

Rust Models

Jim Fawcett

<https://JimFawcett.github.io>

<https://jimfawcett.github.io/Resources/RustModels.pdf>

Why Rust?

- Memory Safety
 - No dangling pointers or null references
 - No reading or writing to unowned memory
 - Rust's type system enforces sane ownership policies.
- No Data Races
 - The same ownership policies applied to thread interactions ensures data race free operation
- Performance
 - As fast as C and C++
- Abstraction without Overhead
 - Traits and Trait objects
 - In the same ballpark as C++

Prologue – Hello World!

- This section assumes you have no experience with Rust.
- Getting started:
 - Install Rust - <https://www.rust-lang.org/tools/install>
 - This takes just a few minutes
 - Puts cargo, Rust's package manager, builder, executer on your path
 - Install Visual Studio Code - <https://code.visualstudio.com/download>
- Now we're ready for a hello world ++ experiment.
 - Create a temporary directory and navigate to that in a command prompt.
 - Issue command: `cargo new hello`
 - Issue command: `cd hello`
 - Issue command: `code .` [opens Visual Studio Code in hello directory]

Hello World

```
x64 Native Tools Command Prompt for VS 2019

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

c:\temp>
cd hello

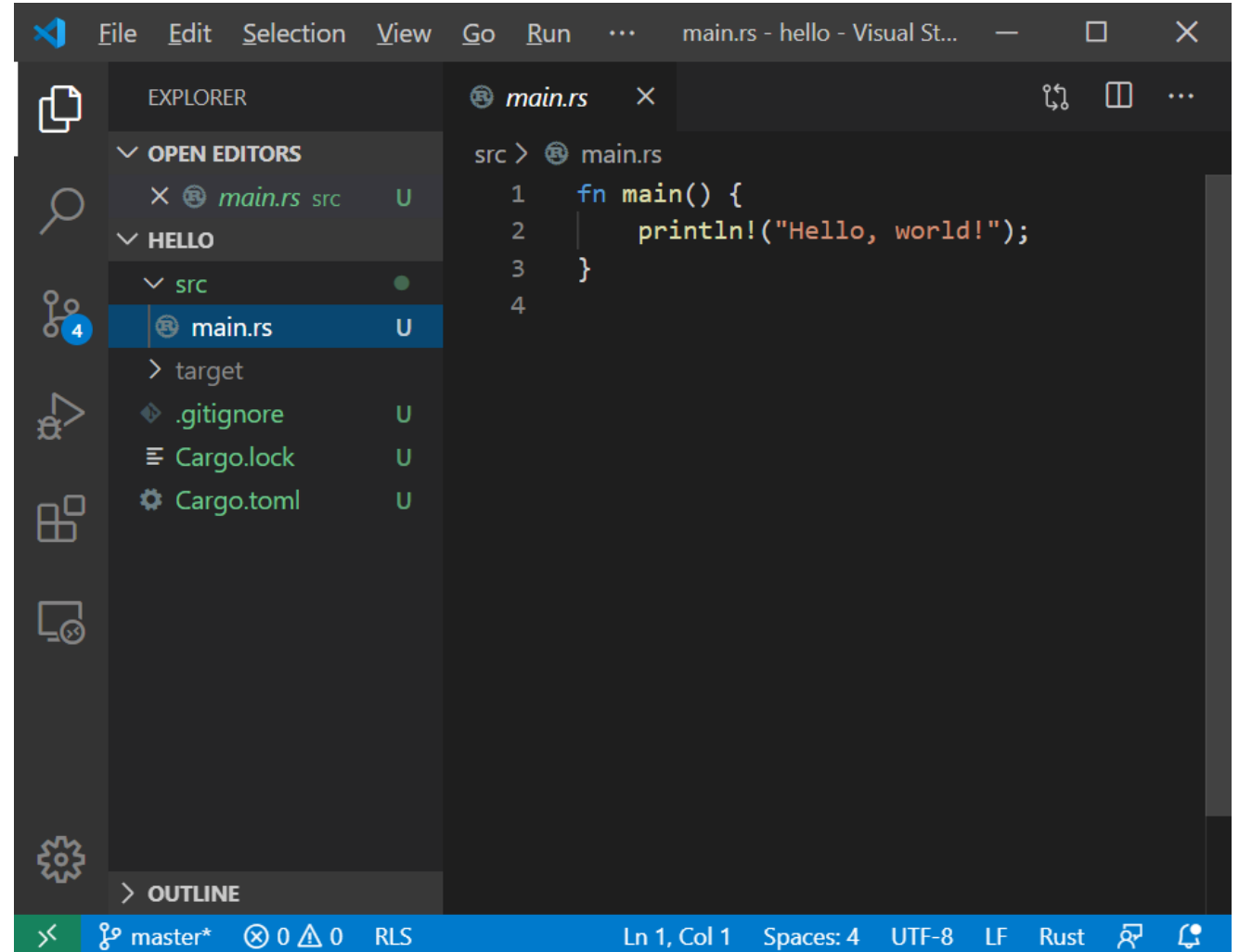
c:\temp\hello>
dir
Volume in drive C is OS
Volume Serial Number is 765A-DAD5

Directory of c:\temp\hello

03/29/2020  09:28 AM    <DIR>        .
03/29/2020  09:28 AM    <DIR>        ..
03/29/2020  09:28 AM                8 .gitignore
03/29/2020  09:28 AM            229 Cargo.toml
03/29/2020  09:28 AM    <DIR>        src
                2 File(s)            237 bytes
                3 Dir(s)  629,056,757,760 bytes free

c:\temp\hello>
code .

c:\temp\hello>
```



The screenshot shows the Visual Studio Code interface. The Explorer view on the left displays the project structure:

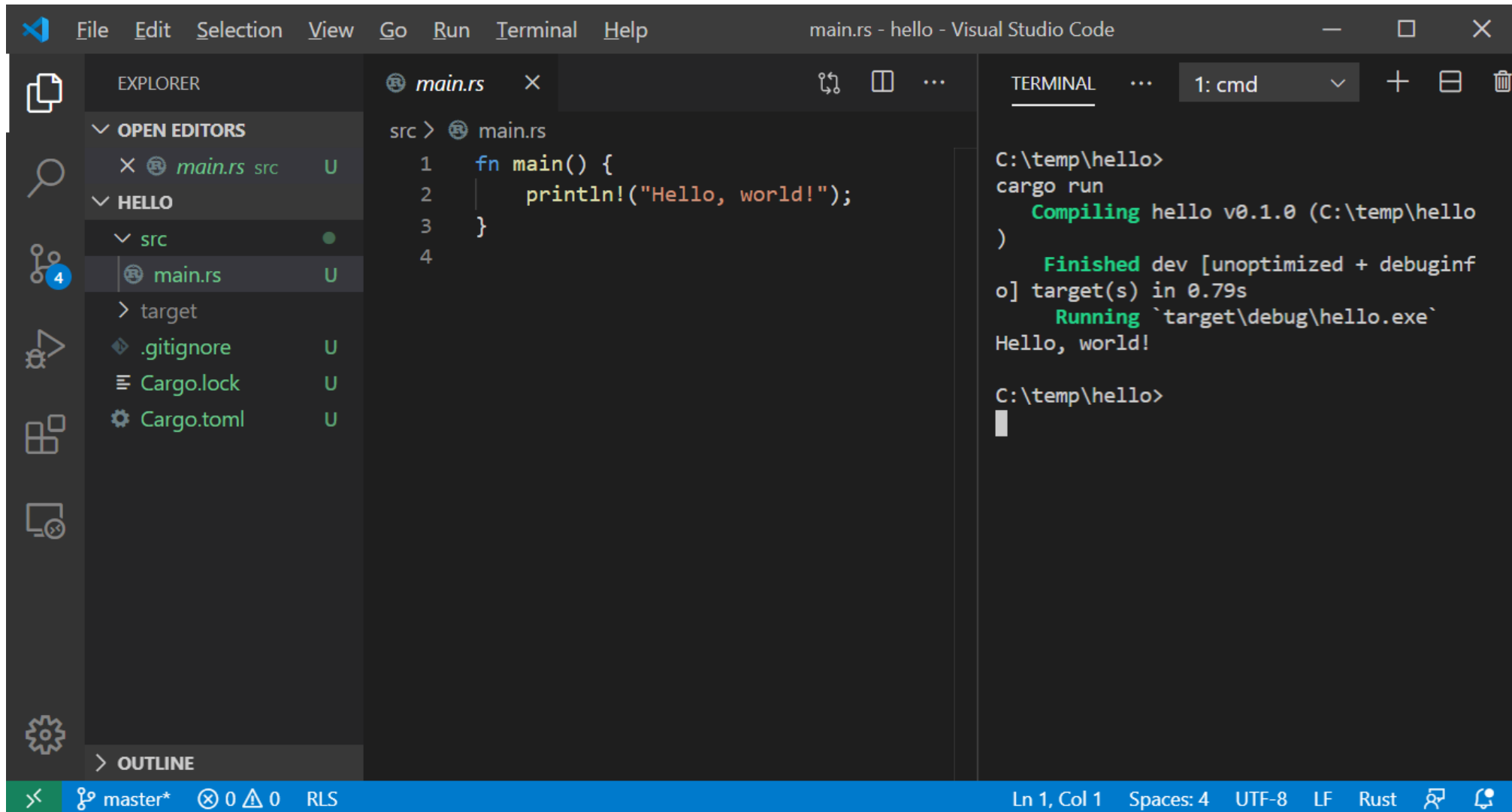
- EXPLORER
 - OPEN EDITORS
 - main.rs src U
 - HELLO
 - src
 - main.rs U
 - target
 - .gitignore U
 - Cargo.lock U
 - Cargo.toml U

The Editor view on the right shows the code for 'main.rs':

```
src > main.rs
1 fn main() {
2     println!("Hello, world!");
3 }
4
```

The status bar at the bottom indicates the current file is 'main.rs' in the 'src' directory, with the following details: master*, 0 errors, 0 warnings, RLS, Ln 1, Col 1, Spaces: 4, UTF-8, LF, Rust.

