A Tour of Rust

the programming language

Jim Fawcett

https://JimFawcett.github.io

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

Tour Prologue

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

- Rust is an interesting and ambitious language, similar to C++, but with some unique differences.
  - Compiles to native code, no need for garbage collection
  - Emphasis on performance
- Rust features:
  - Type Safety – unable to create undefined behavior, by construction
    - Ownership model for all values
  - Objects
    - The language provides the usual set of primitive types
    - All library and user types are created from structs and enums
  - Generics
    - Similar to Java and C# generics, rust has broad support for trait constraints
- Rust tool chain provides Cargo, a package manager, builder, and executor

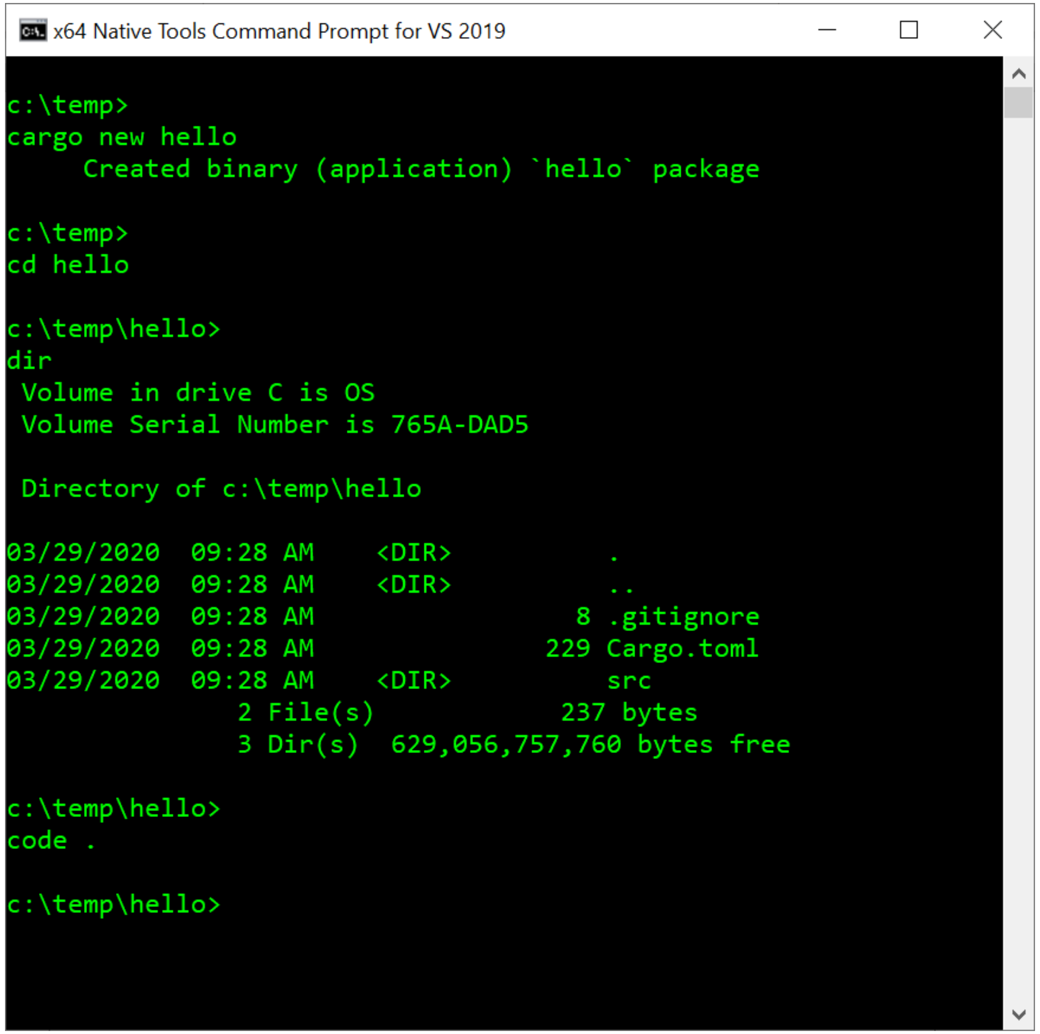
Why Rust?

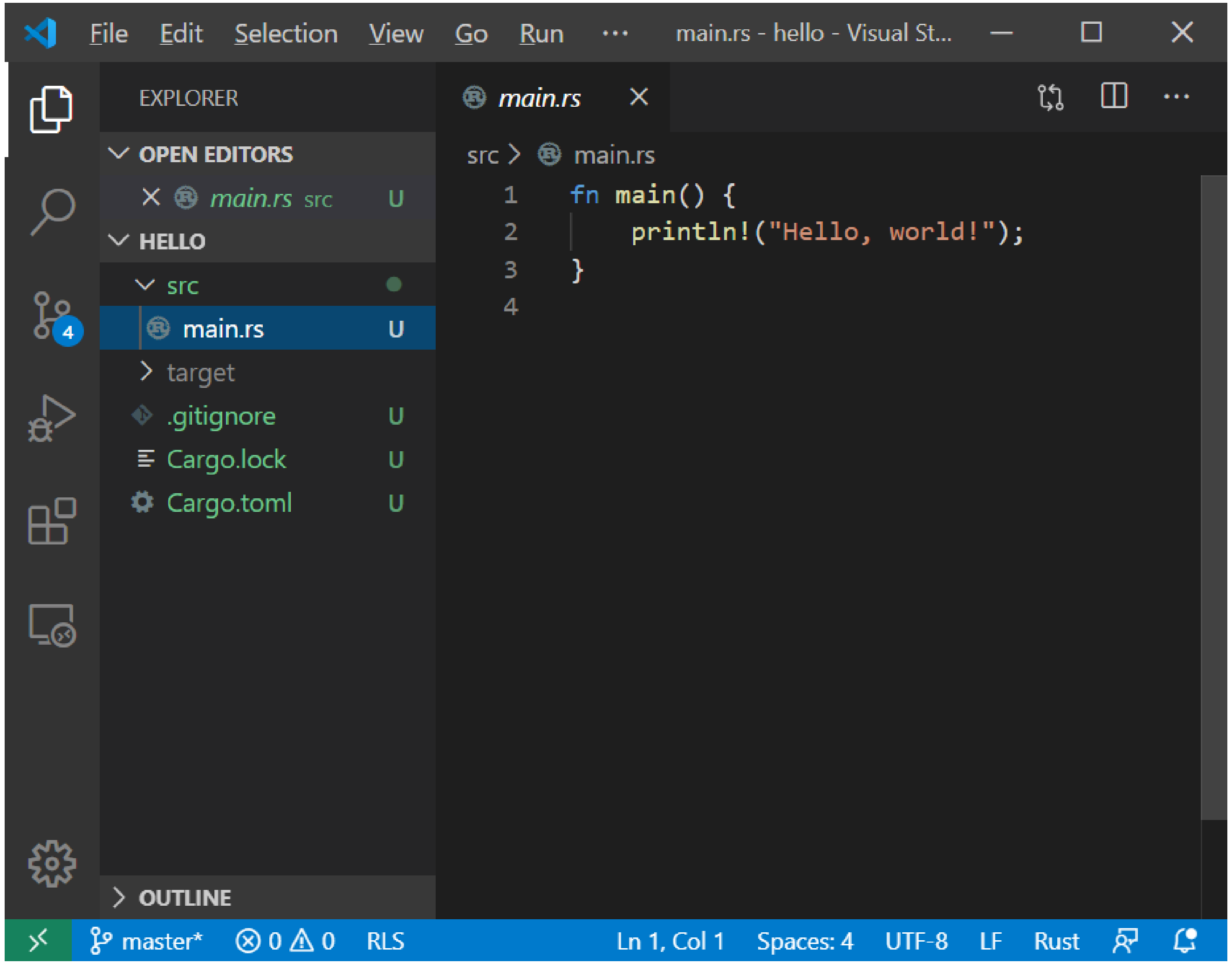
- Memory Safety
  - No dangling pointers or null references
  - No reading or writing to unowned memory
  - Rust’s type system enforces sane ownership policies.
- No Data Races
  - The same ownership policies applied to thread interactions ensures data race free operation
- Performance
  - As fast as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

Hello Rust World!

