Rust Models

Jim Fawcett

https://JimFawcett.github.io

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

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

- Models help us understand important features
  - Use language effectively
  - Accelerate learning process

Models Prologue

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

- Rust is an interesting and ambitious language.
- We will consider Rust 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

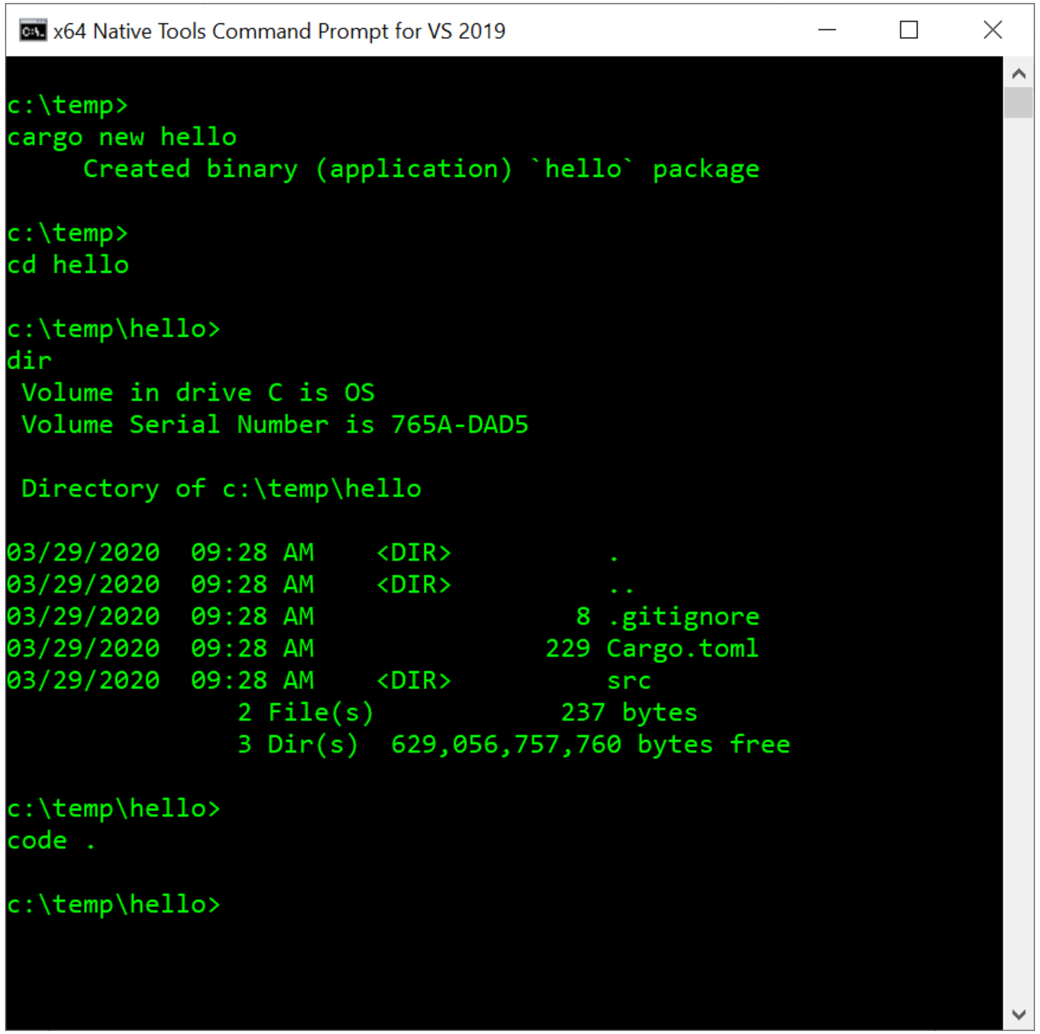
Why Rust?

- 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++

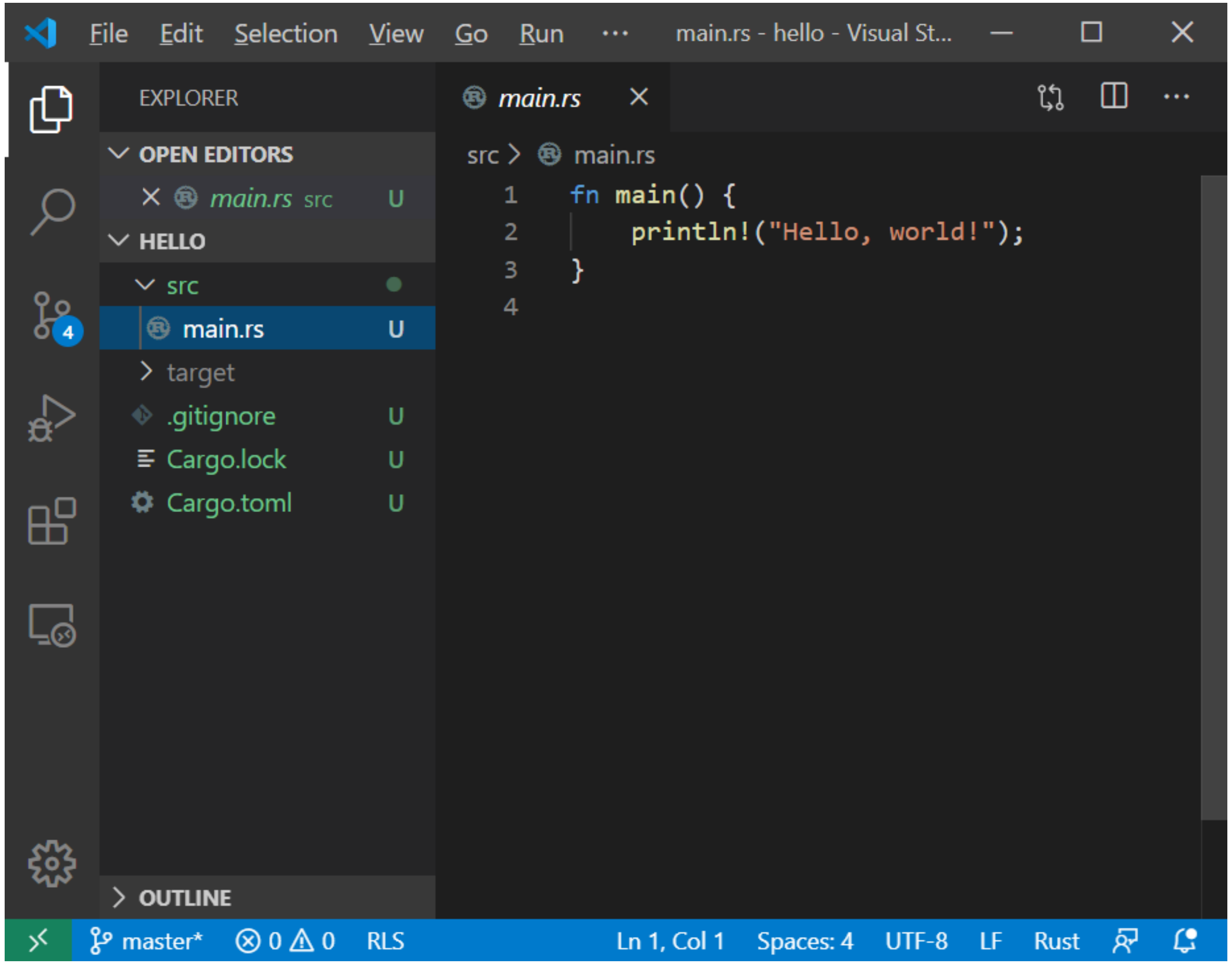
Hello Rust World!