Building and Running with Cargo



The screenshot displays the Visual Studio Code interface with a Rust project. The Explorer panel on the left shows the project structure with files like `main.rs`, `src`, `target`, `.gitignore`, `Cargo.lock`, and `Cargo.toml`. The main editor shows the `main.rs` file with the following code:

```
src > main.rs
1 fn main() {
2     println!("Hello, world!");
3 }
4
```

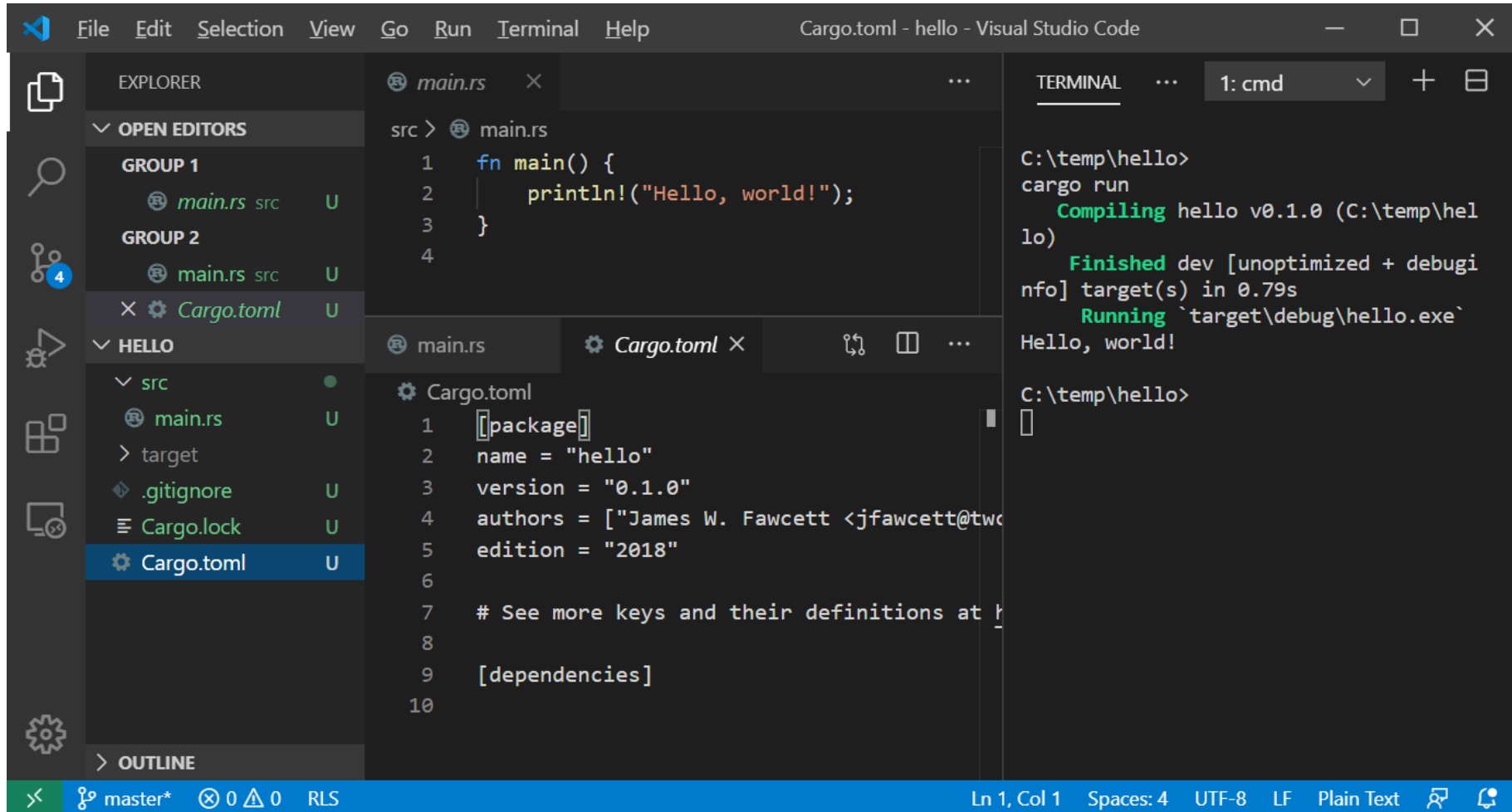
The TERMINAL panel on the right shows the output of the `cargo run` command:

```
C:\temp\hello> cargo run
   Compiling hello v0.1.0 (C:\temp\hello)
 Finished dev [unoptimized + debuginfo] target(s) in 0.79s
Running `target\debug\hello.exe`
Hello, world!

C:\temp\hello>
```

The status bar at the bottom indicates the current branch is `master*`, there are 0 commits and 0 changes, and the Rust language is active.

Cargo.toml – defines package



The screenshot displays the Visual Studio Code interface for a Rust project named 'hello'. The Explorer sidebar on the left shows the project structure, with 'Cargo.toml' highlighted under the 'src' directory. The main editor area shows two files: 'main.rs' and 'Cargo.toml'. 'main.rs' contains a simple Rust program that prints 'Hello, world!'. 'Cargo.toml' defines the package name as 'hello', version '0.1.0', and lists the author as 'James W. Fawcett'. The terminal on the right shows the command 'cargo run' being executed, which compiles the program and runs it, outputting 'Hello, world!'.

```
File Edit Selection View Go Run Terminal Help
Cargo.toml - hello - Visual Studio Code

EXPLORER
OPEN EDITORS
GROUP 1
  main.rs src U
GROUP 2
  main.rs src U
  Cargo.toml U
HELLO
  src
    main.rs U
    target
    .gitignore U
    Cargo.lock U
    Cargo.toml U
  OUTLINE

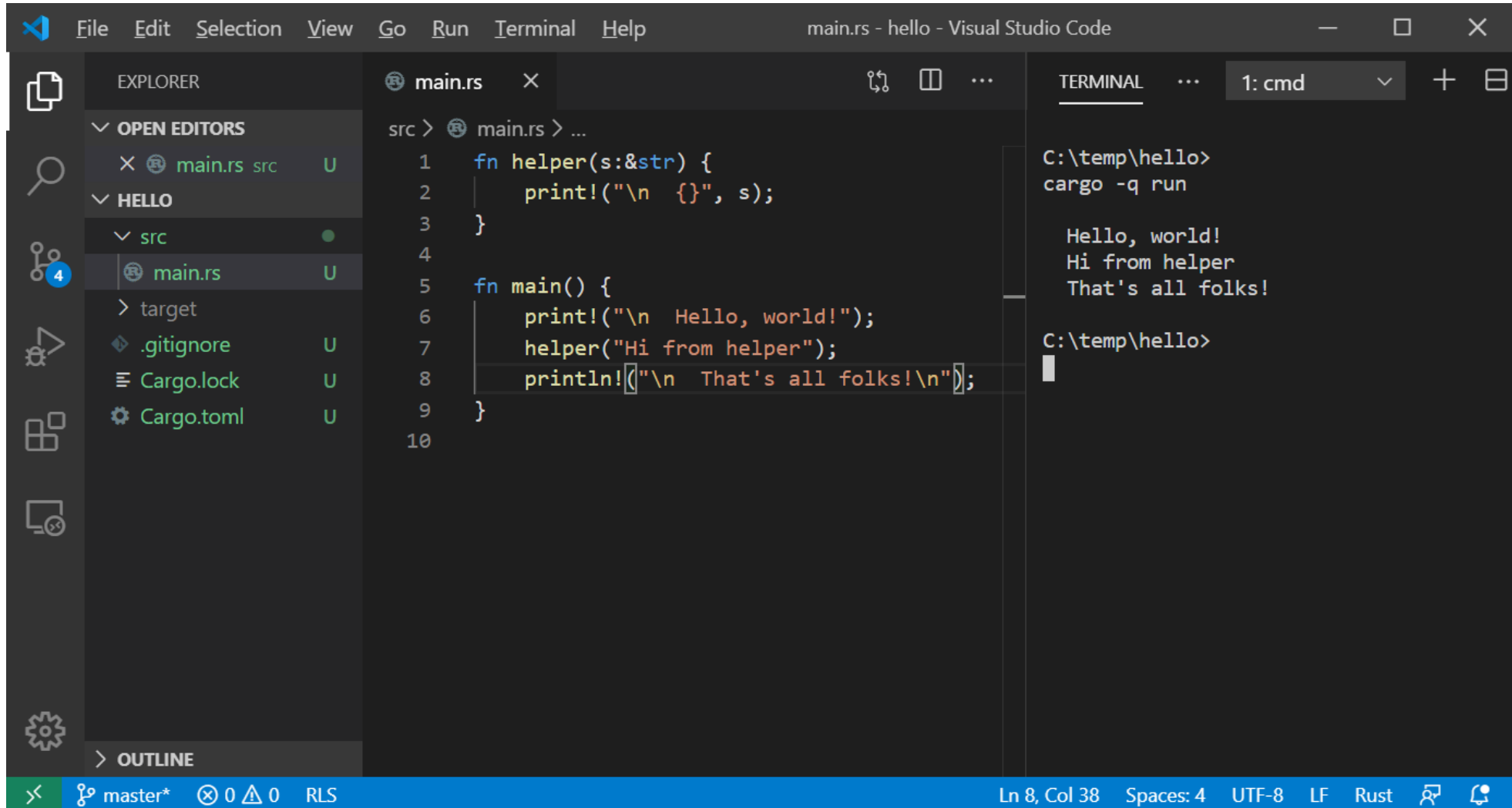
main.rs
src > main.rs
1 fn main() {
2     println!("Hello, world!");
3 }
4

Cargo.toml
1 [[package]]
2 name = "hello"
3 version = "0.1.0"
4 authors = ["James W. Fawcett <jfawcett@tw"]
5 edition = "2018"
6
7 # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
8
9 [dependencies]
10

TERMINAL
1: cmd
C:\temp\hello>
cargo run
Compiling hello v0.1.0 (C:\temp\hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.79s
Running `target\debug\hello.exe`
Hello, world!

C:\temp\hello>
```