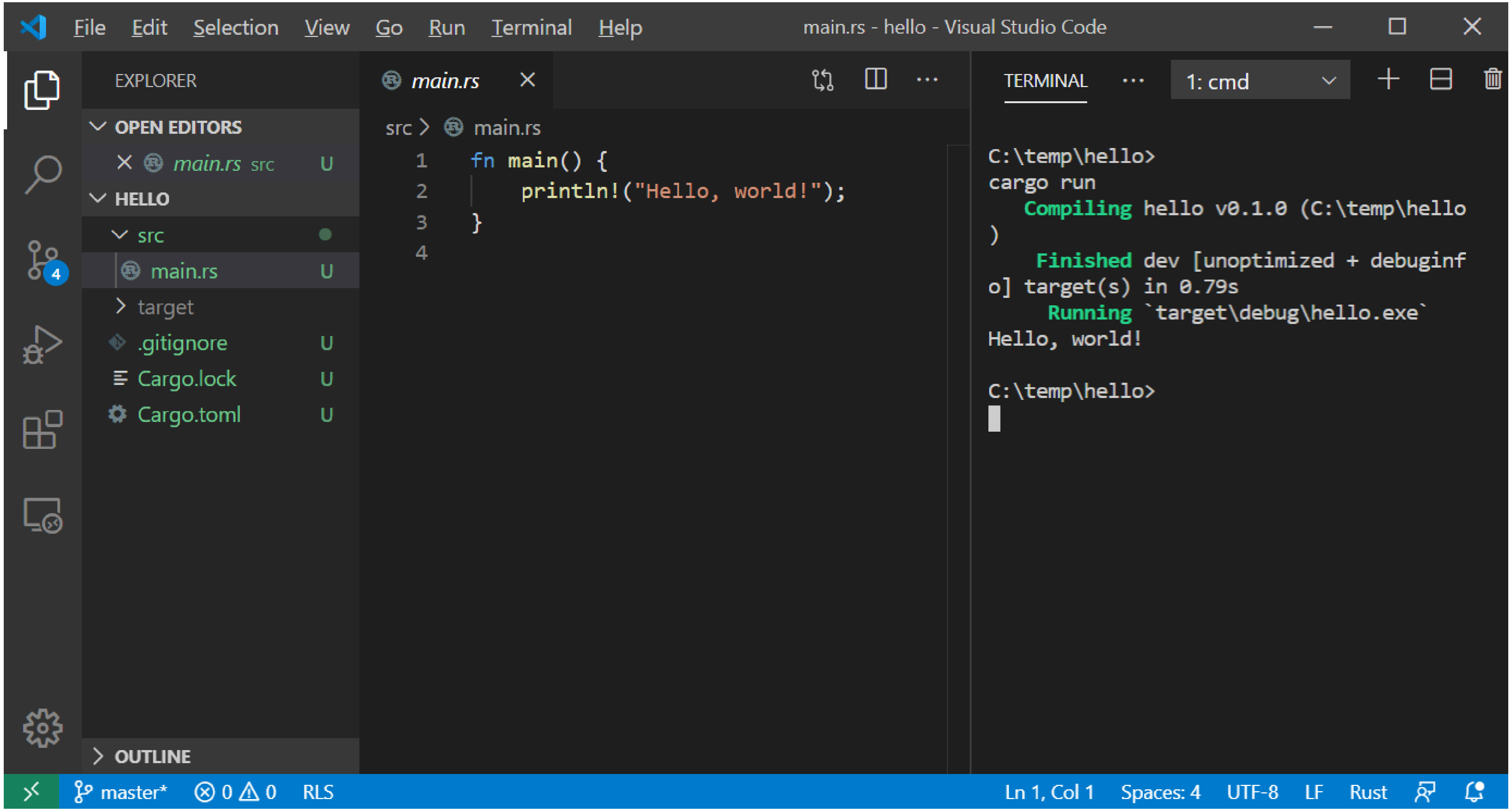
- This section assumes you have no experience with Rust.
- Getting started:
  - Install Rust - https://www.rust-lang.org/tools/install
  - This takes just a few minutes
  - Puts cargo, Rust's package manager, builder, executer on your path
  - Install Visual Studio Code - https://code.visualstudio.com/download
- 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