- 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

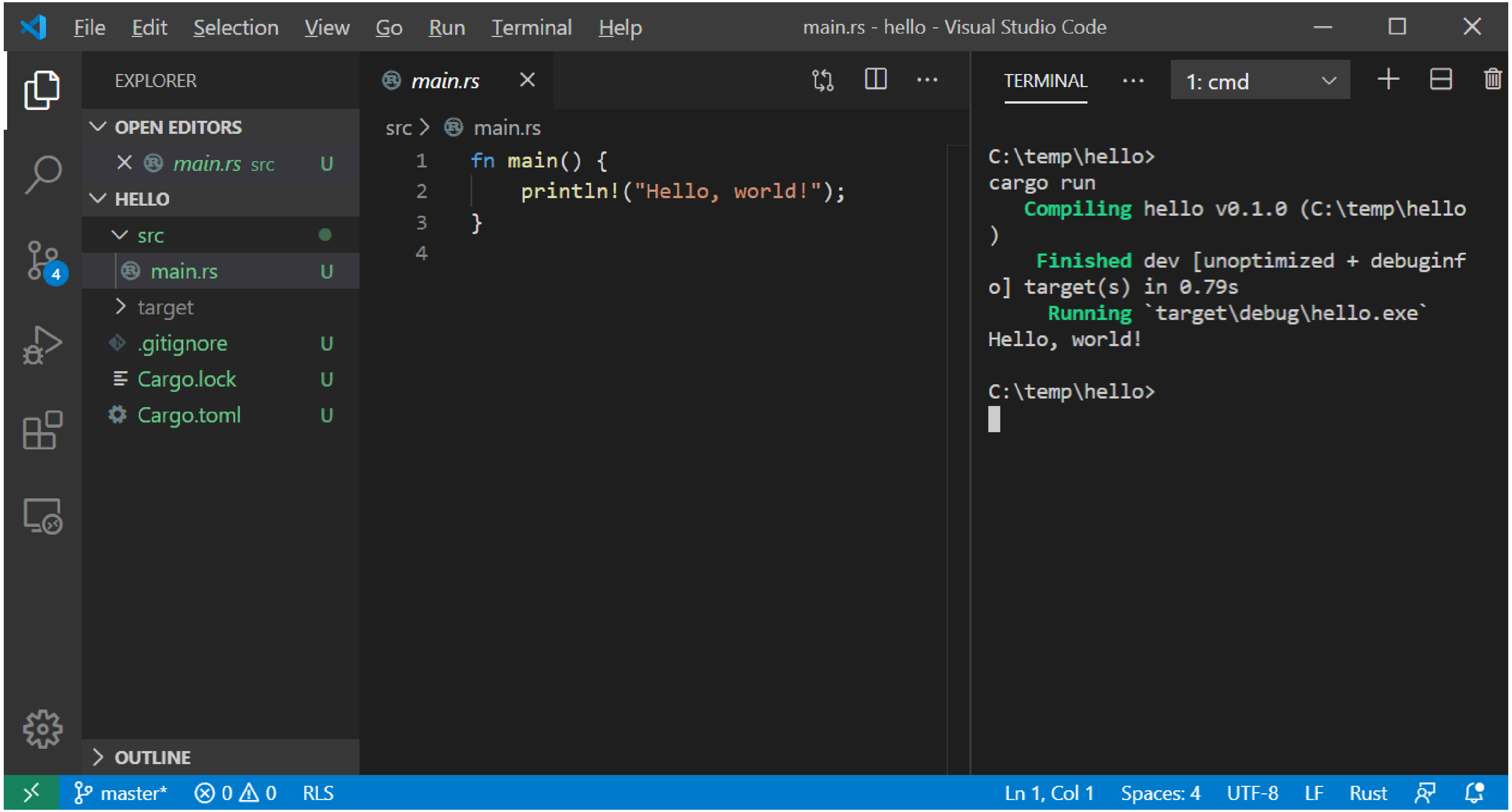


x64 Native Tools Command Prompt for VS 2019
c:\temp>
cargo new hello
Created binary (application) `hello` package
c:\temp>
cd hello
c:\temp\hello>
dir
Volume in drive C is OS
Volume Serial Number is 765A-DAD5
Directory of c:\temp\hello
03/29/2020 09:28 AM <DIR> .
03/29/2020 09:28 AM <DIR> ..
03/29/2020 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>

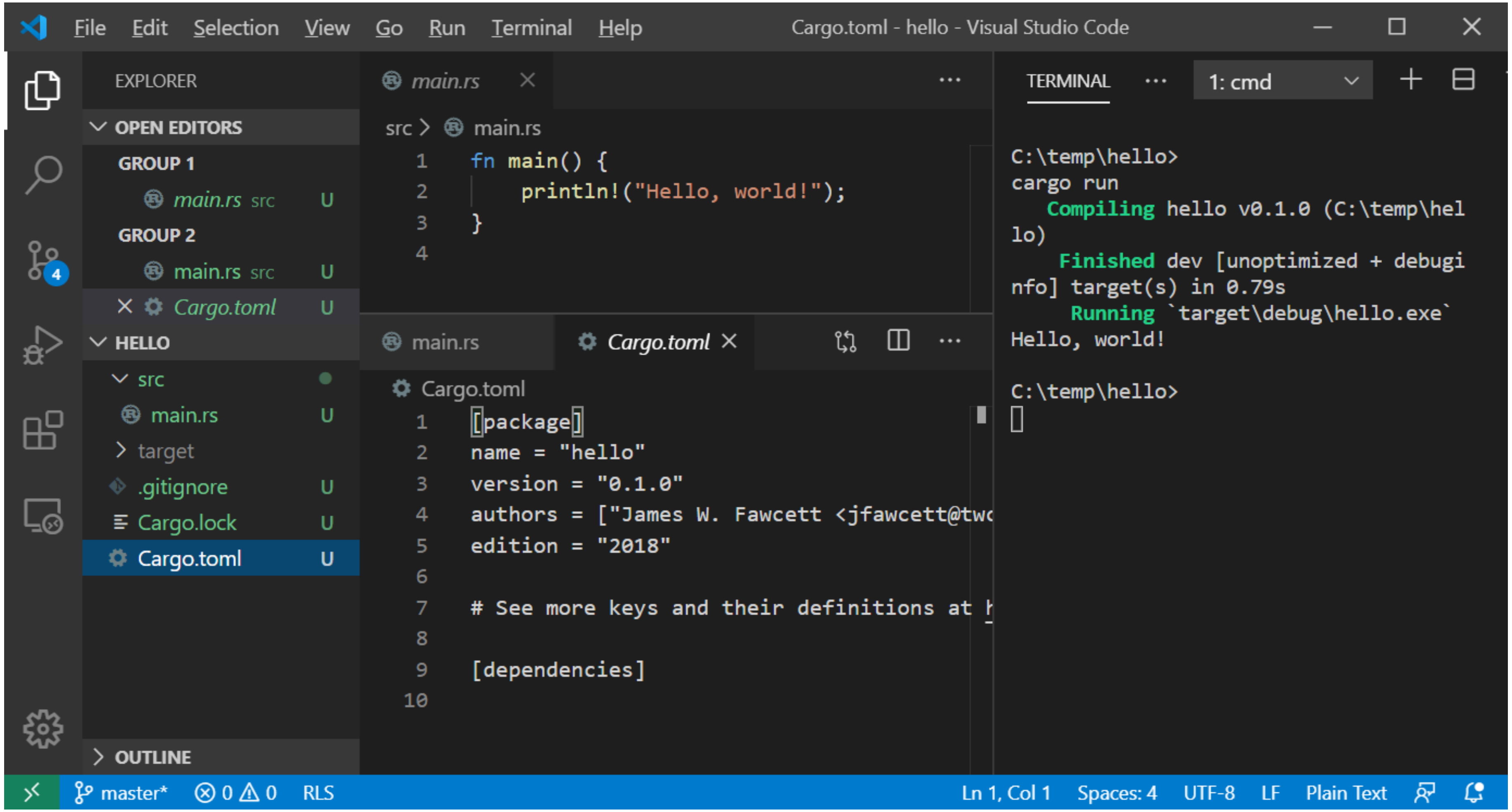


File Edit Selection View Go Run ··· main.rs - hello - Visual St...
EXPLORER
main.rs
OPEN EDITORS
main.rs src U
HELLO
src
main.rs U
target
.gitignore U
Cargo.lock U
Cargo.toml U
OUTLINE
src > main.rs
1 fn main() {
2 println!("Hello, world!");
3 }
4
master* 0 0 RLS Ln 1, Col 1 Spaces: 4 UTF-8 LF Rust