Add another function



The screenshot shows the Visual Studio Code interface with a Rust project named 'hello'. The Explorer panel on the left shows the project structure with 'main.rs' selected. The main editor displays the code in 'main.rs', which includes a new 'helper' function and a call to it in the 'main' function. The output panel on the right shows the program's execution results.

```
src > main.rs > ...
1  fn helper(s:&str) {
2      print!("\n {}", s);
3  }
4
5  fn main() {
6      print!("\n Hello, world!");
7      helper("Hi from helper");
8      println!("\n That's all folks!\n");
9  }
10
```

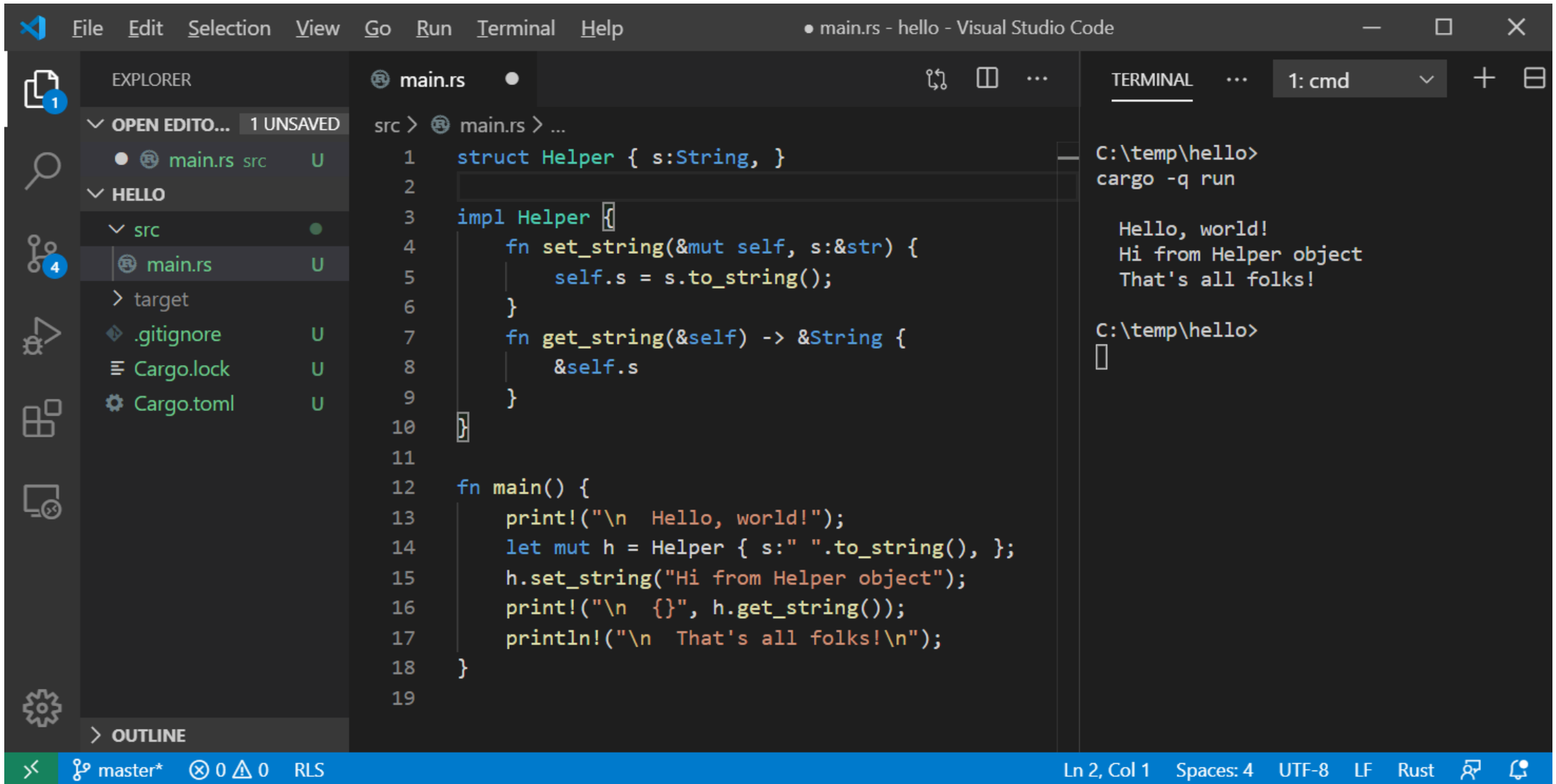
Output:

```
C:\temp\hello>
cargo -q run

Hello, world!
Hi from helper
That's all folks!

C:\temp\hello>
```

Modify to use “object”



The screenshot shows the Visual Studio Code interface with a Rust project named 'hello'. The Explorer panel on the left shows the project structure with 'main.rs' in the 'src' directory. The main editor displays the code for 'main.rs', which defines a 'Helper' struct and implements methods for setting and getting a string. The 'main' function creates a 'Helper' object, sets its string to 'Hi from Helper object', and prints it. The Terminal panel on the right shows the command 'cargo -q run' being executed, resulting in the output: 'Hello, world!', 'Hi from Helper object', and 'That's all folks!'.

```
File Edit Selection View Go Run Terminal Help
• main.rs - hello - Visual Studio Code

EXPLORER
1 OPEN EDITOR... 1 UNSAVED
  • main.rs src U
  HELLO
    src
      4 main.rs U
    > target
    .gitignore U
    Cargo.lock U
    Cargo.toml U
  > OUTLINE

main.rs
src > main.rs > ...
1 struct Helper { s:String, }
2
3 impl Helper {
4     fn set_string(&mut self, s:&str) {
5         self.s = s.to_string();
6     }
7     fn get_string(&self) -> &String {
8         &self.s
9     }
10 }
11
12 fn main() {
13     print!("\n Hello, world!");
14     let mut h = Helper { s:" ".to_string(), };
15     h.set_string("Hi from Helper object");
16     print!("\n {}", h.get_string());
17     println!("\n That's all folks!\n");
18 }
19

TERMINAL 1: cmd
C:\temp\hello>
cargo -q run

Hello, world!
Hi from Helper object
That's all folks!

C:\temp\hello>

```

Ln 2, Col 1 Spaces: 4 UTF-8 LF Rust

More string handling – see lines 17-18

```
File Edit Selection View Go Run Terminal Help
main.rs - hello - Visual Studio Code

EXPLORER
OPEN EDITORS
  X main.rs src U
HELLO
  src
    main.rs U
  target
  .gitignore U
  Cargo.lock U
  Cargo.toml U
  OUTLINE

main.rs
src > main.rs > ...
2
3 impl Helper {
4     fn set_string(&mut self, s:&str) {
5         self.s = s.to_string();
6     }
7     fn get_string(&self) -> &String {
8         &self.s
9     }
10 }
11
12 fn main() {
13     print!("\n Hello, world!");
14     let mut h = Helper { s:" ".to_string(), };
15     h.set_string("Hi from Helper object");
16     print!("\n {}", h.get_string());
17     let s = String::from("Hello from Helper object");
18     h.set_string(&s);
19     print!("\n {}", h.get_string());
20     println!("\n That's all folks!\n");
21 }
22

TERMINAL
1: cmd
C:\temp\hello>
cargo -q run

Hello, world!
Hi from Helper object
Hello from Helper object
That's all folks!

C:\temp\hello>
```

Ln 18, Col 22 Spaces: 4 UTF-8 LF Rust

Why Rust?