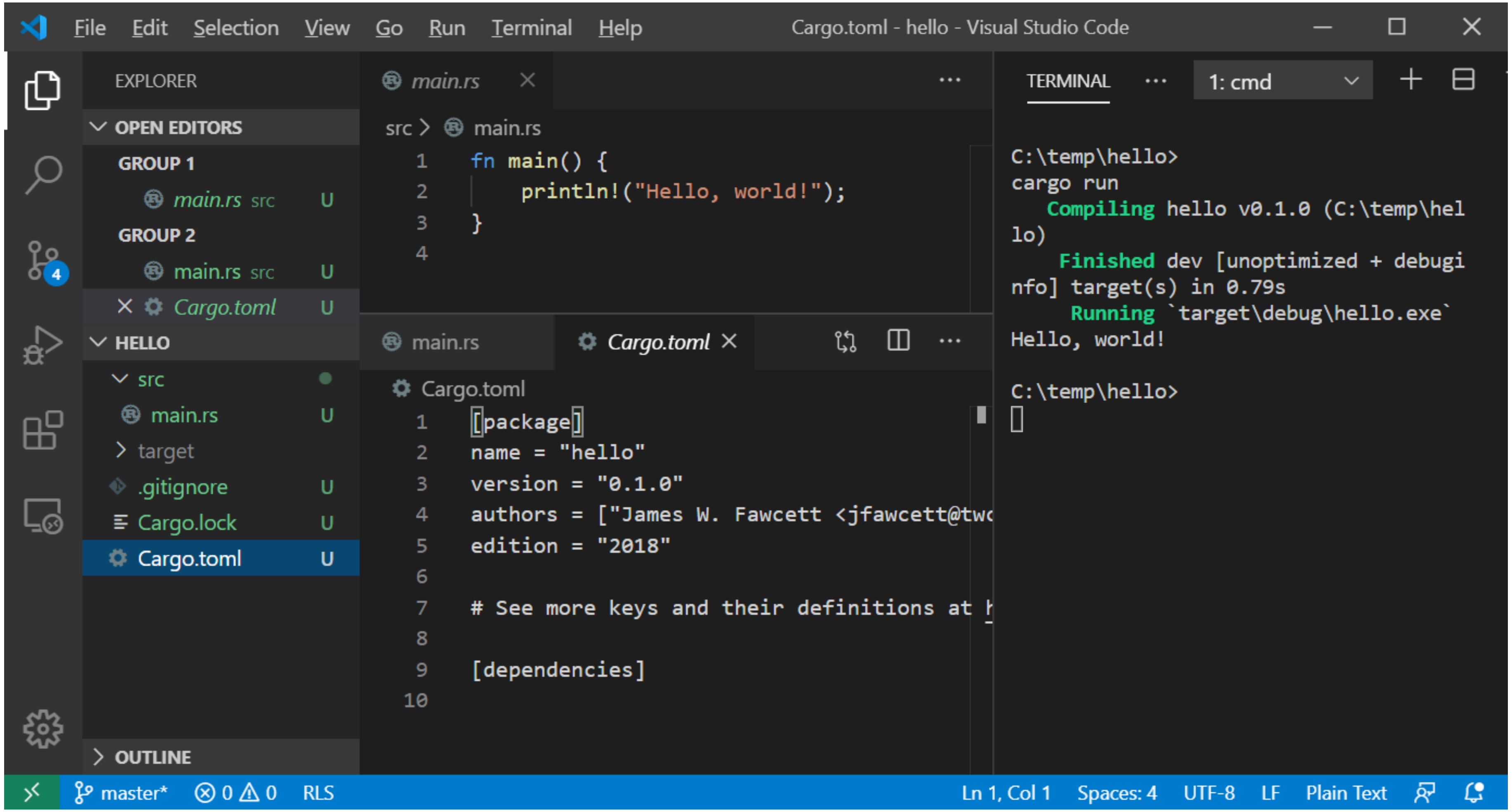

x64 Native Tools Command Prompt for VS 2019
c:\temp>
cargo new hello
Created binary (application) `hello` package
c:\temp>
cd hello
c:\temp\hello>
dir
Volume in drive C is OS
Volume Serial Number is 765A-DAD5
Directory of c:\temp\hello
03/29/2020 09:28 AM <DIR> .
03/29/2020 09:28 AM <DIR> ..
03/29/2020 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
OPEN EDITORS
main.rs src 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
master* 0 0 RLS
Ln 1, Col 1 Spaces: 4 UTF-8 LF Rust