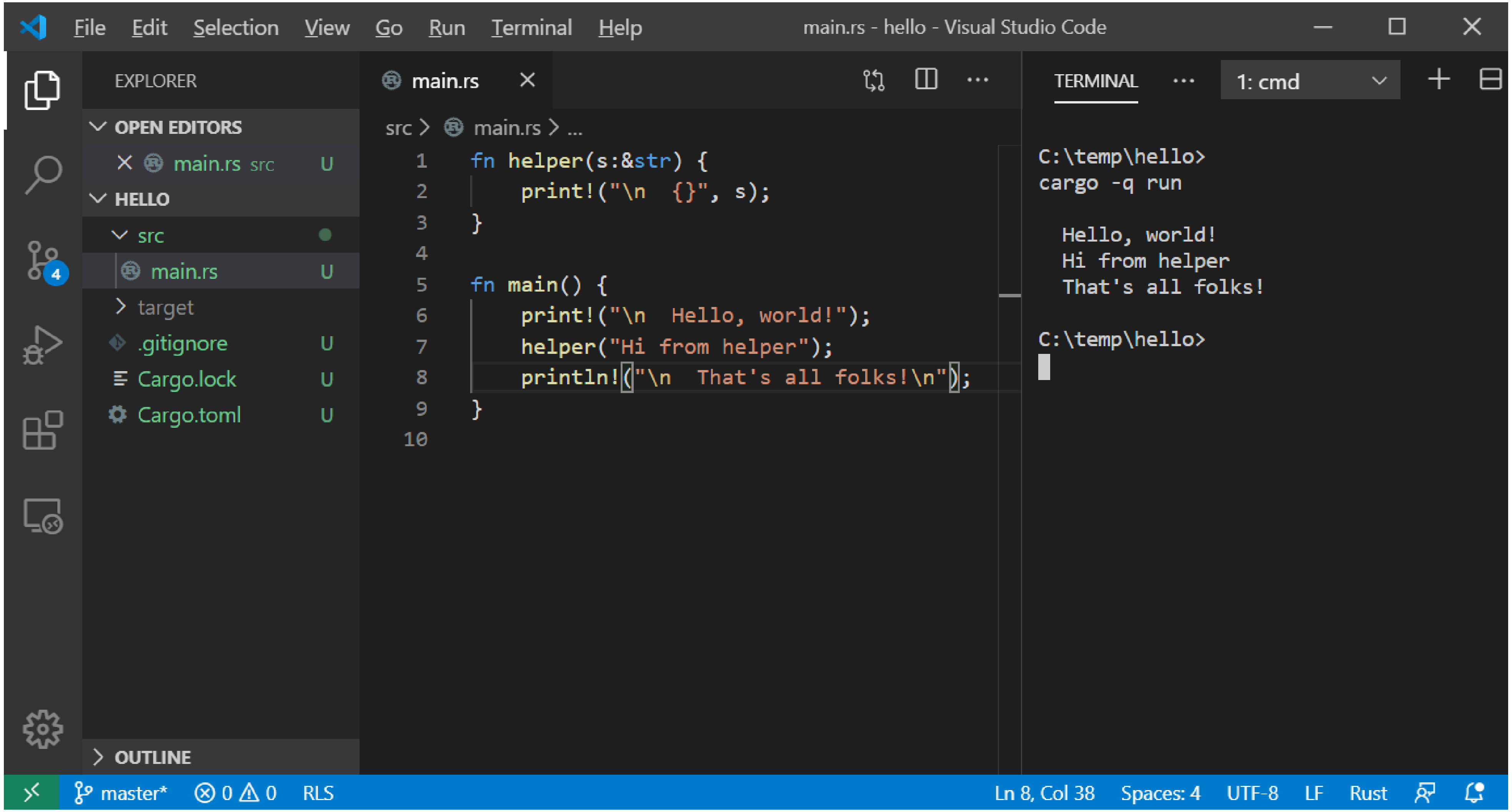
Building and Running with Cargo



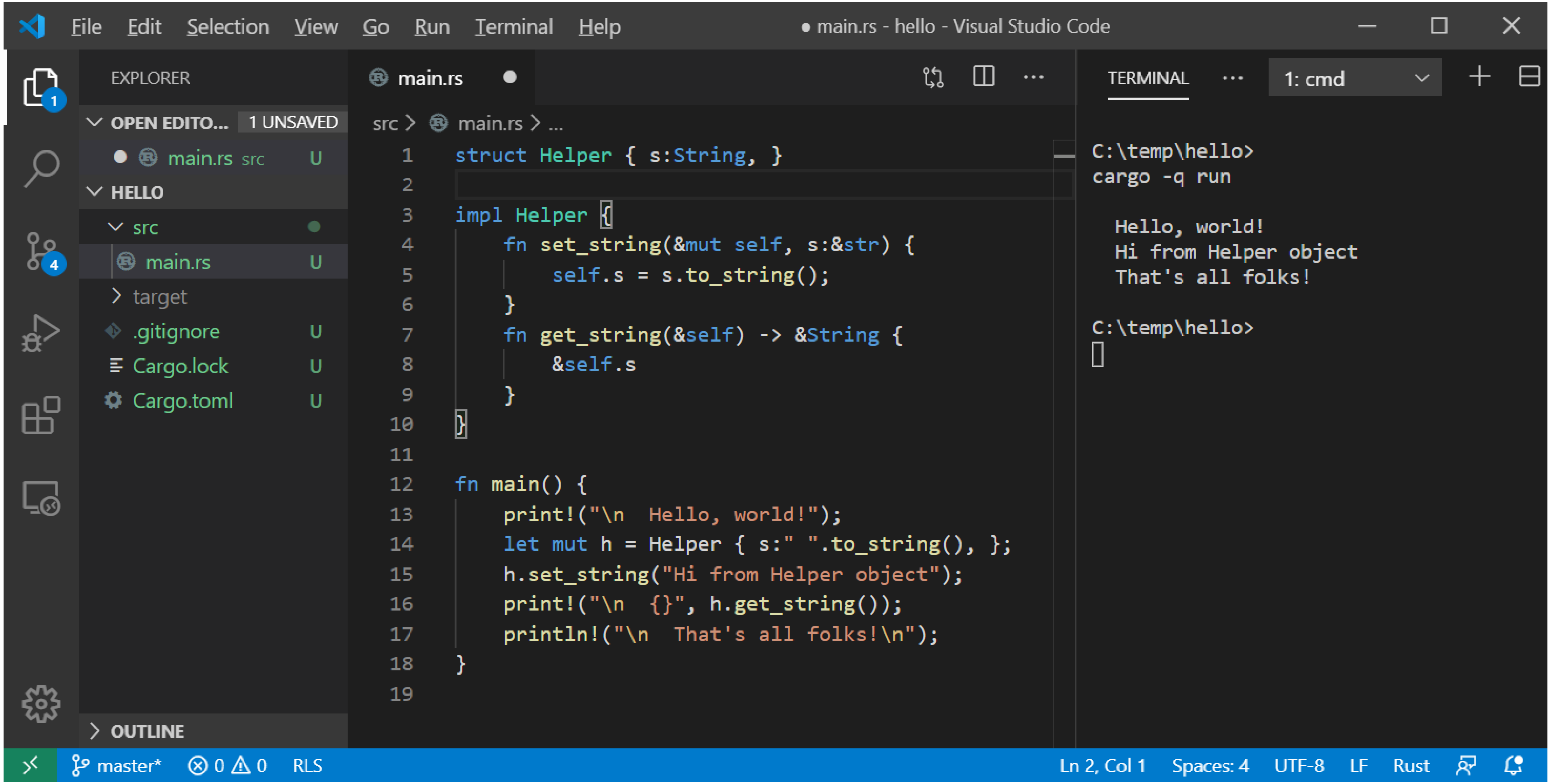
Cargo.toml – defines package



Add another function



Modify to use “object”



More string handling – see lines 17-18

```rust
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
```

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

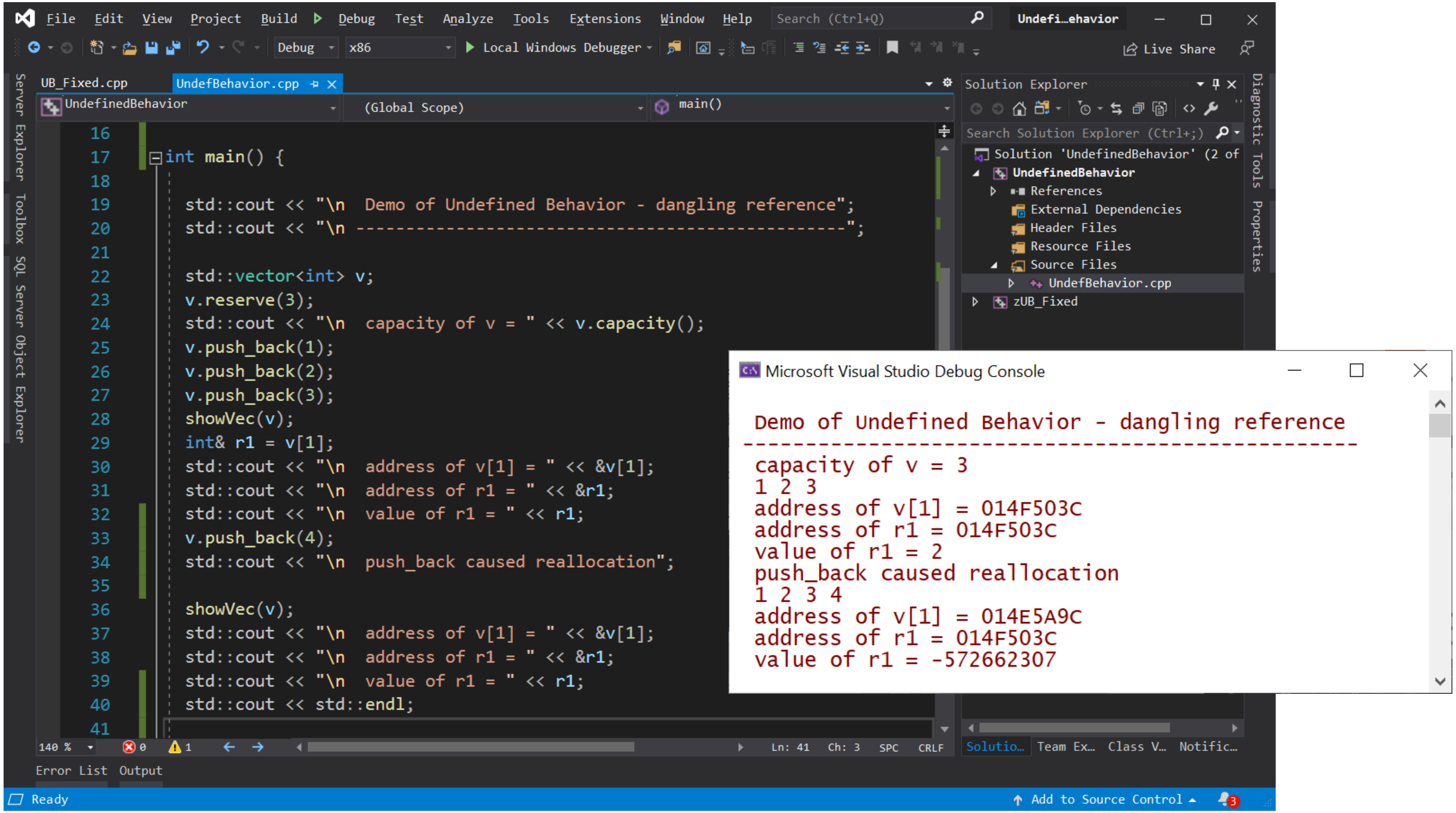
  Hello, world!
  Hi from Helper object
  Hello from Helper object
  That's all folks!