- Memory Safety
 - No dangling pointers or null references
 - No reading or writing to unowned memory
 - Rust's type system enforces sane ownership policies.
- No Data Races
 - The same ownership policies applied to thread interactions ensures data race free operation
- Performance
 - As fast as C and C++
- Abstraction without Overhead
 - Traits and Trait objects
 - In the same ballpark as C++

Undefined Behavior – C++ dangling reference

The screenshot displays the Visual Studio IDE with a C++ project named 'UndefinedBehavior'. The main function in 'UndefBehavior.cpp' is shown, demonstrating a dangling reference. The code creates a vector 'v' with a capacity of 3, pushes back three elements, and then pushes back a fourth element, which causes undefined behavior. A reference 'r1' is created to 'v[1]', and its value is printed before and after the undefined behavior. The output in the 'Microsoft Visual Studio Debug Console' shows that the value of 'r1' changes from 2 to -572662307 after the undefined behavior occurs.

```
20 int main() {
21
22     std::cout << "\n Demo of Undefined Behavior - dangling reference";
23     std::cout << "\n -----";
24
25     std::vector<int> v;
26     v.reserve(3);
27     std::cout << "\n capacity of v = " << v.capacity();
28     v.push_back(1);
29     v.push_back(2);
30     v.push_back(3);
31     showVec(v);
32     int& r1 = v[1];
33     std::cout << "\n address of v[1] = " << &v[1];
34     std::cout << "\n address of r1 = " << &r1;
35     std::cout << "\n value of r1 = " << r1;
36     v.push_back(4);
37     showVec(v);
38     std::cout << "\n address of v[1] = " << &v[1];
39     std::cout << "\n address of r1 = " << &r1;
40     std::cout << "\n value of r1 = " << r1;
41     std::cout << std::endl;
```

Microsoft Visual Studio Debug Console

```
Demo of Undefined Behavior - dangling reference
-----
capacity of v = 3
1 2 3
address of v[1] = 013A3254
address of r1 = 013A3254
value of r1 = 2
1 2 3 4
address of v[1] = 01395DFC
address of r1 = 013A3254
value of r1 = -572662307
```

Undefined Behavior – C++ index out of bounds

The screenshot shows the Visual Studio IDE with a C++ project named 'UndefinedBehavior'. The code in 'UndefBehavior.cpp' is as follows:

```
42  
43     std::cout << "\n Demo of Undefined Behavior - out of bounds index";  
44     std::cout << "\n -----";  
45  
46     int array[3]{ 1, 2, 3 };  
47     std::cout << "\n ";  
48     for (size_t i = 0; i <= 3; ++i) {  
49         std::cout << array[i] << " ";  
50     }  
51     std::cout << std::endl;  
52 }
```

The code attempts to access `array[3]`, which is out of bounds for an array of size 3. The output window shows the following text:

```
Demo of Undefined Behavior - out of bounds index  
-----  
1 2 3 -858993460  
  
C:\su\temp\UndefinedBehavior\Debug\UndefinedBehavior.exe (process  
13708) exited with code 0.  
Press any key to close this window . . .
```

The output demonstrates the undefined behavior of accessing memory outside the array's bounds, resulting in a garbage value (-858993460) being printed.

Rust won't allow mutation with an active borrow

```
1 fn main() {
2     let mut v = Vec::<i32>::with_capacity(3);
3     v.push(1);
4     v.push(2);
5     v.push(3);
6     print!("\n v capacity = {}", v.capacity());
7
8     let r1 = &v[1];
9     print!("\n address of v[1] = {:?}", &v[1] as *const i32);
10    print!("\n address of r1 = {:?}", r1 as *const i32);
11
12    v.push(4); // fails to compile, can't mutate while borrowed
13    print!("\n address of v[1] = {:?}", &v[1] as *const i32);
14    print!("\n address of r1 = {:?}", r1 as *const i32);
15
16    println!("\n\n Hello, Ownership!\n");
17 }
18
```

error: aborting due to previous error

For more information about this error, try `rustc --explain E0502`.

error: could not compile `type_safety`.

To learn more, run the command again with --verbose.

C:\temp\type_safety>

Rust allows mutation if we don't use the borrow

The screenshot shows the Visual Studio Code editor with a Rust project named 'type_safety'. The Explorer sidebar on the left shows the project structure with 'src/main.rs' selected. The main editor displays the code in 'main.rs'.

```
1 fn main() {  
2     let mut v = Vec::<i32>::with_capacity(3);  
3     v.push(1);  
4     v.push(2);  
5     v.push(3);  
6     println!("\n v capacity = {}", v.capacity());  
7  
8     let r1 = &v[1];  
9     println!("\n address of v[1] = {:?}" , &v[1] as *const i32);  
10    println!("\n address of r1   = {:?}" , r1 as *const i32);  
11  
12    v.push(4); // fails to compile, can't mutate while borrowed  
13    println!("\n address of v[1] = {:?}" , &v[1] as *const i32);  
14    //println!("\n address of r1   = {:?}" , r1 as *const i32);  
15  
16    println!("\n\n Hello, Ownership!\n");  
17  
18 }
```

The Terminal on the right shows the command 'cargo -q run' being executed. The output displays the memory addresses of v[1] and r1, and the message 'Hello, Ownership!'.

```
C:\temp\type_safety>  
cargo -q run  
  
v capacity = 3  
address of v[1] = 0x8df574  
address of r1   = 0x8df574  
address of v[1] = 0x8dba84  
  
Hello, Ownership!  
  
C:\temp\type_safety>
```

The status bar at the bottom indicates the current file is 'main.rs' at line 14, column 7, with 4 spaces, UTF-8 encoding, and LF line endings. The Rust compiler is also indicated.

Prologue – Hello Ownership!

- Rust's ownership policies:
 - Every value has one and only one owner
 - Ownership can be transferred with a move
 - Ownership can be borrowed with a reference
 - References hold a view into value
 - Original value's owner can't mutate value while borrowed
 - Immutable references can be shared
 - Mutable references are exclusive
 - Borrowing ends when reference goes out of scope or is dropped
 - This fits very well with pass by reference function arguments
 - Values are, by default, immutable, but can be made mutable
 - `let x = 3; // x is immutable`
 - `let mut y = 3; // y is mutable`

Hello Rust Ownership

```
File Edit Selection View Go Run Terminal Help
main.rs - hello_ownership - Visual Studio Code

EXPLORER
1 OPEN EDITOR... 1 UNSAVED
  main.rs src U
  HELLO_OWNERSHIP
    src
      main.rs U
    target
    .gitignore U
    Cargo.lock U
    Cargo.toml U
  OUTLINE

main.rs
src > main.rs > ...
1  #[allow(unused_variables)]
2  #[allow(unused_assignments)]
3  fn main() {
4      let x = 5;
5      // x = 6; // fails to compile, not mutable
6      let mut y = 5;
7      y = 6;
8
9      let ry = &mut y;
10     *ry = *ry + 1;
11     // y = y + 1;
12     // compile fails, value of y borrowed
13     print!("\n value of ry = {}", *ry);
14     // print!("\n value of y = {}", y);
15     // compile fails print! takes y by ref
16     // but already have mut ref of y
17
18     y = y + 1; // ok, borrow dropped
19     print!("\n value of y mutated to {}", y);
20
21     println!("\n\n Hello, ownership again!\n");
22 }

TERMINAL
1: cmd
C:\temp\hello_ownership> cargo -q run

value of ry = 7
value of y mutated to 8

Hello, ownership again!

C:\temp\hello_ownership>
```

Ln 12, Col 14 Spaces: 4 UTF-8 LF Rust

Model

- “A model of a system or process is a theoretical description that can help you understand how the system or process works, or how it might work.”
- [collinsdictionary.com](https://www.collinsdictionary.com)
- Models help us understand important features of a language
 - Use language effectively
 - Accelerate learning process

Prologue

https://JimFawcett.github.io/RustStory_Models.html

- Rust is an interesting and ambitious language.
- We will consider Models for:
 - Type Safety
 - Ownership
 - Objects
 - User-Defined Types
 - Generics
 - Code Structure, Compilation, and Execution
- Chapter 1 of the Rust Story
 - https://jimfawcett.github.io/RustStory_Prologue.html

Rust Type System

https://jimfawcett.github.io/RustStory_Models.html#types

Type Safety

- A program is well defined if no execution can exhibit undefined behavior.
- A language is type safe if its type system ensures that every program is well defined.
- A non-type safe language may introduce undefined behavior with:
 - Integer overflow, e.g., wrap-around
 - Buffer overflow – out of bounds access
 - Use after free – access unowned memory
 - Double free – corrupt memory manager
 - Race conditions – mutation without exclusive ownership

Unsafe Type System – C++ and C

- Allows dangling references
 - `std::vector<int> v { 1, 2, 3 };`
 - `int& ri = v[1];`
 - `v.push_back(4);`
 - `push_back` may cause reallocation of vector memory, leaving `ri` dangling.
- Allows out of bounds indexing
 - `int j = v[4];`
- Program continues to run, results in undefined behavior
- Mitigations:
 - Use C++ references only for passing arguments to functions.
 - Prefer passing by const reference to avoid side-effects.
 - Don't use arrays; use `std::vector`.
 - Use range-based for loop when iterating through collections like vector.

