifetime parameters

Summary

- Unsafe Rust
- Foreign Function Interface (FFI)
- Macros
- Interior Mutability
- Lifetimes