C:\temp\hello>
```

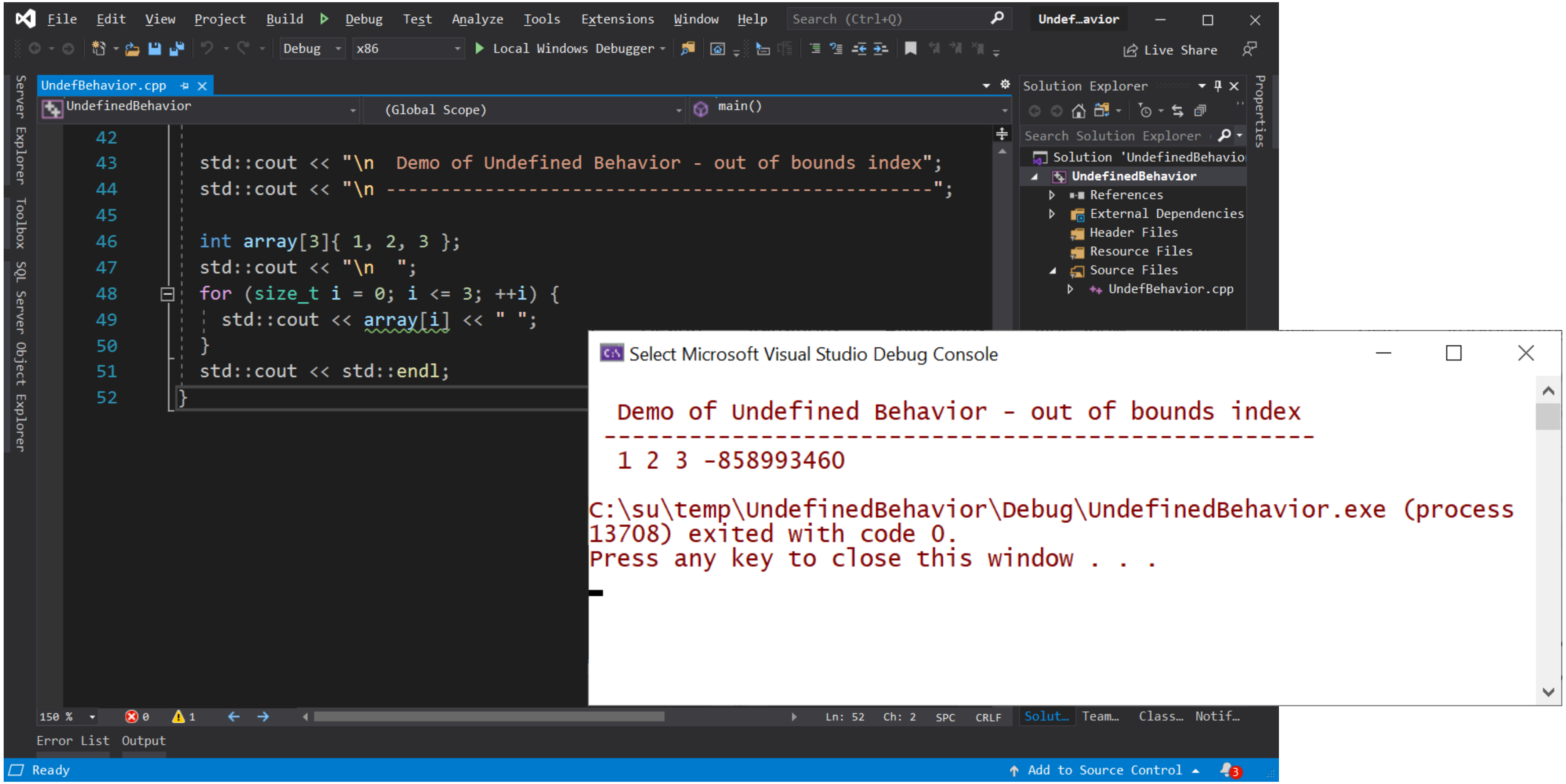
Why Rust?

- 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



Undefined Behavior – C++ index out of bounds



In defense of C++ - Dangling Reference

- If we had used an iterator:
  - `auto iter1 = ++v.begin();`
  - `v.push_back(4);`
  - `Std::cout << *iter1; // throws exception – no undefined behavior`
- It is standard practice to access containers with iterators, so well-crafted C++ will not exhibit undefined behavior.
- The difference:
  - With Rust you can’t get undefined behavior (UB) – most often programs fail to compile if they would have UB.
  - With C++ code has to be well-crafted to avoid UB, errors are discovered at run-time, not compile-time.

In defense of C++ - Index out of Bounds

- If we had used a range-based for loop:
  - ```
    for(auto item : array) {
        std::cout << item << " ";
    }
    ```

  there is no chance of out-of-bounds indexing
- It is standard practice to traverse containers with range-based for loops, so well-crafted C++ will not exhibit undefined behavior.
- The difference:
  - With Rust you can't get undefined behavior (UB) – out of bounds index causes panic (exit) with no chance to access unowned memory.
  - With C++ code has to be well-crafted, using standard idioms, to avoid UB.
    ```

Ownership Model

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

Rust Ownership

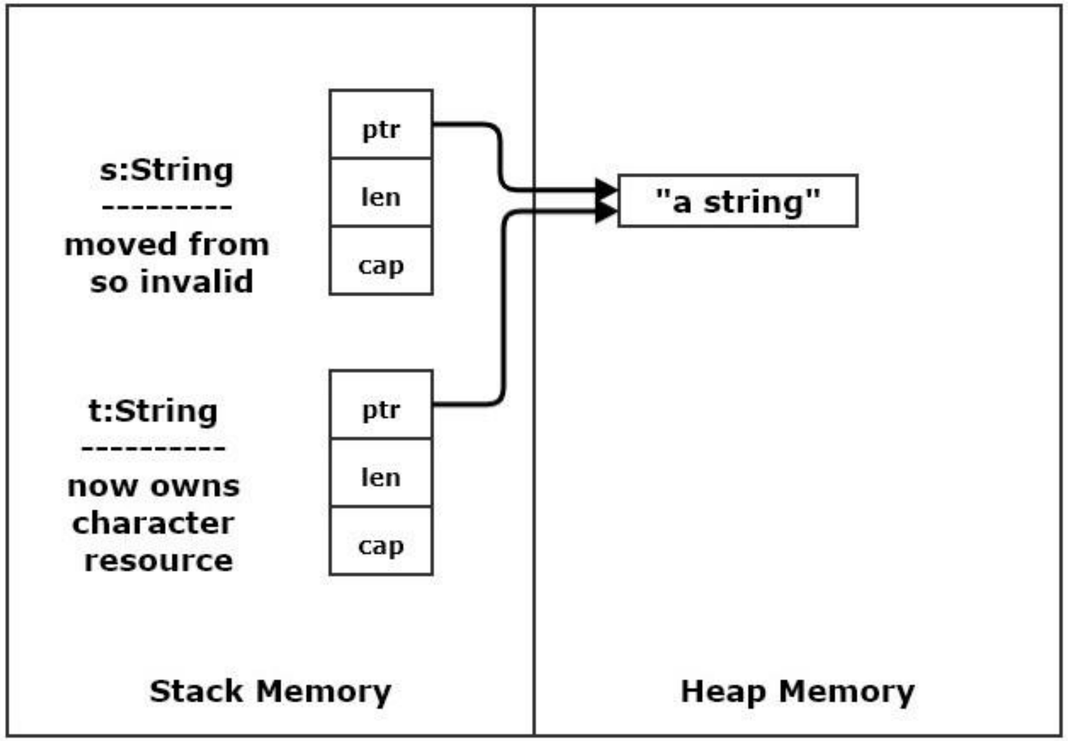
- Ownership rules are, in principle, quite simple:
  - Rust enforces **Read-Write-Locks** on data access at compile-time.
  - Any number of readers may access value simultaneously.
  - Writers get exclusive access to value – no other readers or writers.
- What are readers and writers?
  - Any variable bound to a value with no mut qualifier is a reader.
    - Original owner: `let s = String::from(“a string”);`
    - References to the data: `let r = &s;`
  - Any variable bound to a value with mut qualifier is a writer:
    - Original owner: `let mut s = String::from(“another string”);`
    - References to the data: `let mut r = &s;`