Safe Type System - Rust

- Rust is a type safe language, avoiding undefined behavior.
- Rust's type system prevents data races in multi-threaded programs.
- Rust's type system ensures this behavior by:
 - Preventing mutation combined with aliasing
 - Ensure memory safety
 - Preventing mutation, aliasing, and lack of access ordering
 - Avoid data races

Safe Type System - Rust

- Prevent dangling references:
 - `let mut v = Vec::<f64>::new();`
 - `v.push(1.0); v.push(2.0);`
`v.push(3.0);`
 - `let r1 = &v[1];`
 - `// v.push(4.0); // fails to compile`
 - `Print!("\n r1 = {?:}", r1);`
 - `drop(r1);`
 - `v.push(4.0); // ok`
- `v` owns vector
- `r1` borrows ownership
- `v` can't mutate until `r1` is dropped or goes out of scope
- Going out of scope calls `drop` implicitly

Safe Type System - Rust

- Prevent access to unowned memory:
 - `let mut v = Vec::<f64>::new();`
 - `v.push(1.0), ...`
 - `for x in v {`
 `print!("\n {}", x)`
 `}`
 - `for i in 0...3 {`
 `print!("\n {}", v[i]);`
 `}`
 - If bound 3 is incorrect, will panic, not allowing access to unowned memory.
- Rust knows the size of almost all values, and uses that in for loops like the first.
- We can cause access to unowned memory, but that results in aborting the program without allowing access to unowned memory.

Safe Type System - Rust

- Rust is a type safe language, avoiding undefined behavior.
- Rust's type system prevents data races in multi-threaded programs, based on its ownership model.
 - We won't discuss data races further in these notes.
- It provides level of performance and access to machine resources needed for system programming.

Ownership Model

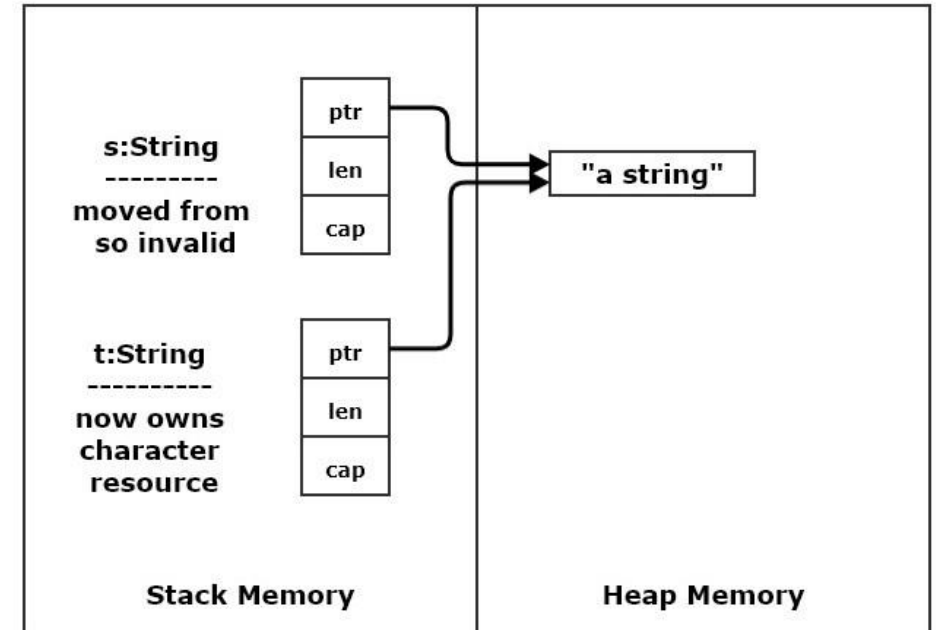
https://jimfawcett.github.io/RustStory_Models.html#ownership

Rust Ownership

- Values in Rust have only one owner, defined by a `let` statement:
 - `let x = 3;`
 - `let y = x; // initialized with copy of x so x is still valid`
 - `let u = String::from("a string");`
 - `let v = u; // u's value moved to v, u is no longer valid`
- What's the difference here?
 - `x` is an `i32` (integer), is blittable, so is copy-able
 - `u` is a `String`, is not blittable (`String` chars are stored in heap), so is moved.
 - A move transfers ownership to the target and invalidates the source.
- Blittable values can be copied, non-blittable values can only be moved. A value is blittable if it can be copied with `memcpy`.

Move

- `let s = String::from("a string");`
 - `s` consists of a control block in stack memory and a character array in the heap.
- `let t = s;`
 - `s`'s **control block** is blitted to `t`
 - That preserves the pointer to the heap character array.
 - So now `t` owns the string and `s` is marked as invalid.
- This is fast. Characters are not copied, only the small control block is copied.



Immutable References

- Any number of immutable references may be declared for a value:
 - `let mut s = String::from("a string");`
 - `let r1 = &s;`
 - `let r2 = &s;`
- The original owner can not mutate until all active references are dropped or go out of scope:
 - `fn show(s:&String) { ... }`
 - `let mut t = String::from("another string");`
 - `show(&t);`
 - `t.push_str(" with more stuff");` // mutation ok, &t when out of scope

Mutable References

- Only one mutable reference may be declared for a value:
 - `let mut s = String::from("a string");`
 - `let r1: &mut String = &mut s;`
 - `// let r2: &mut String = &mut s; // won't compile`
 - `// let r3 = &s; // won't compile`
- The original owner can not mutate until active reference is dropped or goes out of scope (same as before):
 - `fn show(s:&String) { ... }`
 - `let mut t = String::from("another string");`
 - `show(&t); // copies reference to show stack frame, e.g., a borrow`
 - `t.push_str(" with more stuff"); // mutation ok, &mut t went out of scope`

Ownership summary

- These simple rules provide memory safety:
 - `let x = y ==>` copy if blittable, otherwise move `==>` transfer of ownership
 - Can't use `y` if moved from
 - `let r1 = &x; let r2 = &x; ==>` may have any number of immutable references
 - `x` may not be mutated while there are active references
 - `let mut z = ...`
 - `Let r3 = &mut z; ==>` may only have one mutable reference
- References become inactive when they go out of scope or are dropped:
 - `drop(r3);`
- Prefer use of references for pass by reference functions and methods

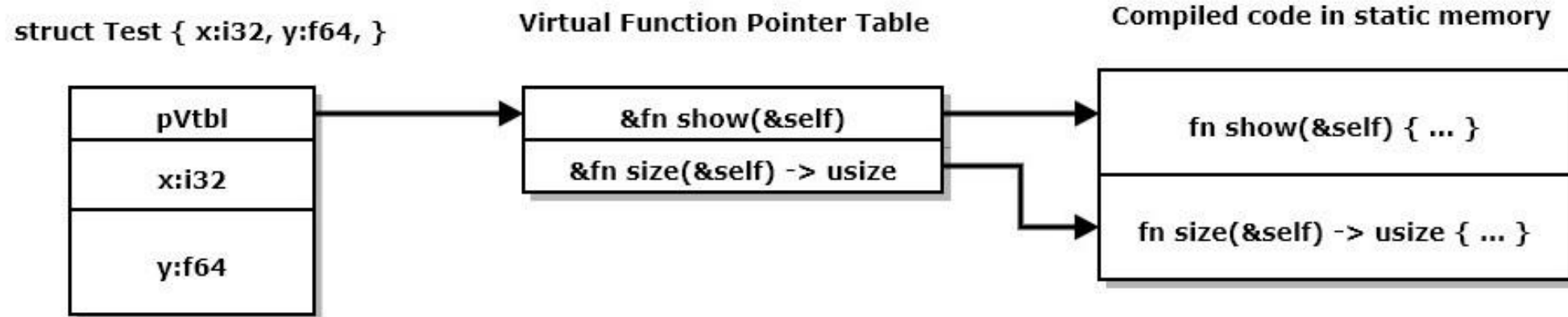
Rust Object Model

https://jimfawcett.github.io/RustStory_Models.html#objmodel

Rust Object Model

- Rust does not have classes but structs are used in a way very similar to the way classes are used in C++.
- Structs have:
 - Composed members, may be instances of language or user defined types.
 - Aggregated members, using the `Box<T>` construct:
 - `Box<T>` acts like a `std::unique_ptr<T>` in C++.
 - Methods - functions that accept `&self` which is a reference to the instance invoking the function.
 - `&self` is similar to the C++ pointer `this`.
 - Traits - implemented by a struct, similar to Java or C# interfaces.
 - Access control - uses the keyword `pub`.
 - Anything not decorated with `pub` is private but accessible in the local crate.