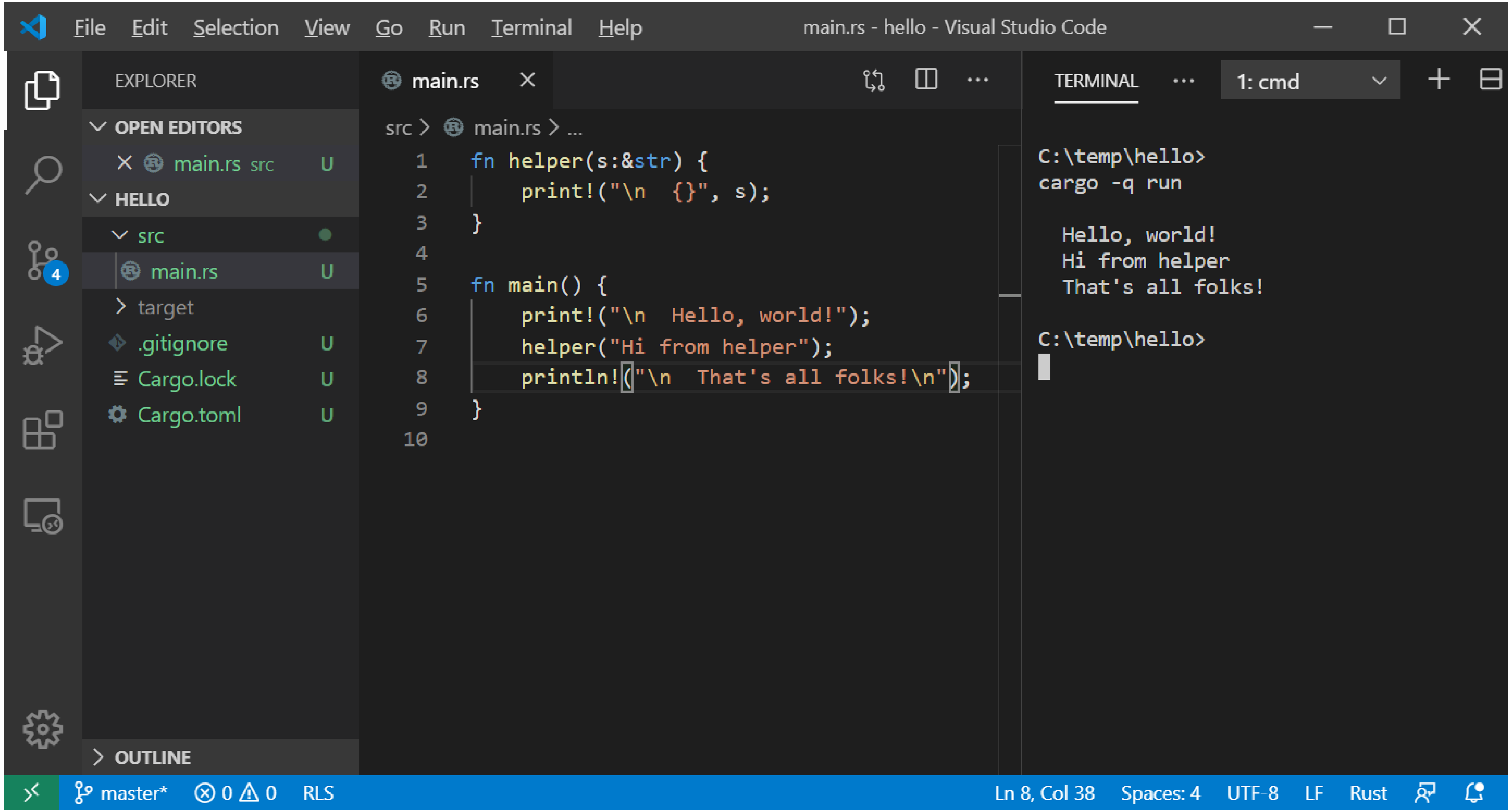
Building and Running with Cargo



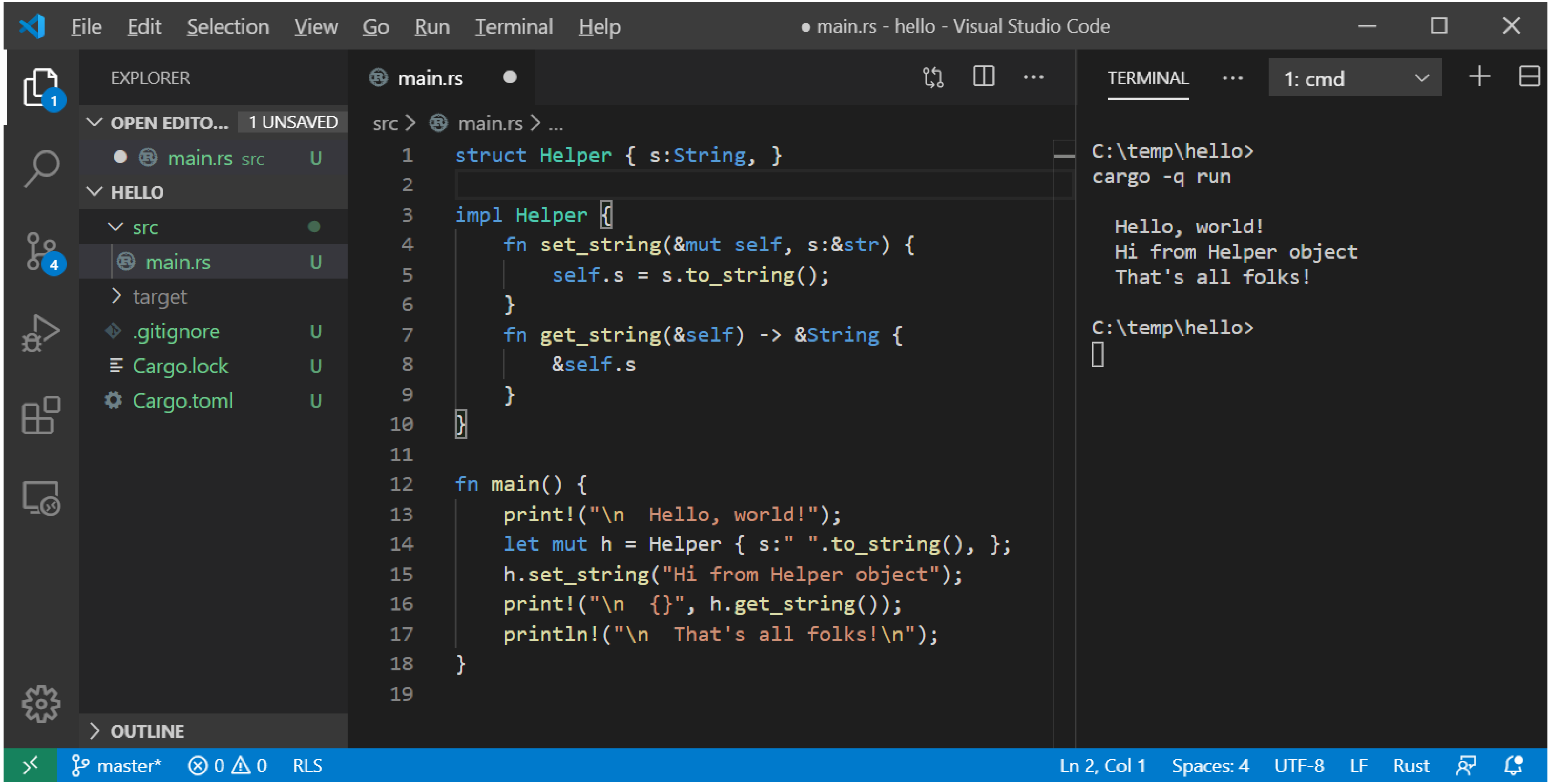
Cargo.toml – defines package



Add another function



Modify to use “object”



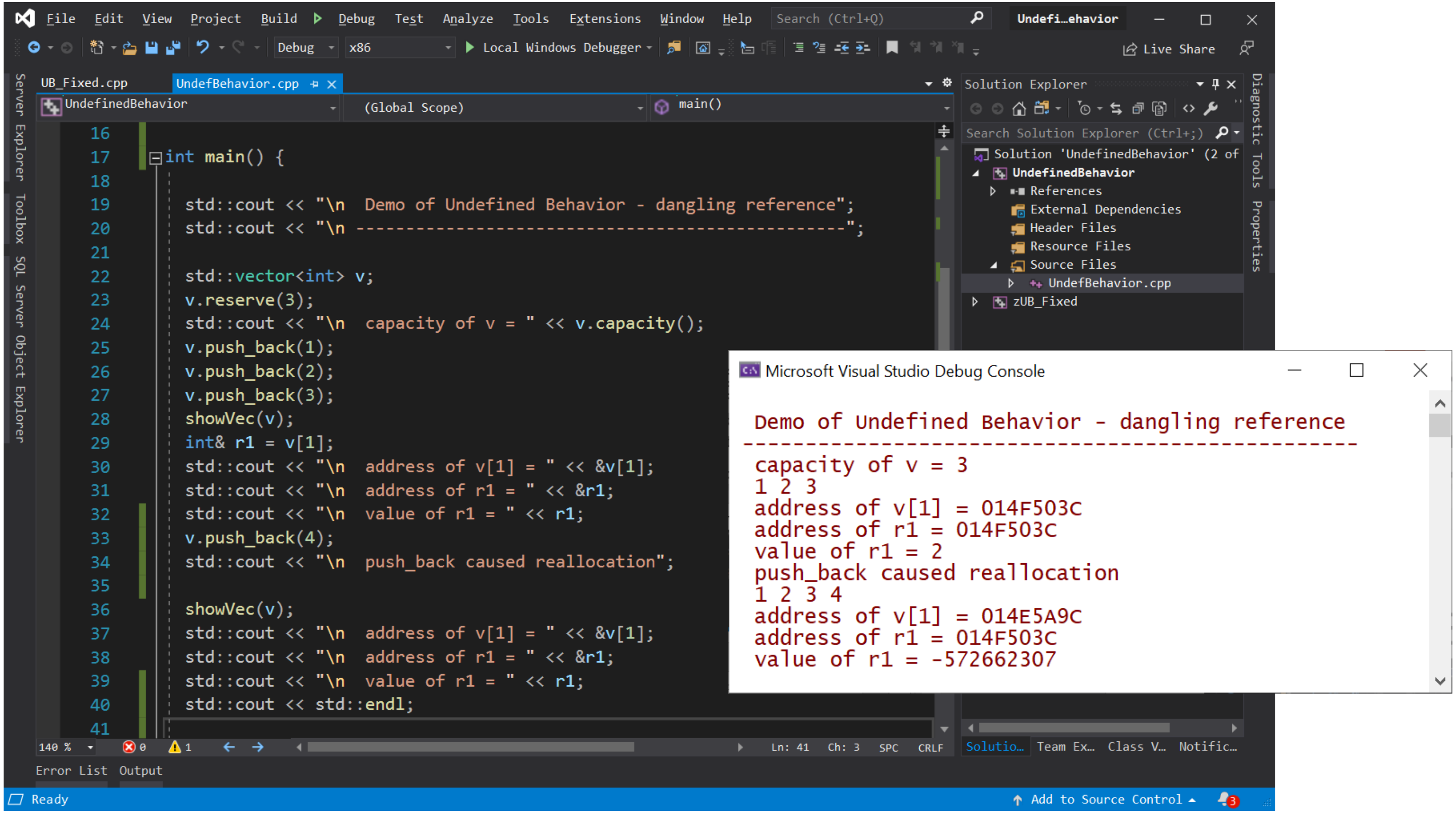
Why Rust?