Copies, Moves

- Copy
  - Data resides in one contiguous block of memory (blittable)
  - `let x = 3.5;`
  - `let y = x;`
  - y gets copy of x’s value ==> two separate locations holding the same value.
  - Copy binding creates new owner of new data.
- Move
  - Data resides in two or more blocks, usually one in stack, one in heap.
  - `let s = String::from(“a string”);`
  - `let t = s;`
  - s value moved to t, s becomes invalid
  - Move binding transfers ownership

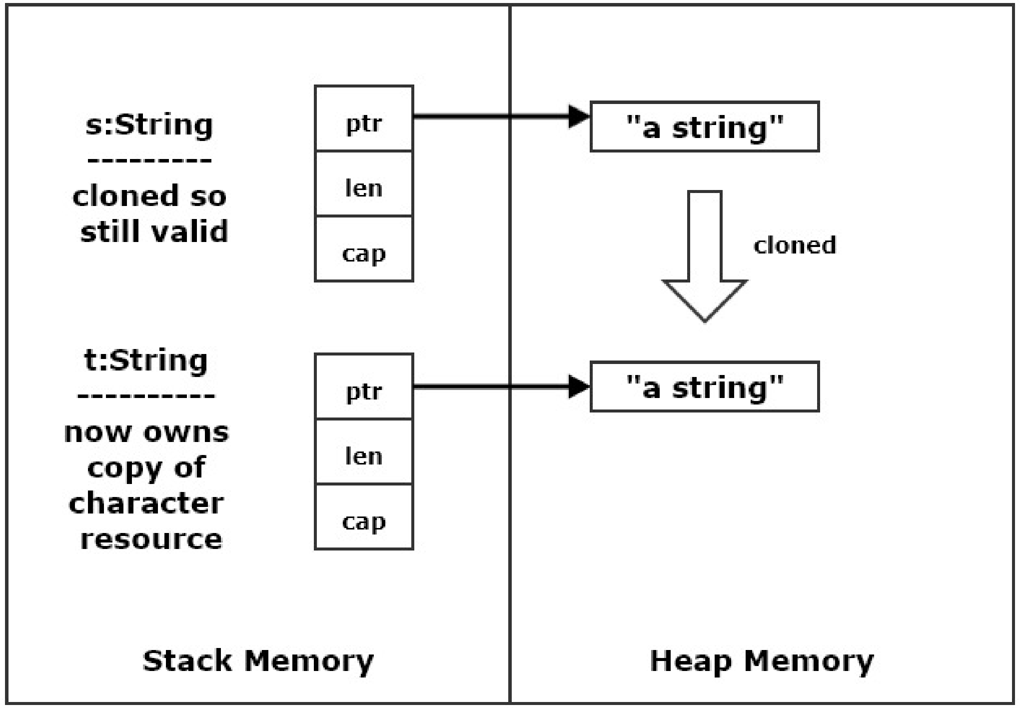
Rust Move versus Copy

- Rust will copy any value contained in a single contiguous block of memory (blittable)
  - `let x = 2;`
  - `let y = x;    // copy`
- Any value requiring separate parts, like the string shown in the right panel will be moved.
  - `let s = String::from(“a string”);`
  - `let t = s;`
    `// value moved from s`
    `// t owns string, s invalid`



Rust Clone

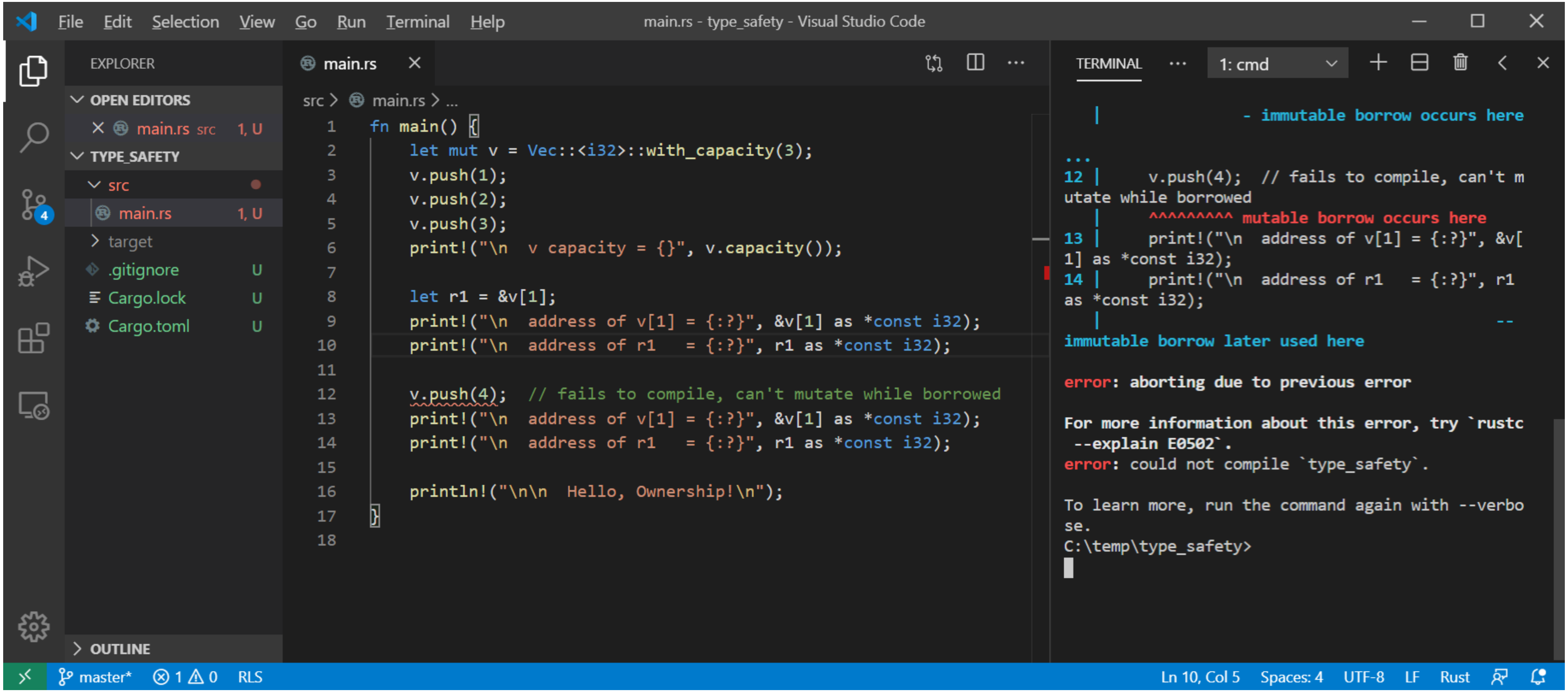
- Often a type satisfies clone trait (if not you can add that).
- This allows moves to be avoided by explicitly calling clone() to make a copy.
  - `let t = s.clone(); // s still valid`
- Clone must always be called explicitly. Rust wants you to know when you invoke an expensive operation.