Rust Object Model



- trait Show : Debug { ... }
- trait Size { ... }
- struct Test { x:i32, y:f64, }
- impl Show for Test { ... }
- impl Size for Test { ... }
- impl Test { ... }

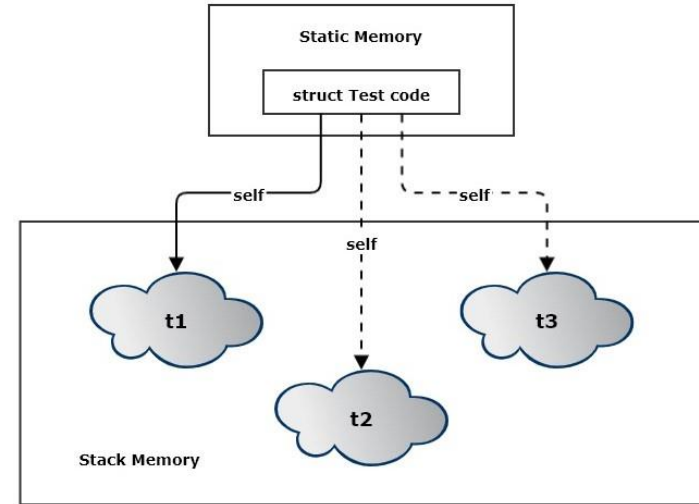
Table 1. Test Struct Memory Layout		
Component	Address	Size - bytes
Test Struct	8190584	16
y:f64	8190584	8
x:i32	8190592	4
ptr to Vtbl	8190596	4

Implementing Traits and Methods

- trait Size {
 fn size(&self) -> usize;
}
- trait Show : Debug {
 fn show(&self) {
 print!("\n {:?}", &self);
 }
}
- pub struct Test { x:i32, y:f64, }
- impl Size for Test {
 fn size(&self) -> usize {
 std::mem::size_of::<Test>()
 }
}
- impl Show : Debug {
 fn show(&self) {
 print!("\n {:?}", &self);
 }
}
- impl Test {
 pub fn new() -> Self {
 Self { x:42, y:1.5, }
 }
}

Copy and Move Types

- Copy types have **instances** that can be copied and assigned.
 - `let t = Test::new();`
 - `let u = t; // copy`
 - `t = u; // assign`
 - Value types implement Copy and Clone traits
- Move types have instances that are moved instead of copied. Any type that does not implement Copy is a move type.
- Moveable types can implement the Clone trait.
- Test is a value type.



```
• trait Size {  
    fn size(&self) -> usize;  
}  
  
• trait Show : Debug {  
    fn show(&self) {  
        print!("\n {:?}", &self);  
    }  
}  
  
• pub struct Test { x:i32, y:f64, }  
  
• impl Size for Test {  
    fn size(&self) -> usize {  
        std::mem::size_of::<Test>()  
    }  
}
```

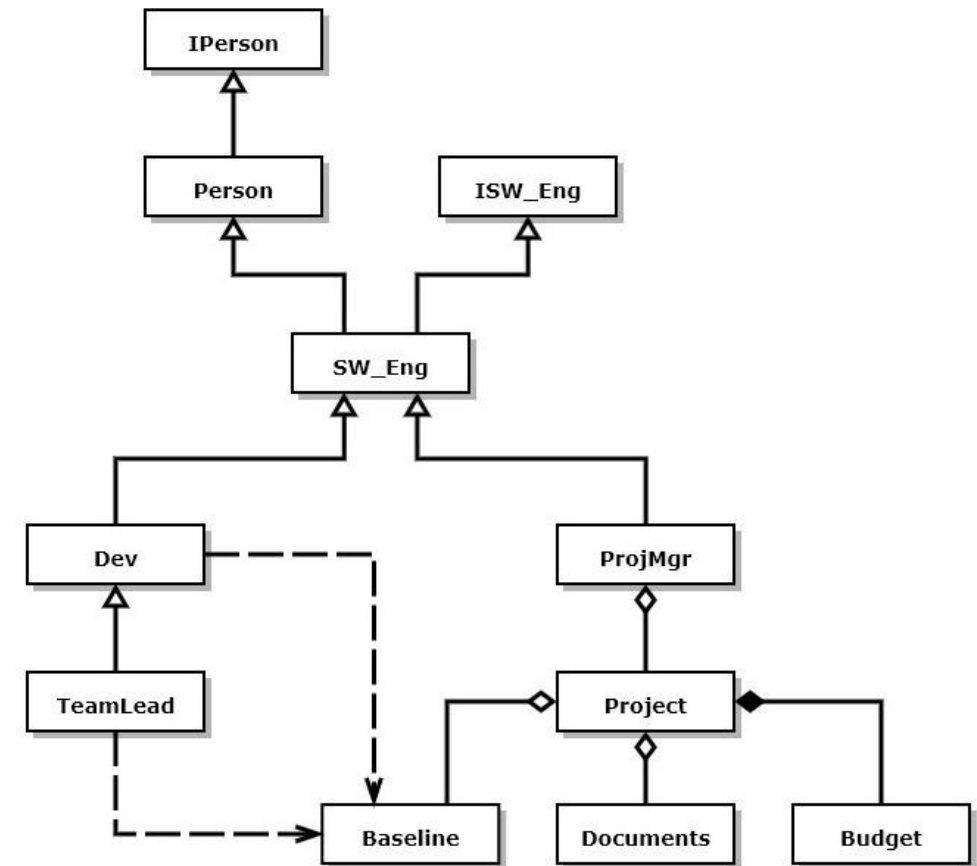
```
• impl Show : Debug {  
    fn show(&self) {  
        print!("\n {:?}", &self);  
    }  
}  
  
• impl Test {  
    pub fn new() -> Self {  
        Self { x:42, y:1.5, }  
    }  
}
```

Comparison with C++

- C++ object model provides:
 - Composition
 - Aggregation
 - Inheritance
 - Most classes can be value types:
 - Copy constructors
 - Assignment operator overloads
 - Destructors
 - Many are value types by default
 - Members are primitive types or STL containers
- Rust object model provides:
 - Composition
 - Aggregation
 - Traits
 - Provide functions but no data
 - Some structs are Copy, but many must be Move.
 - No overloads, so no overloaded assignment operators
 - Move types can implement clone() but that is never called implicitly

C++ Person Class Hierarchy Example – from C++ Models

- The class structure shown on the right represents a software development organization.
- Software Engineers inherit the person type and implement the ISW_Eng interface. SW_Eng is an abstract base class for all software engineers.
- Any function that accepts a pointer to SW_Eng will also accept pointers to Devs, TeamLeads, and ProjMgrs.
- If ISW_Eng defines a pure virtual method, say doWork(), any derived class can override that method.
 - Devs doWork that devs do
 - TeamLeads doWork that team leads do
 - ProjMgrs doWork that project managers do
- So the doWork() method binds to code based on the type of object bound to an ISW_Eng pointer.



Rust Generics

https://jimfawcett.github.io/RustStory_Models.html#generics

Rust Generics

- Rust Generics define constraints that limit the types that will compile.
- Rust generics do not support specializations that broaden the number of types that can be used.

- Generic functions:

- ```
fn demo_ref<T>(t:&T) where T:Debug {
 show_type(t);
 show_value(t);
}
```
- ```
fn show_type<T>(_value:&T) where T:Debug {  
    let name = std::any::type_name::<T>();  
    print!(  
        "\n TypeId: {:?}, size: {:?}",  
        name, size_of::<T>()  
    )  
}
```

- Generic structs:

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



# Code Structure

[https://jimfawcett.github.io/RustStory\\_Models.html#structure](https://jimfawcett.github.io/RustStory_Models.html#structure)

# Code Structure

- Source code is written in files
- For many software systems file structures become large and hard to understand.
- To support readability and maintenance, we create packages that consist of a few files with a single purpose and document the purpose and design in comments.
  - Source files are units of construction
    - Binaries - /src/main.rs – has main function, builds to an executable
    - Libraries - /src/lib.rs – builds to library
    - Modules - /src/\*.rs – loaded when building binaries and libraries
  - A Crate is a unit of translation
  - Crates start as a set of source files in the /src directory and compile to a single file:
    - Binaries - /target/debug/[package\_name].exe on windows
    - Libraries - /target/debug/lib[package\_name].rlib

# Crate

- The source form of a crate is composed of:
  - A crate root, `main.rs` or `lib.rs`, and a set of zero or more supporting source files called modules, all found in the `/src` folder.
  - The crate root loads any modules identified with the keyword `mod` at the top of its source.
    - `mod some_module` → loads `some_module.rs`
  - Each module may also load other modules.
  - The crate may specify dependencies on other crates and import their definitions into the root or any of its modules.
- The translation form of a crate is a single compiled file, e.g., one of:
  - `/target/debug/[package_name].exe`
  - `/target/debug/lib[package_name].rlib`