- 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++

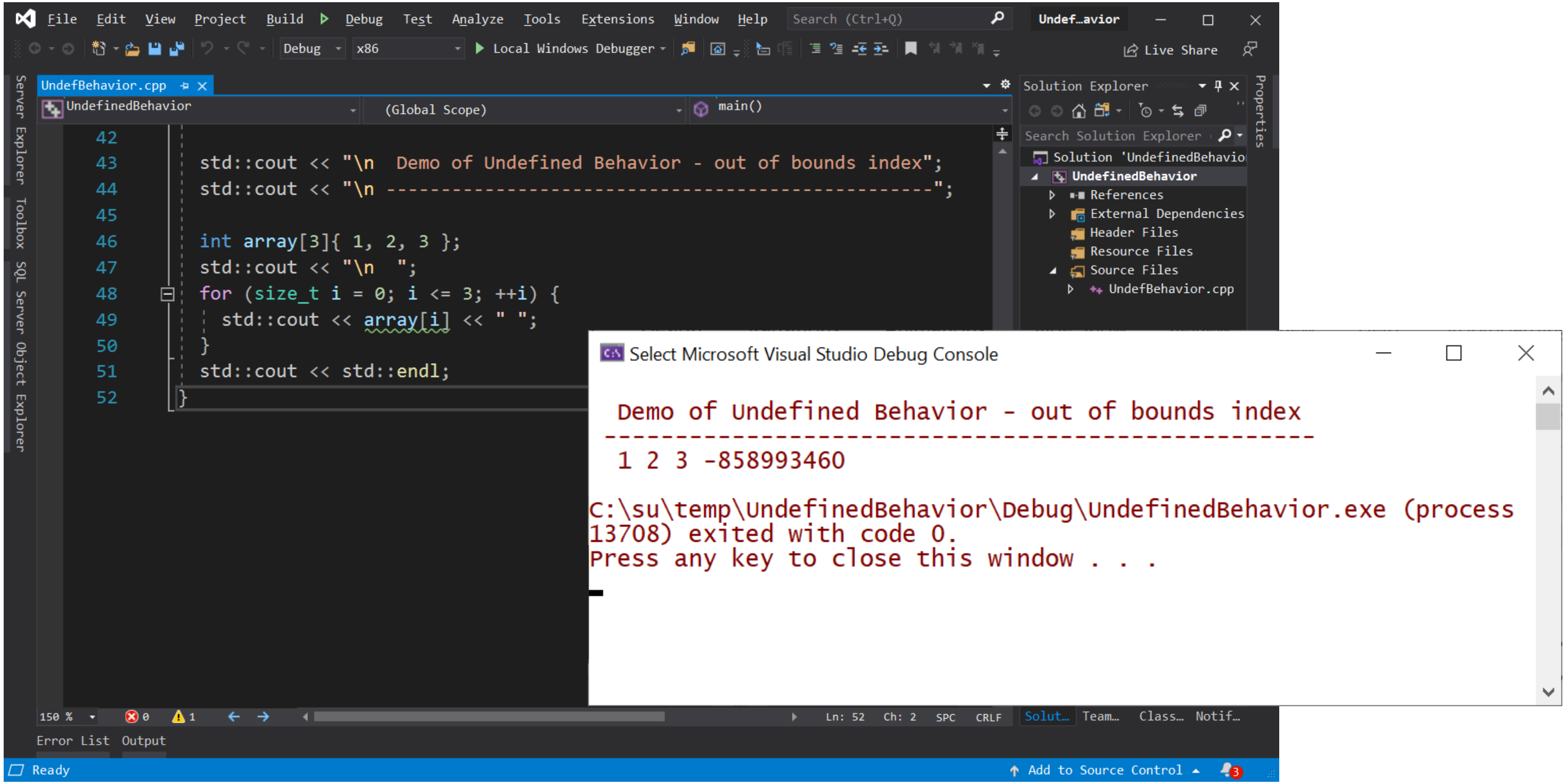
Type Safety

- 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

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) – programs fail to compile if they would have UB.
  - 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.
  - C++ code has to be well-crafted, using standard idioms, to avoid UB.
    ```

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 with its Ownership model:
  - Prevent mutation combined with aliasing
    - Ensure memory safety
  - Prevent mutation, aliasing, and lack of access ordering
    - Avoid data races

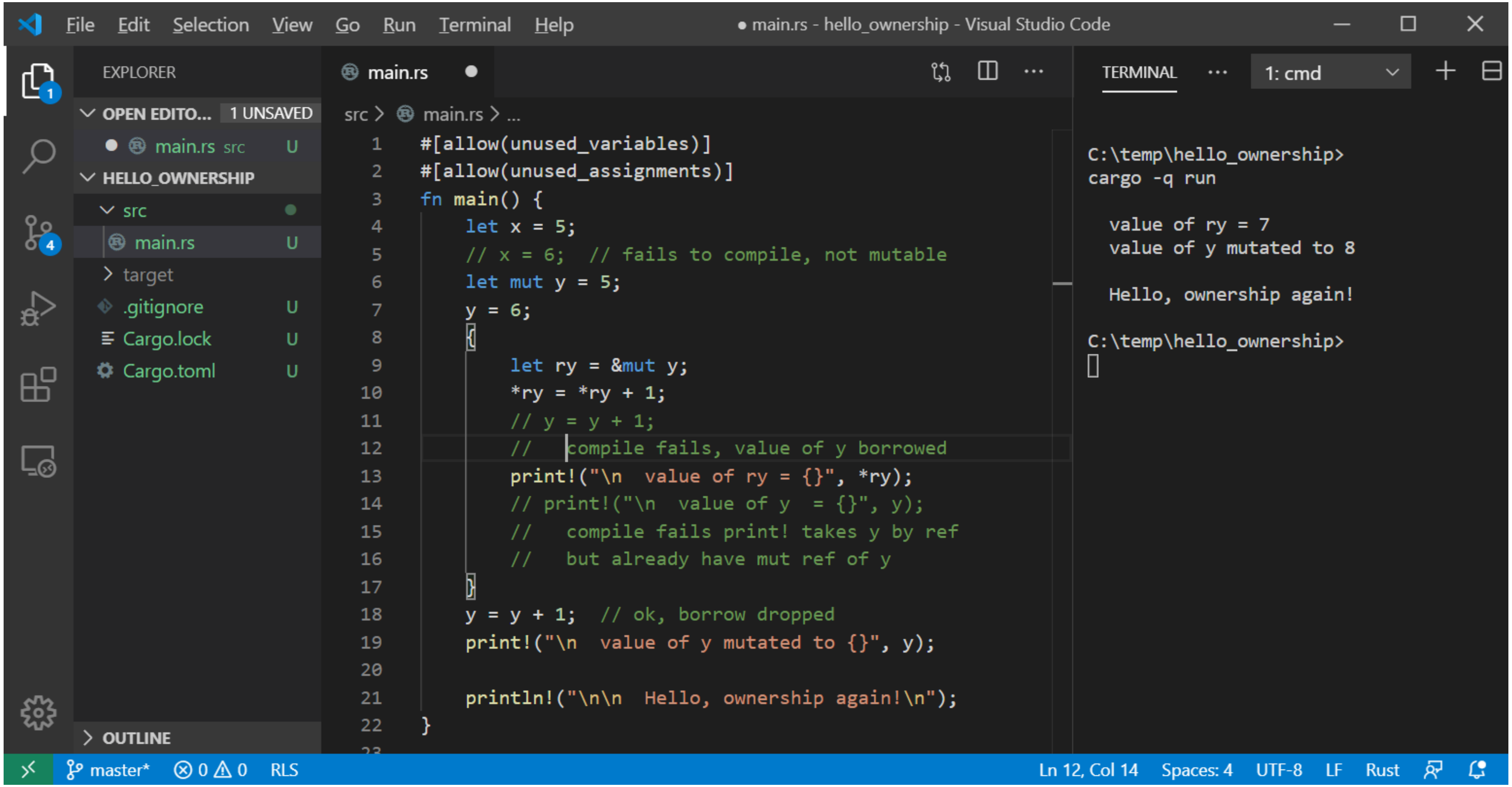
Rust Ownership

- 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;`

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
  - Variables 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