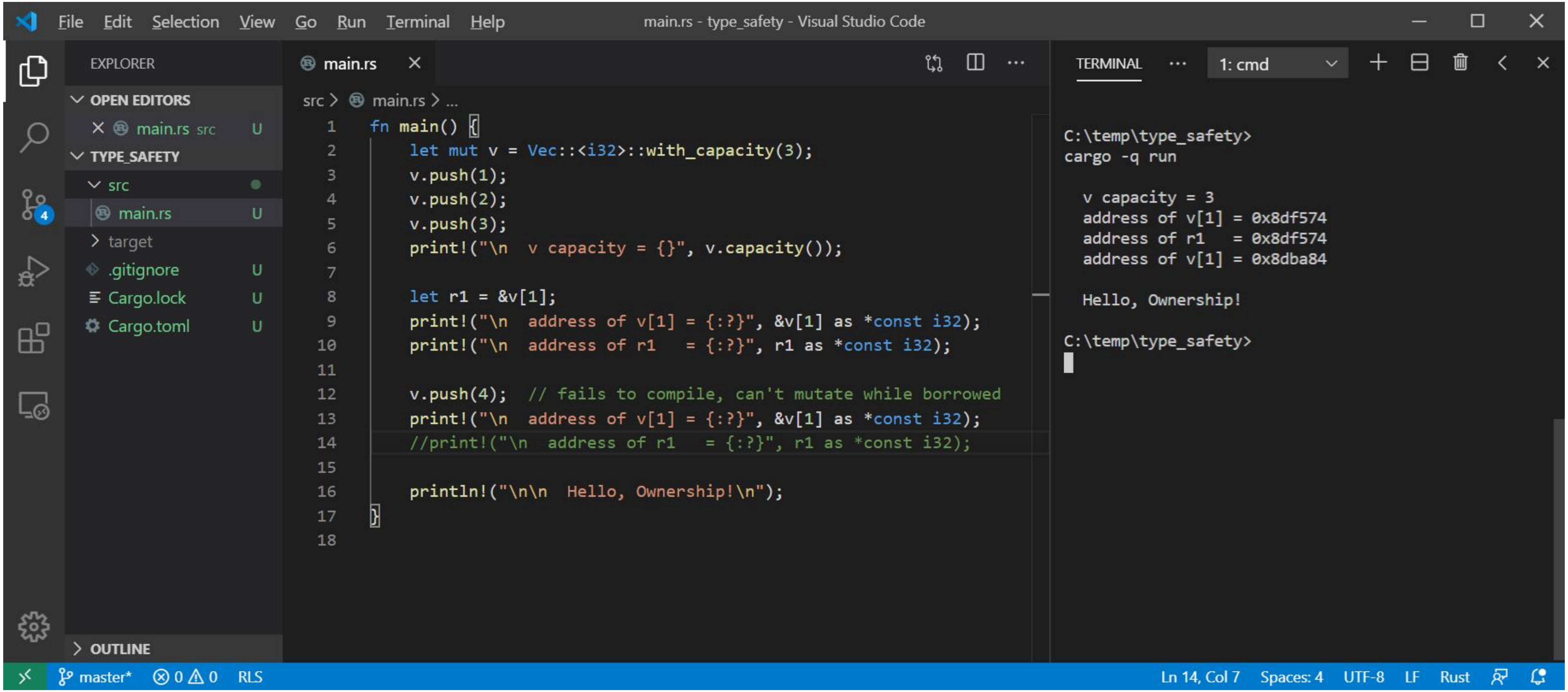
References and RwLocking

- Non-mutable Vec and references - all readers:
  - `let v = vec![1,2,3];`
  - `let r1 = &v;  let r2 = &v;   // each has view of v's data`
- Mutable Vec, non-mutable references – creating reference inhibits Vec mutation:
  - `let mut v = vec![1,2,3];`
  - `let r1 = &v;  let r2 = &v;   // each has view of v's data`
  - r1 and r2 borrow v's data // v cannot mutate while borrows are active
  - Borrows end when they go out of scope or are dropped, drop(r1);
- Mutable data, mutable reference – writer v's ability to write borrowed
  - `let mut v = vec![1,2,3];`
  - `let mut r = &v;  // r has borrowed v's ability to mutate`
  - v cannot mutate until borrow ends

Rust won’t allow mutation with an active reference



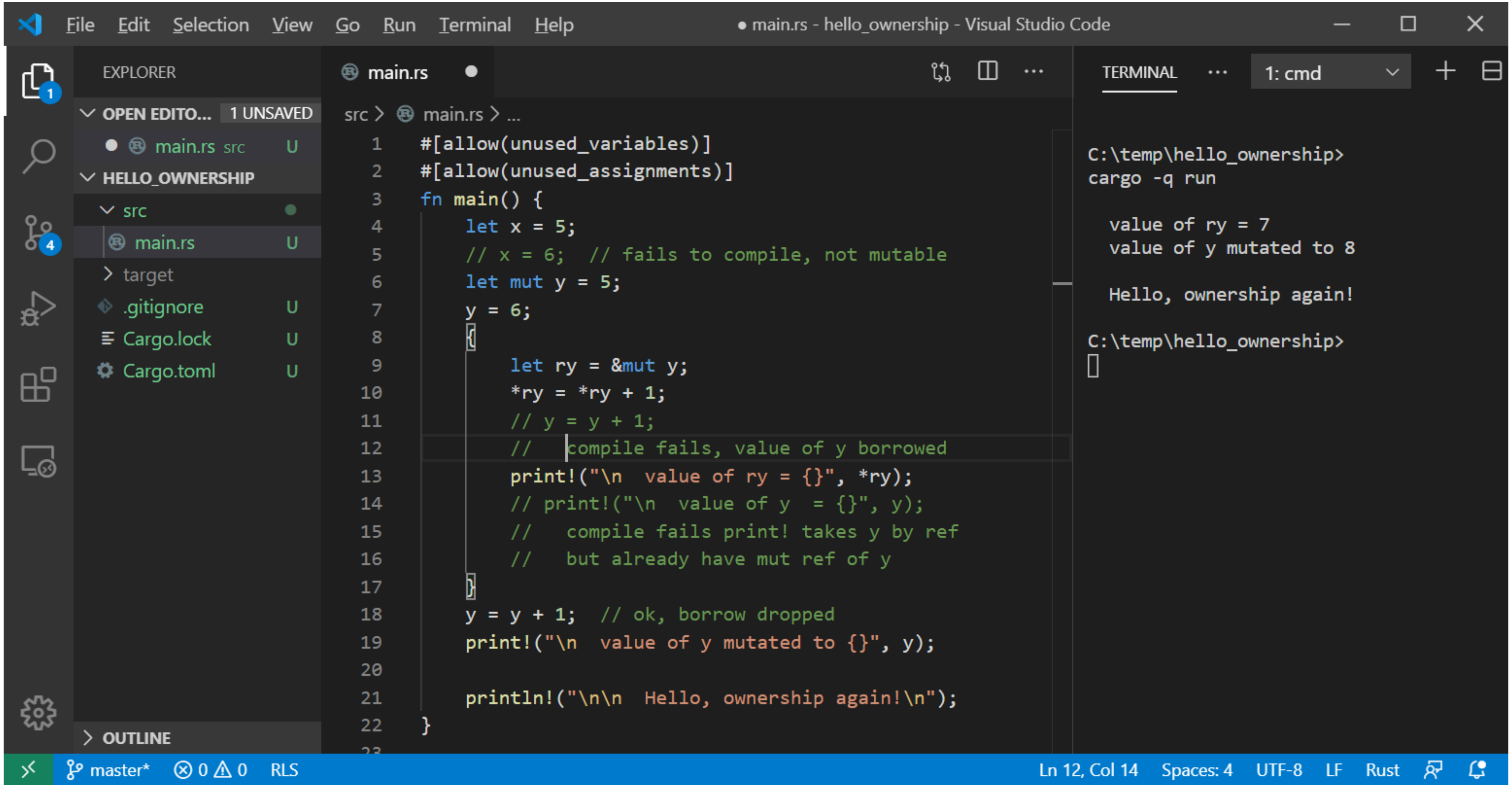
Rust allows mutation if we don't use the reference



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



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 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 refence to avoid side-affects.
  - 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.

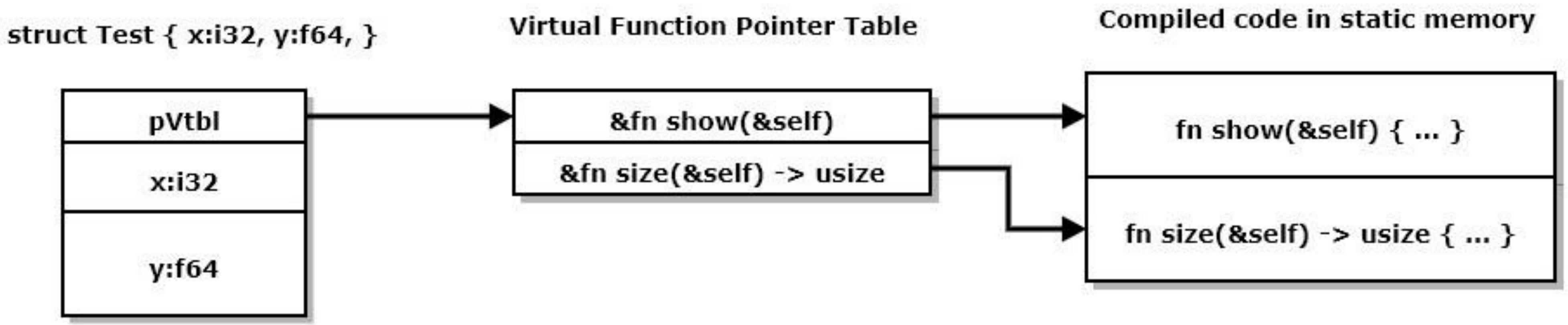
Rust Object Model

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

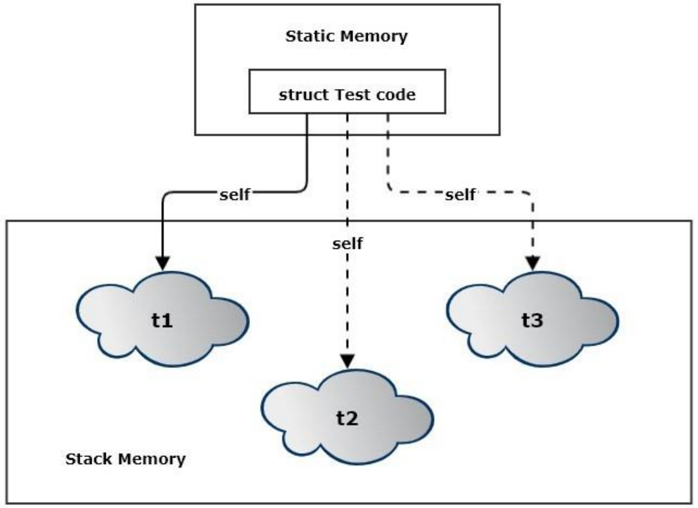
Implementing Traits and Methods

- ```
  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, }
      }
  }
```