# Packages

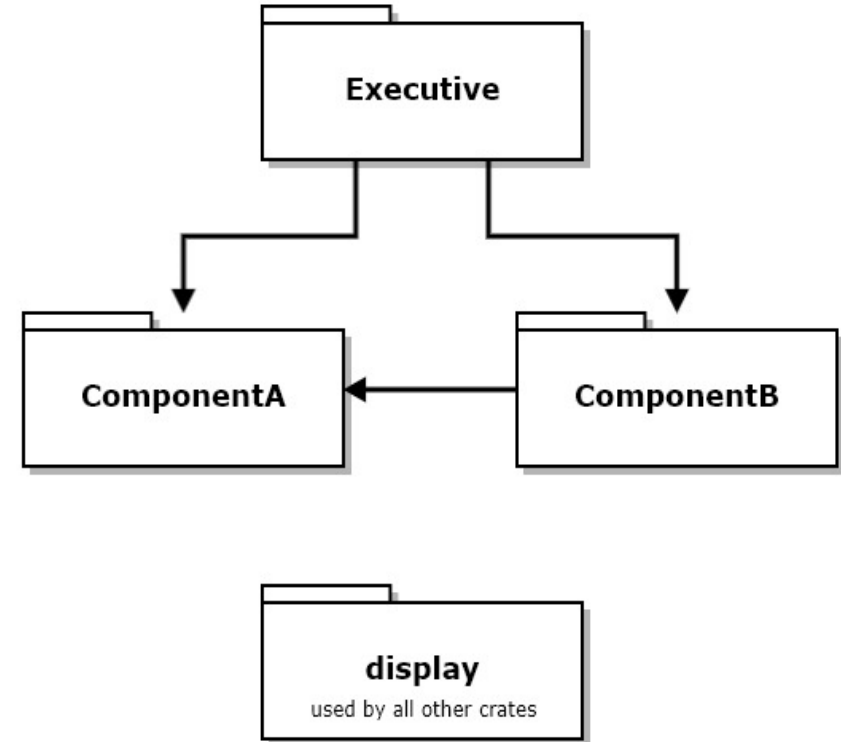
- A Package is a collection of directories and files that are the basis for builds
  - Cargo.toml – specifies package metadata, dependencies, and optional directives
  - /src – directory containing a binary or library source crate
  - /target – directory containing translated binaries or libraries
  - /examples – directory containing example code that exercises the package library
- The Rust build system is transitive
  - Builds start with the package root cargo.toml
  - Parse it to find dependencies
  - Load the depending library and parse its cargo.toml
  - ...
  - Build the local crate along with its dependencies

# Library Crate Construction Co-Tests

- For anything other than trivial example code it's very useful to test as we build code:
  - A library crate is created with the command **cargo new --lib [package-name]**.
  - That builds a lib.rs containing a single configured test that asserts  $2 + 2 = 4$ .
    - This is simply a demonstration of how to build test cases for a library.
    - Each test passes if, and only if, there are no failed assertions.
  - Every time we add a few lines of code in the lib.rs file we add small tests, each in a configured test block and then build and execute with the command:  
**cargo test**  
in a terminal window located in the crate root folder.
  - This “co-test” process allows us to very quickly find errors. If a test fails, the problem is almost certain to be in the few lines of code we entered after the last test.

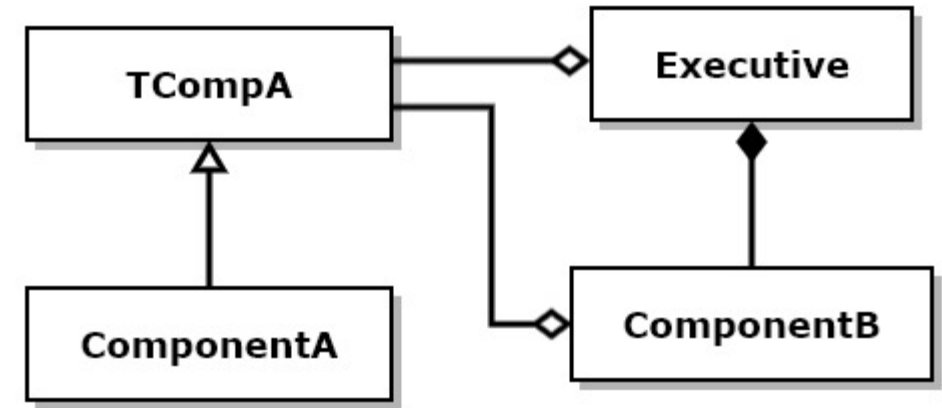
# Example – Crates and Packages

- The diagram at the right shows a set of crates that work together to implement some functionality.
- The diagram shows dependency relationships between crates.
- The ComponentA crate provides an interface and object factory to allow ComponentB and Executive to use it without binding to its implementation details.
- The Executive package consists of all three of these crates.
- Code for this example:  
<https://github.com/JimFawcett/RustBasicDemos/> in code\_structure\_demo



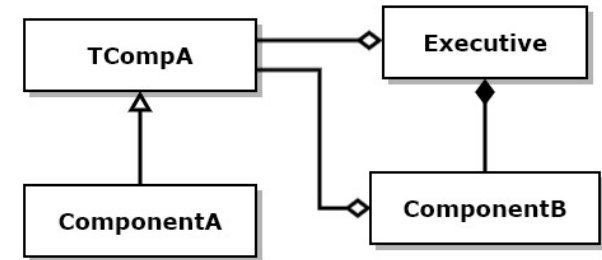
# Example – Traits and Structs

- This diagram shows structs that are defined in each of the files from the previous slide.
  - TCompA is an interface<sup>1</sup> trait for ComponentA
  - ComponentA implements the trait to provide exported services
  - ComponentB doesn't provide an interface
  - ComponentB uses ComponentA through its interface trait and factory<sup>2</sup>
  - Executive composes ComponentB and uses ComponentA through its trait and factory



- 
1. Rust does not have an interface construct. We use traits with virtual functions for that purpose.
  2. ComponentA's factory is implemented with a function, declared and implemented in ComponentA.

# Use of Interfaces and Factories



- If you look at interface trait TCompA you will see it has no implementation detail.

```
pub Trait TCompA {
 fn do_work(&self);
 fn get_msg(&self) -> String;
 fn set_msg(&mut self, m:&str);
}
```

```
pub fn get_instance() -> Box<dyn TCompA> {
 Box::new(ComponentA::new())
}
```

- Executive and ComponentB use ComponentA's factory function, get\_instance to avoid binding to the concrete ComponentA type.
- That means that Executive and ComponentB have no source dependencies on ComponentA. ComponentA can change any of its implementation without affecting Executive or ComponentB as long as the interface, TCompA, and factory function signature, get\_instance, don't change.

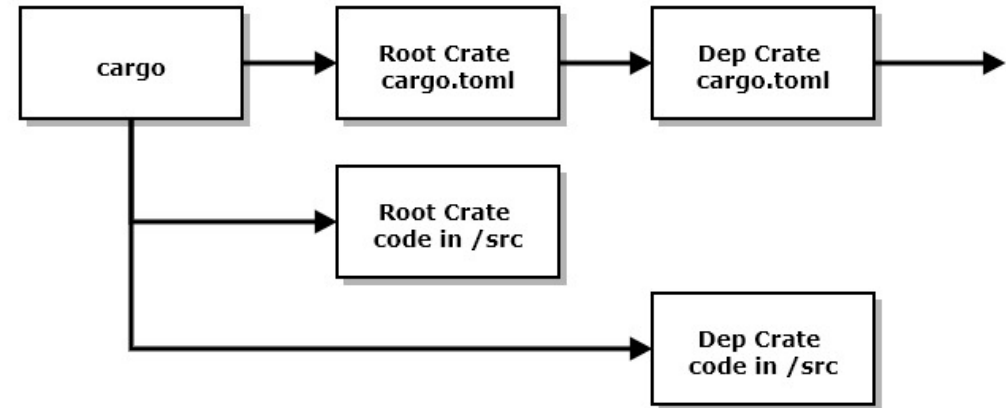


# Build Process

[https://jimfawcett.github.io/RustStory\\_Models.html#build](https://jimfawcett.github.io/RustStory_Models.html#build)

# Compilation Model

- Rust compilation is a transitive depth first search process.
- The cargo build tool starts by parsing the package's cargo.toml file, looking for dependencies and build attribute specifications.
- For each dependency cargo parses its dependencies transitively until it reaches a cargo.toml with no dependencies.
- It then builds that crate root with its loaded modules, then returns to the previous crate in the dependency tree.
- When it returns to the build package it builds the files in /src and deposits its results in /target.
- If any of the dependencies have current builds, that library in /target is used and files in /src are not built.



- Note that cargo.toml files may list zero or more dependencies, so the dependency structure is a tree, not a list.

# Execution Model

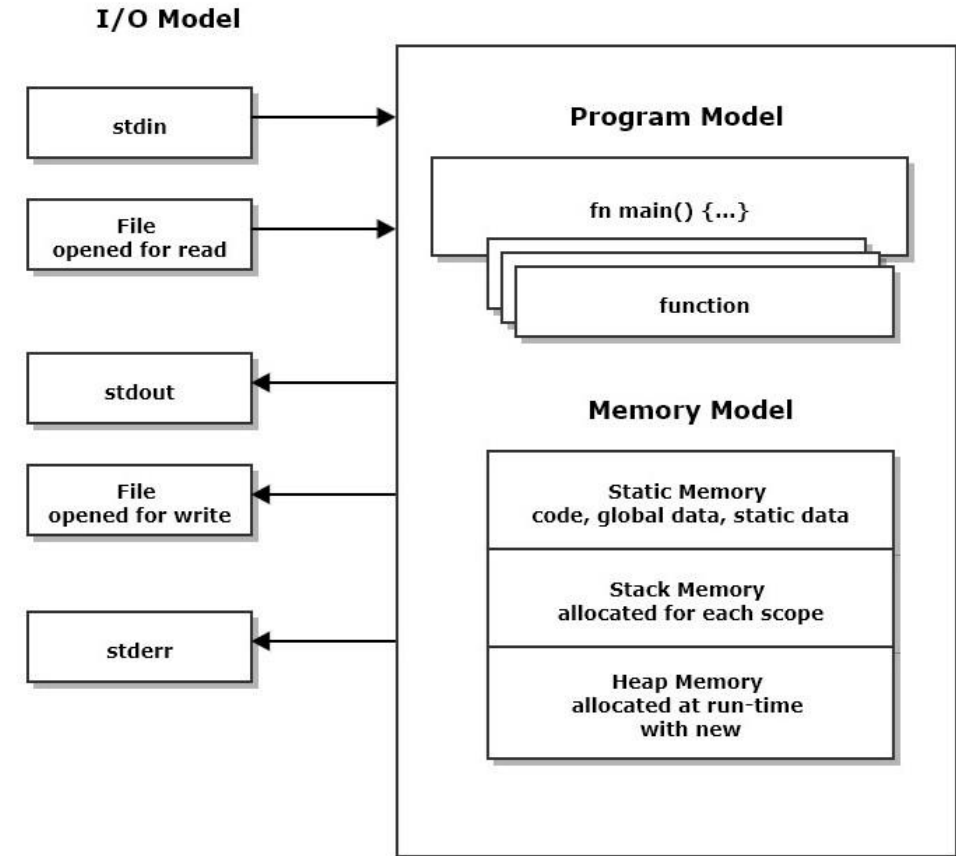
[https://jimfawcett.github.io/RustStory\\_Models.html#execution](https://jimfawcett.github.io/RustStory_Models.html#execution)