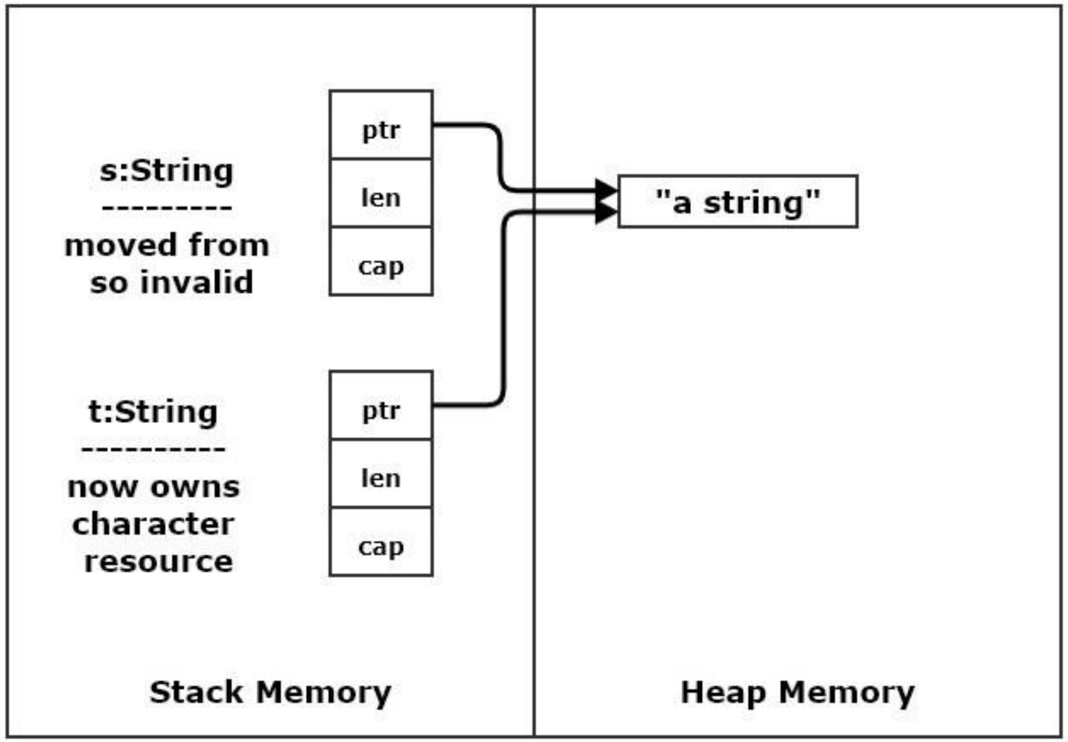


Copies, Moves

- 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**

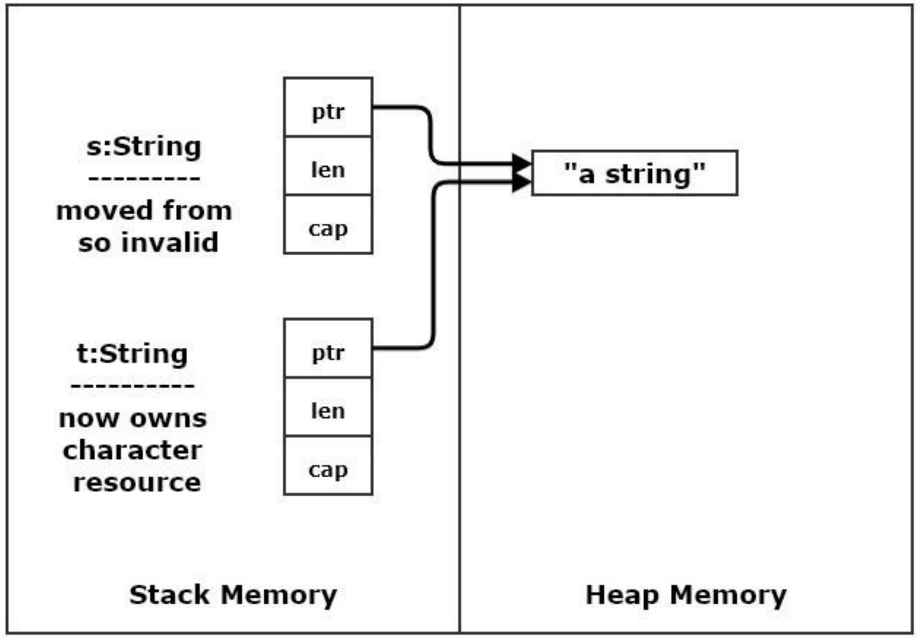
Rust Move versus Copy

- 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`



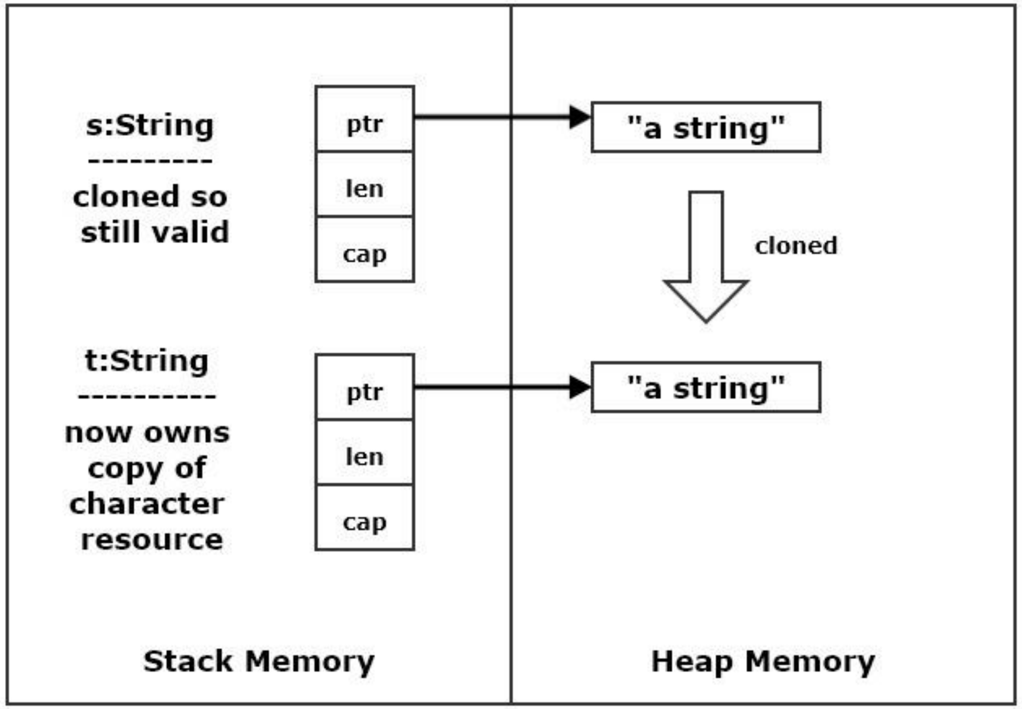
Move

- 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 will be identified as invalid by borrow-checker.
- This is fast. Characters are not copied, only the small control block is copied.



Rust Clone

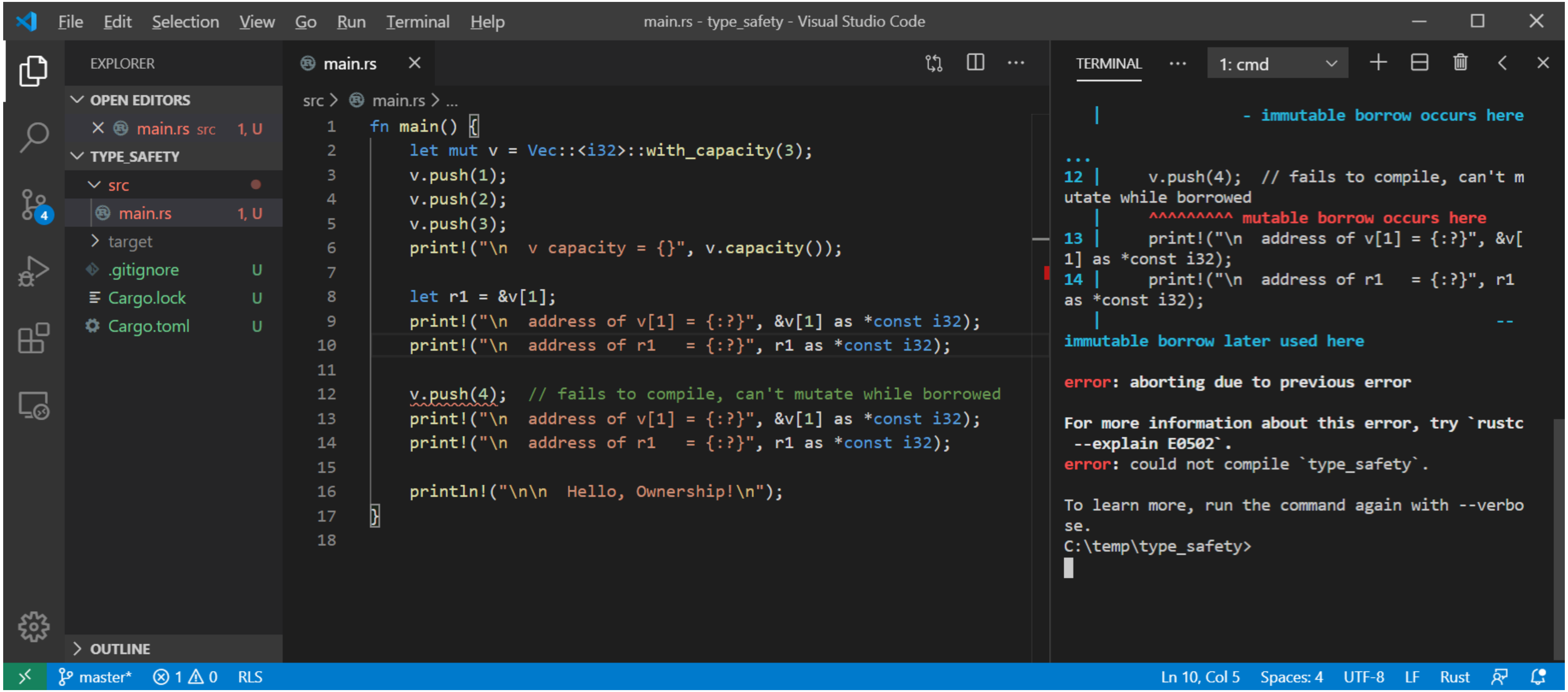
- Often a type satisfies clone trait (if not you can add that).
- This allows moves to be avoided by explicitly calling clone() to make a copy.
  - `let t = s.clone();`
    `// s still valid`
- Clone must always be called explicitly. Rust wants you to know when you invoke an expensive operation.