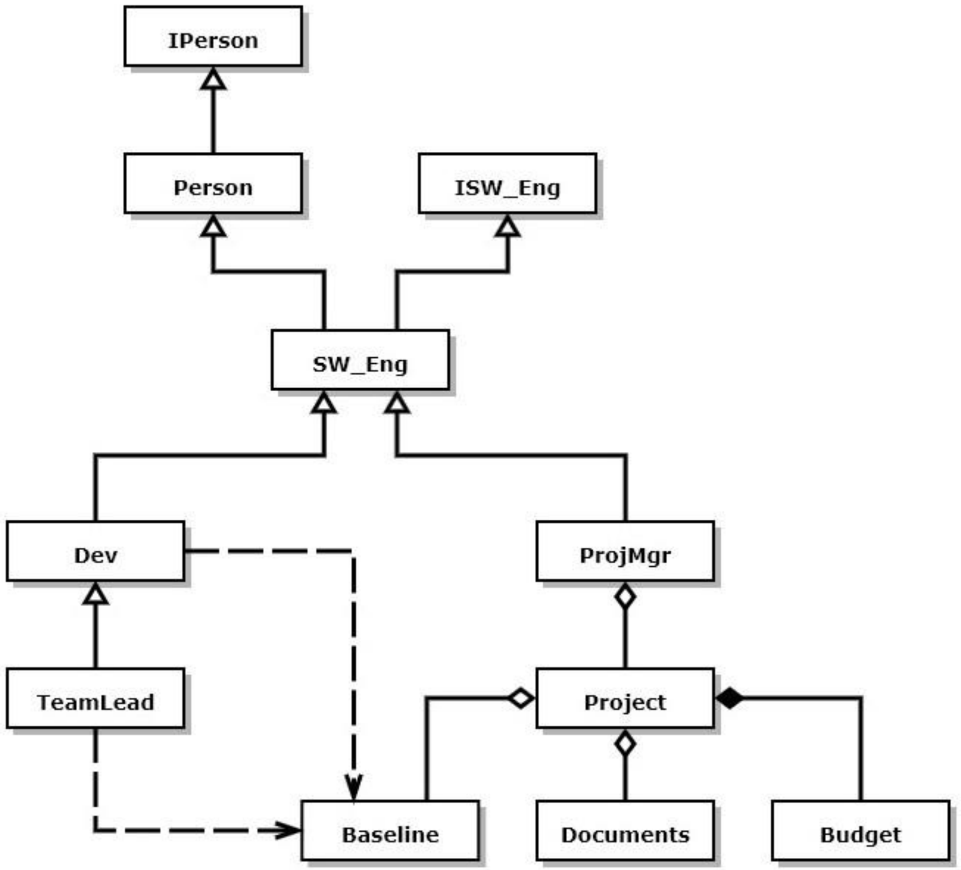
Comparison with C++

- 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: Debug>(_value:&T) {
        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 }
    ```
    ```

Traits

Code 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

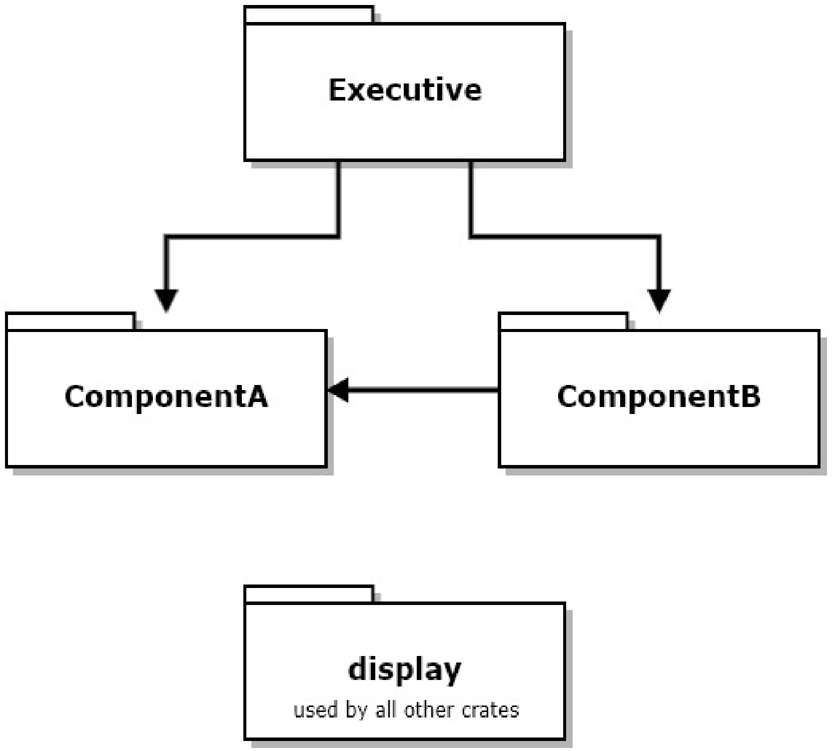
- A Package is a collection of directories and files that are the basis for builds
    - Cargo.toml – specifies package metadata, dependencies, and optional directives
    - /src – directory containing a binary or library source crate
    - /target – directory containing translated binaries or libraries
    - /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.