# Program Execution

- There are three ways to execute code in a fully formed crate, using cargo:
  - Execution of binaries:  
If the crate root is a binary, e.g., main.rs, the command  
**cargo run**  
will execute the program
  - Testing libraries:  
If the crate root is a library, e.g., lib.rs, the command  
**cargo test**  
will run any tests configured at the end of the library. Tests pass if there are no assertions in the test code, and fail if there are.
  - Running examples:  
For library crates, if you create a /examples folder and put demonstration modules there, then the command  
**cargo run --example an\_example**  
will run the code in an\_example.rs, assuming that you've supplied a main function for that module. The user expects that this code will demonstrate use of library functionality.

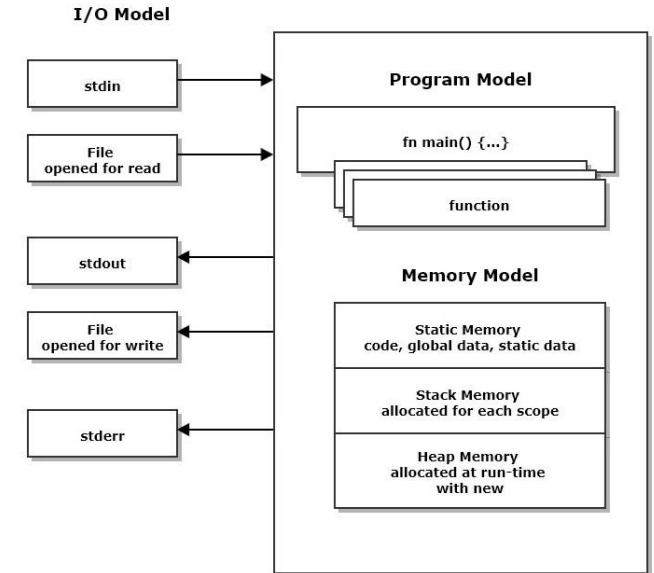
# Program Execution

- When the executable for a program is loaded:
  - Initialization code provided by the compiler executes
  - Then the function main is entered.
    - main is just a function that is defined to the linker as the entry point for processing.
- Any function may call other functions within the executable.



# Use of program memory

- When the thread of execution enters a function an allocation of stack memory is used to store function parameters and any local data defined in the function.
  - The same thing happens for every scope, defined by a matching pair of braces, { and }. For example, an if statement, using braces, allocates stack memory to hold data local to its scope.
- A program may place any of its entities, e.g., an instance of a user-defined type, into static memory, stack memory, or heap memory.
- We will discuss consequences of that later in the next slide.



# Memory Model

- Static memory is used to store code and entities that live for the entire program execution
- Stack memory is used as scratch-pad to store information needed in each scope, e.g., local data. It becomes invalid when the thread of execution leaves the scope.
- Heap memory is used to store entities that live from the time the program creates them with a call to new until the program discards them with a call to delete

| Static Memory            |
|--------------------------|
| Code<br>global constants |
| global data              |
| static local data        |

| Stack Memory                               |
|--------------------------------------------|
| main stack frame                           |
| main's control<br>stack frames             |
| function called in main<br>stack frame     |
| function's control<br>stack frames         |
| function called by function<br>stack frame |

| Heap Memory        |
|--------------------|
| allocated memory   |
| unallocated memory |

# Control of entity placement in memory

- The compiler places all code and global data in static memory.
- A program can place an entity instance in static memory by qualifying its declaration with the keyword `static`. Rust statics should be immutable.
- Rust code also places entities in stack memory by calling a function, placing function parameters and local data in its stack frame.
- Also, every local scope, defined by braces, `{` and `}`, creates a new allocation of stack memory to hold data local to that scope.
- An entity instance is placed in heap memory by declaring `Box::new(Point { x: 1.0, y:2.0, z:3.5 })`. The heap allocation is returned when the `Box` instance goes out of scope.

| Static Memory            |
|--------------------------|
| Code<br>global constants |
| global data              |
| static local data        |

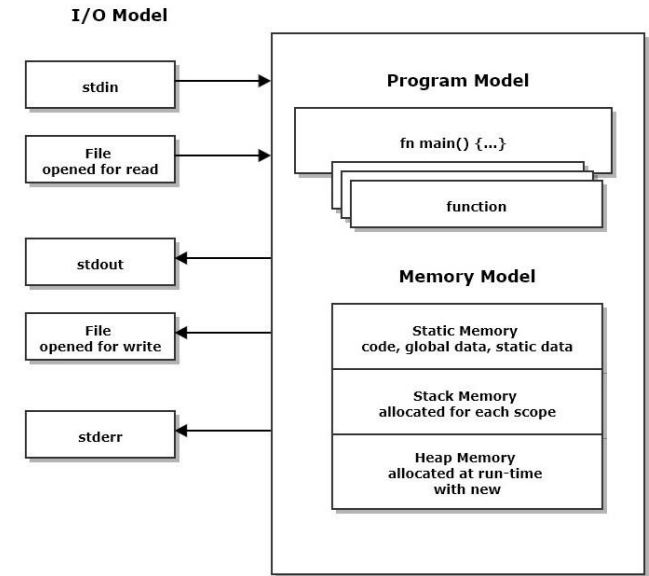
| Stack Memory                               |
|--------------------------------------------|
| main stack frame                           |
| main's control<br>stack frames             |
| function called in main<br>stack frame     |
| function's control<br>stack frames         |
| function called by function<br>stack frame |

| Heap Memory        |
|--------------------|
| allocated memory   |
| unallocated memory |



# Interaction with the Execution Environment

- There are two primary ways for a Rust program to observe and use its execution environment:
  - Use a stream object like `std::stdin` or `std::stdout`.
  - Types for streams are provided by the standard library, via import statements:  
use `std::io::prelude::*`, use `std::fs::File`, ...
- The program may use services of its platform API by using `std::ffi` (Foreign Function Interface) in an `unsafe` block or by using a crate that wraps that:
  - <https://github.com/retep998/winapi-rs>



# Epilog

[https://jimfawcett.github.io/RustStory\\_Models.html#epilogue](https://jimfawcett.github.io/RustStory_Models.html#epilogue)

# Conclusions

- If you understand the 7 models, we've covered, I think you will find Rust syntax and semantics to be convenient and sensible.
- Some particular parts of the language discussed in the Rust Story but not here are intricate and require some study to master:
  - String syntax and semantics because the only character type Rust recognizes in its native strings is utf-8, which uses multi-byte characters of varying sizes.
  - Life-time annotation needed for some scenarios using generics.
  - Many crates in <https://crates.io> are used routinely by knowledgeable Rust developers, but some take significant amounts of time and effort to use effectively.
- Rust avoids undefined behavior by incorporating a safe type system. That is based on the ownership rules we've discussed. It takes a while to get use to the rules, but compiler error messages are usually very good.

# Presentation Resources

- The ideas discussed in this presentation are drawn from a web page:

[https://jimfawcett.github.io/RustStory\\_Models.html](https://jimfawcett.github.io/RustStory_Models.html)

which is part of the Rust Story:

[https://jimfawcett.github.io/RustStory\\_Prologue.html](https://jimfawcett.github.io/RustStory_Prologue.html)

- And code examples for the story are documented here:

<https://jimfawcett.github.io/RustBasicDemos.html>

- These slides are available here:

<https://jimfawcett.github.io/Resources/RustModels.pdf>

# Background

- The material for this presentation comes from the github website:
  - <https://JimFawcett.github.io>,  
<https://jimfawcett.github.io/Resources/RustModels.pdf>
- The site provides a curated selection of code developed for graduate software design courses at Syracuse University
- It also contains tutorial and reference materials related to that code.
- Some of that is presented in the form of “stories”
- Rust Models is the title of the first chapter of a “[Rust Story](#)”
  - The story is a detailed walk-through of the Rust programming language. It provides reference material for a set of [repositories](#) that hold source code for utilities, tools, components, and demonstrations.