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 ownership   // 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 r = &mut v;  // r has exclusively borrowed v's ownership`
  - v cannot mutate until borrow ends

Rust won’t allow mutation with an active reference



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
- Mutable 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

- 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.

Traits

- Traits provide a contract – function specifications – that guarantee behavior.
  - Any type that implements the Clone trait can be cloned by calling `clone()`.
- Functions can accept arguments specified with either types or traits.
  - Specifying arguments with traits is more powerful – and more expensive.
  - Function will process any argument with a specified trait regardless of their type.
- If a type implements a trait, the trait methods become part of the public interface for that type, e.g., methods that can be called.
- You can even implement traits on existing types, much like C# extension methods.

Common Traits

- Derivable Traits
  - #[derive(Debug)]
  - Debug, Display, Copy, Clone
  - PartialEq, Eq, PartialOrd, Ord
  - Hash, Default
- Common Rust Traits
  - ToString, From, Into
  - AsRef, DeRef
  - Iterator
  - Read, Write
- https://stevedonovan.github.io/rustifications/2018/09/08/common-rust-traits.html
- https://stevedonovan.github.io/rust-gentle-intro/

Implementing Traits and Methods

- ```
  trait Size {
      fn size(&self) -> usize;
  }
  ```
- ```
  trait Show : Debug {
      fn show(&self) {
          print!("\n  {:?}", &self);
      }
  }
  ```
- ```
  #[derive(Debug, Copy, Clone)]
  pub struct Test { x:i32, y:f64, }
  ```
- ```
  impl Size for Test {
      fn size(&self) -> usize {
          std::mem::size_of::<Test>()
      }
  }
  ```
- ```
  impl Show for Test {}
  // using default impl
  ```
- ```
  impl Test {
      pub fn new() -> Self {
          Self { x:42, y:1.5, }
      }
      ...
  }
  ```

Rust Object Model – Static Binding

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

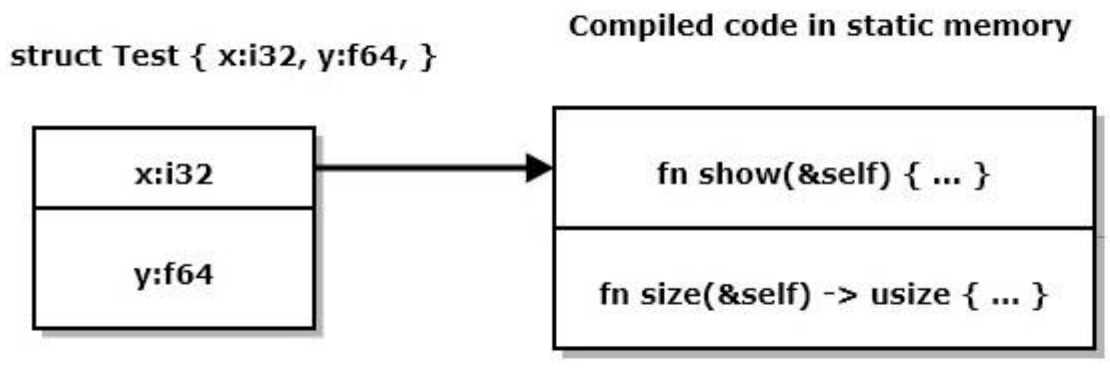
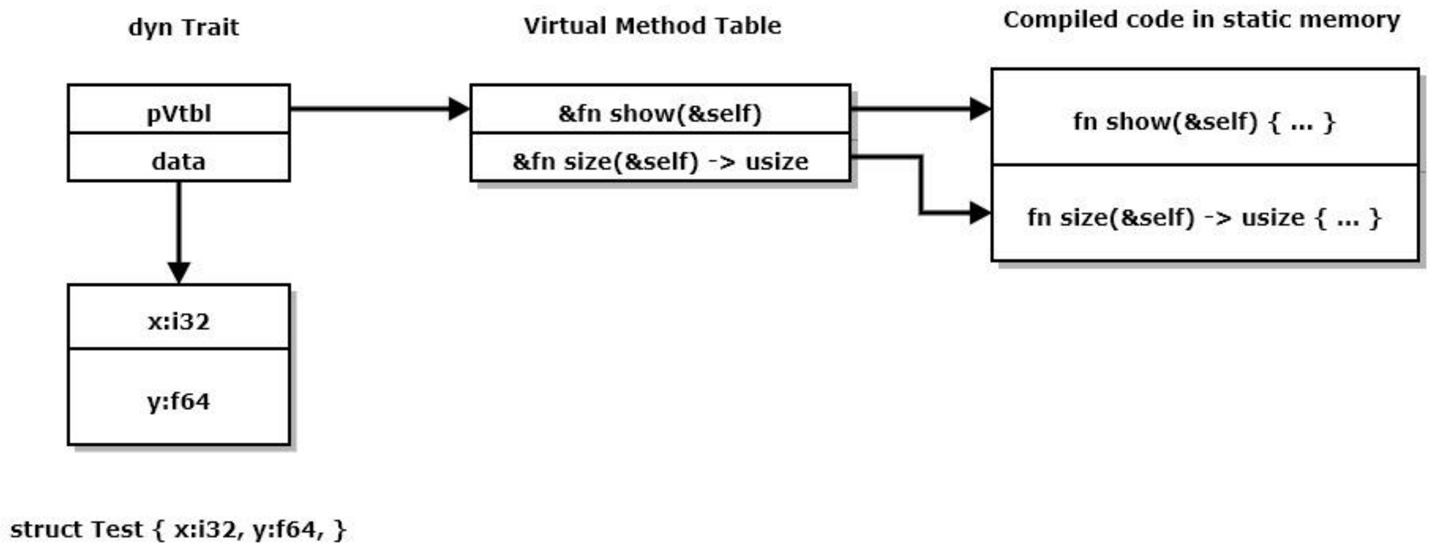


Table 1. Test Struct Memory Layout		
Component	Address	Size - bytes
Test Struct	8190584	16
y:f64	8190584	8
x:i32	8190592	4
padding	8190596	4

Rust Object Model – Dynamic Binding

```
• trait Show : Debug { … }
• trait Size { … }
• struct Test { x:i32, y:f64, }
• impl Show for Test { … }
• impl Size for Test { … }
• impl Test { … }
```