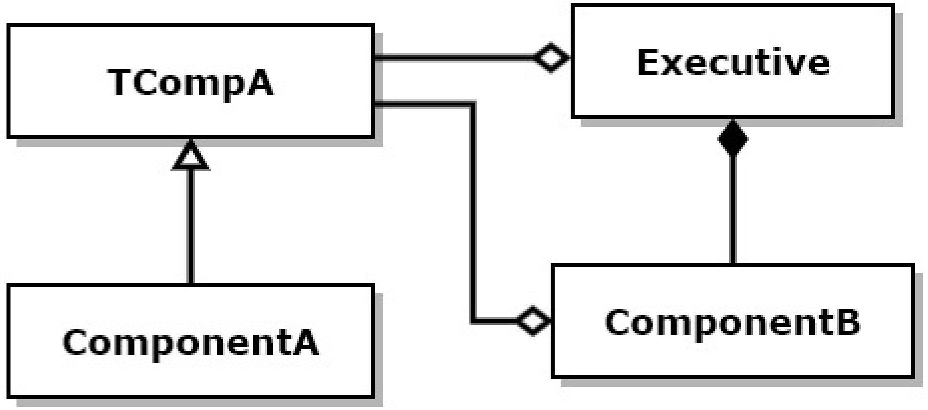
Example – Crates and Packages

- 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

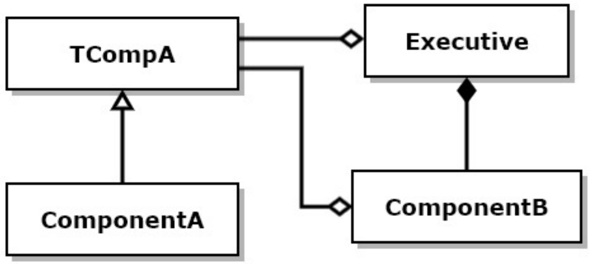
- This diagram shows structs that are defined in each of the files from the previous slide.
  - TCompA is an interface[1] trait for ComponentA
  - ComponentA implements the trait to provide exported services
  - ComponentB doesn't provide an interface
  - ComponentB uses ComponentA through its interface trait and factory[2]
  - 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())
}
```

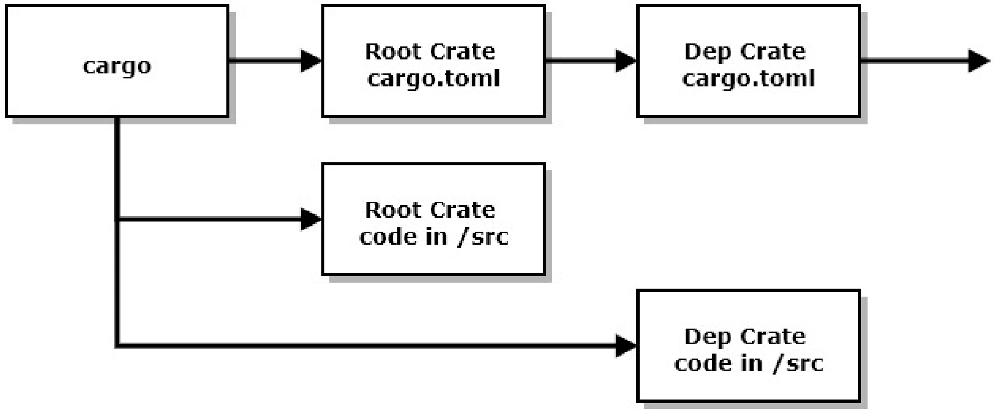
- Executive and ComponentB use ComponentA's factory function, get_instance to avoid binding to the concrete ComponentA type.
- That means that Executive and ComponentB have no source dependencies on ComponentA. ComponentA can change any of its implementation without affecting Executive or ComponentB 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

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

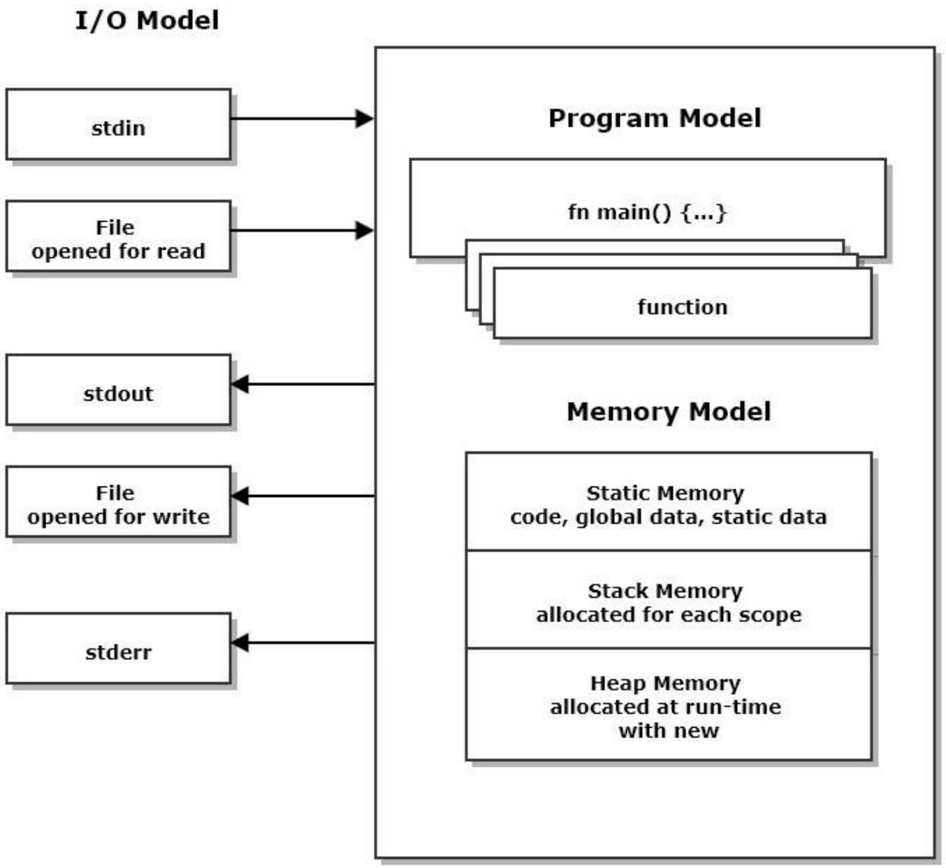
https://jimfawcett.github.io/RustStory_Models.html#execution

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.

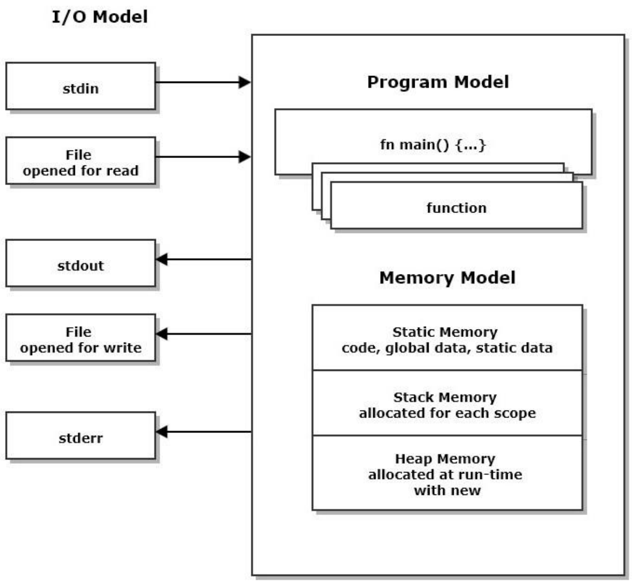
Program Execution

- 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.



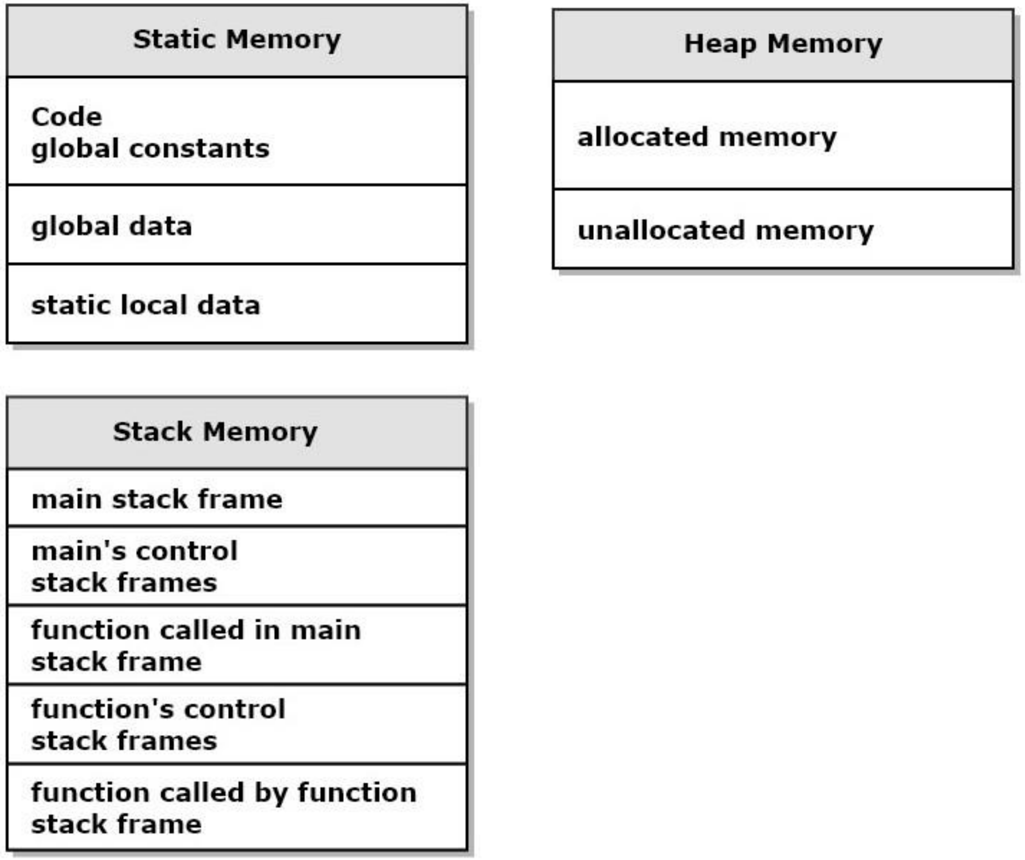
Use of program memory

- 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.



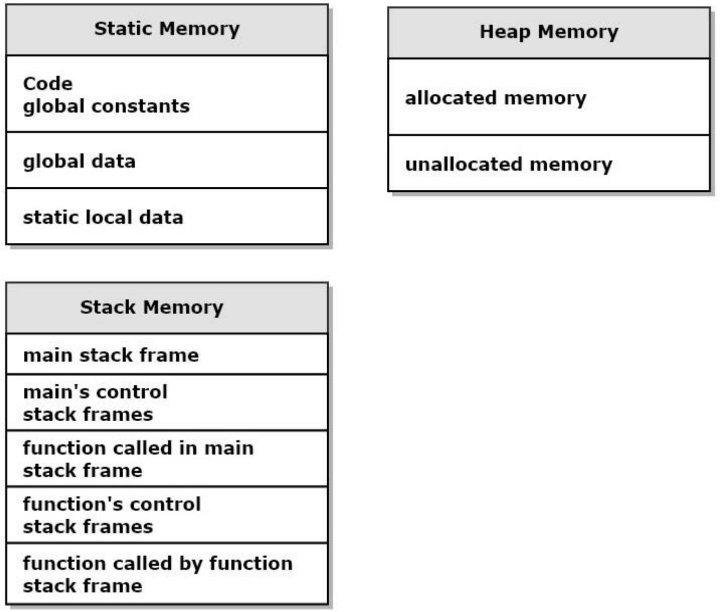
Memory Model

- 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



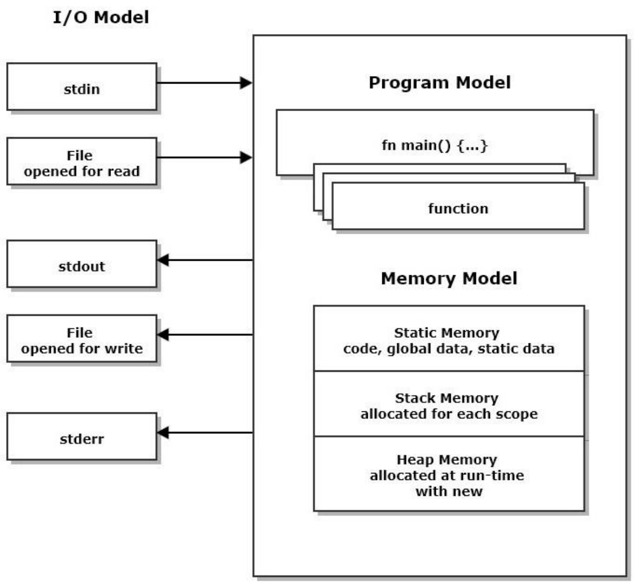
Control of entity placement in memory

- 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.



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

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

  which is part of the Rust Story:
  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.