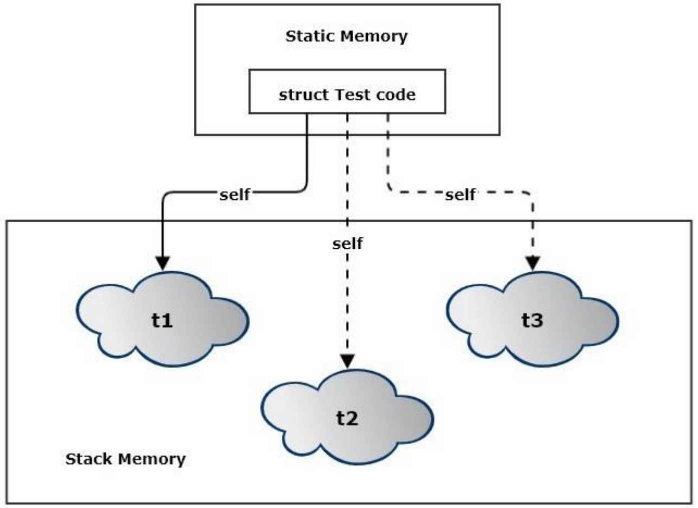

```
• fn size_is(o:&dyn Size) ->usize {
    o.size()
  }
```

size_is(…) doesn't know anything about Test. It does know Size::size

```
• let mut t = Test { x:42, y:1.5, };
  print!(
    "size of t = {:?}", size_is(&t)
  );
```

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 but not Copy.
- Test is a value type.




```
• trait Size {
      fn size(&self) -> usize;
  }

• trait Show : Debug {
      fn show(&self) {
          print!("\n  {:?}", &self);
      }
  }

• #[derive(Debug, Copy, Clone)]
  pub struct Test { x:i32, y:f64, }

• impl Size for Test {
      fn size(&self) -> usize {
          std::mem::size_of::<Test>()
      }
  }
```

```
• impl Show for Test {}
  // using default impl

• impl Test {
      pub fn new() -> Self {
          Self { x:42, y:1.5, }
      }
  }
```

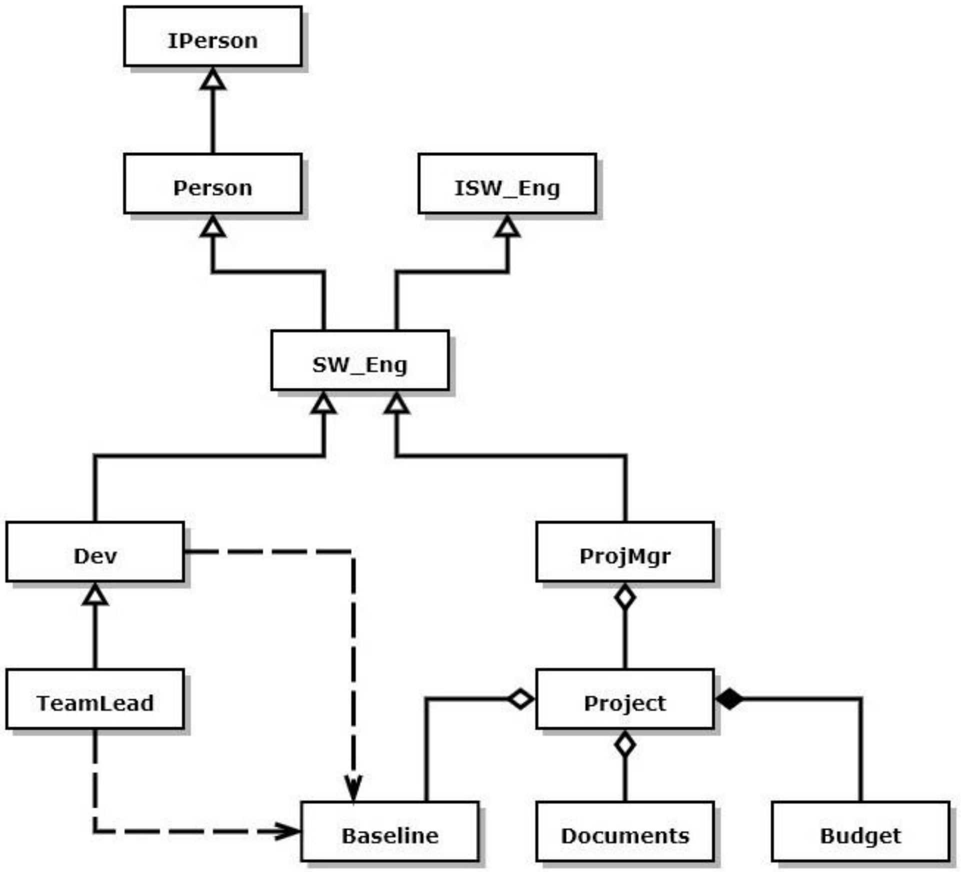
Comparison with C++

- 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

- Rust Generics define trait 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

- Traits provide a contract – function specifications – that guarantee behavior.
  - Any type that implements the Clone trait can be cloned by calling `clone()`.
- Functions can accept arguments specified with either types or traits.
  - Specifying arguments with traits is more powerful – and more expensive.
  - Function will process any argument with a specified trait regardless of their type.
- If a type implements a trait, the trait methods become part of the public interface for that type, e.g., methods that can be called.
- You can even implement traits on existing types, much like C# extension methods.

Traits – Note: these traits don't use T, but their implementation does

- ```
  trait Show : Debug {
    fn show(&self) {
      print!("\n  {:?}", &self);
    }
  }
  ```
- ```
  trait Size {
    fn size(&self) -> usize;
  }
  ```
- ```
  fn size_is(o:&dyn Size) ->usize {
    o.size()
  }
  ```

size_is(o:&dyn Size) accepts both ordinary and generic arguments

- ```
  #[derive(Debug, Copy, Clone)]
  pub struct Point<T>{ // public type
    x:T, y:T, z:T,     // private data
  }
  ```

- ```
  impl<T> Show for Point<T>
    where T:Debug {}  // using default impl
  ```
- ```
  impl<T> Size for Point<T> {
                           // must provide impl
    fn size(&self) -> usize {
        std::mem::size_of::<Point<T>>()
    }
  }
  ```
- ```
  let mut t =
    Point { x:0.0, y:1.0, z:0.5, };
  ```
- ```
  t.show();
  ```
- ```
  print!(
    "\n  size of t = {:?}", size_is(&t)
  );
  ```
  ```

Generics Summary

- Generics help us build flexible code:
  - Create collections that can hold many different types, but we need only one design.
- Generics with traits provide even more help
  - Define functions and methods that accept arguments that satisfy a trait specification.
  - Much more flexible than defining functions that take specific typed arguments.
  - Allows us to specify that only some categories of types should be accepted, e.g., move-able, or clone-able, or display-able.

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 an /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.

Rust Pain Points

- Ownership
  - Conceptually simple, must handle details to compile ☹
  - Compiler error messages are very good ☺
- Error Handling
  - All cases of all errors have to be handled to compile
  - Many examples use naïve handling, e.g., panic.
    - Not a good idea for anything other than demo code.
- Strings
  - String, &str, OsString, &OsStr, PathBuf, &Path
  - No indexing, can use iterator
- Explicit conversions
  - Virtually no implicit conversions

Rust Gain Points

- Ownership
  - No undefined behavior or data races by construction
- Error Handling
  - No surprises at run-time.
  - Get coherent error messages instead of aborts.
- Strings
  - Utf-8 strings can represent characters from many languages and math symbols
- Explicit conversions
  - No surprises from unexpected conversions
- Suitable for safety critical applications, e.g., vehicle control, medical and financial applications.
  - Need all of the above
  - Eliminates many of the vectors for malware threats

Epilog

https://jimfawcett.github.io/RustStory_Prologue.html

Conclusions

- If you understand the 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, String and Str, 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/RustStoryRepo.html

- These slides are available here:
  https://jimfawcett.github.io/Resources/RustTour.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. Many repos 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.