Lesson 2.1: Cargo

Cargo is Rust's comprehensive general-purpose tool, serving as the package manager, build system (replacing make files), test runner, and documentation generator. New projects are created with cargo new, which generates a Cargo.toml configuration file and source structure, and can be built and run with cargo run.

Key Takeaways

- Cargo makes systems programming easier by combining many essential development tools into one.
- Cargo.toml is the authoritative config file, using the TOML format, and defines crucial package details like name and semantic versioning.
- Cargo defaults to compiling in debug mode; adding --release provides compiler optimizations, significantly increasing code speed and reducing binary size.
- Dependencies can be conveniently added using the cargo add command, which automatically updates the Cargo.toml file.

Key Concepts

- **Cargo:** The primary tool for managing Rust projects, handling everything from dependency installation to running tests.
- **Cargo.toml:** The package configuration file that dictates package metadata, versioning, and dependencies.
- **Semantic Versioning (SemVer):** A version numbering standard (Major.Minor.Patch) used by Rust to indicate the scope of changes (breaking, features, or bugs).
- **Build Artifacts (target)**: The directory where Cargo outputs compiled binaries, which should be ignored by version control software.

Lesson 2.2: Variables

Variables are declared using the let keyword, and although Rust is strongly typed, type annotations are often omitted when the compiler can infer the type. Variables are immutable by default in Rust, a deliberate design choice that enhances safety, concurrency, and speed.

Key Takeaways

- To allow a variable's value to change, the mut keyword must be explicitly used (e.g., let mut bunnies).
- The Rust compiler is helpful, providing contextual error messages and suggestions on how to fix errors, such as suggesting the addition of mut.
- The let statement supports **destructuring**, allowing multiple variables to be initialized simultaneously using a pattern like a tuple.
- **Constants** are declared using const (not let), require a type annotation, use SCREAMING_SNAKE_CASE convention, and must be set to a value that can be determined at compile time.

Key Concepts

- **Immutability by Default:** The policy that prevents data changes unless mut is used, which improves safety and enables better compiler optimizations for speed and concurrency.
- **Type Inference:** The Rust compiler's ability to determine the appropriate type of a variable without explicit type annotation.
- **Constant (const):** An immutable, fast variable type whose value is inlined at compile time and is often placed outside of functions at module scope.

Lesson 2.3: Scope

Variables have a scope that begins at creation and lasts until the end of the surrounding block (statements inside curly braces), and values are immediately dropped when they go out of scope. The compiler enforces scope rules at compile time, preventing issues like accessing a variable defined in a nested block from outside.

Key Takeaways

- Since there is no garbage collector, values are promptly dropped when their scope terminates.
- Shadowing allows a variable to be redefined with the same name in a nested block or even the same scope.
- Shadowing enables redefining a variable with different mutability (e.g., mutable to immutable) or even changing its underlying type.
- When an inner variable shadows an outer variable, the inner variable is dropped when its block ends, restoring access to the outer variable.

Key Concepts

- **Scope:** The code region where a variable is valid, extending from declaration to the end of the current block**.**
- **Block:** A section of code delimited by curly braces that defines a scope boundary.
- **Shadowing:** Redefining a variable using let in a way that hides the previous variable with the same name, even allowing a change in type.

Lesson 2.4: Memory Safety

Rust guarantees memory safety at compile time by strictly requiring that all variables be initialized with a value before they are used. This prevents the use of uninitialized variables, which, in languages like C, can lead to unpredictable results known as undefined behavior.

Key Takeaways

- Code attempting to use a declared but uninitialized variable will not compile, resulting in an error.
- If a variable's initialization is conditional, the compiler must guarantee that the variable is initialized in all possible branches before it is used later.

Key Concepts

- **Initialization Requirement:** A core component of Rust's memory safety guarantee, ensuring variables always hold a defined value when accessed.
- **Undefined Behavior:** A dangerous state in other languages (like C) where using an uninitialized variable leads to arbitrary, platform-dependent results.

Lesson 2.5: Exercise A - Variables

Exercise A guided the student through creating a new project using cargo new, modifying package metadata (version) in Cargo.toml, and implementing variable declarations and simple arithmetic. The process highlighted the practical necessity of using let mut for mutable variables and correctly placing constants at module scope.

Key Takeaways

- The project configuration (Cargo.toml) is where the version number is officially changed.
- If variable assignment is attempted without let mut, a compilation error results because the variable is immutable by default.
- Common syntax errors include forgetting the semicolon at the end of a statement or forgetting the let keyword.
- For organization, constants should typically be defined outside the main function at module scope.

Key Concepts

- **cargo new:** The foundational command used to set up a new Rust project structure, including the Cargo.toml and src/main.rs files.
- **Module Scope Constant Placement**: The recommended location for constants, allowing them to be shared across the entire module.

Lesson 2.6: Functions

Functions are defined using the fn keyword and use snake case naming conventions, with parameters defined as name: type and the return type specified after an arrow (->). Rust allows functions to be called regardless of their location in the file, enabling chronological source code arrangement.

Key Takeaways

- The preferred, idiomatic way to return a value is using a tail expression, which is the final expression in a block written without a semicolon.
- Functions do not support variable numbers of arguments or different argument types for the same parameter.
- Macros (like println!) look like function calls but are identified by an exclamation mark (!) and offer more flexible argument handling than standard functions.

Key Concepts

- **Function Definition (fn):** The keyword used to declare a function, followed by its name, parameters, and optional return type.
- **Tail Expression:** An expression at the end of a block (including a function body) that, when lacking a semicolon, is automatically returned as the block's value.
- **Macro (!):** A facility used to generate code, often used for functions like println! that need to handle variable arguments.

Lesson 2.7: Exercise B Functions

Exercise B reinforced the rules of variable scope, demonstrated how to fix type mismatches using variable shadowing, and provided practice in writing a simple function with a tail expression return. The exercise also confirmed the requirement that variables must be guaranteed to be initialized before use, even across conditional branches.

Key Takeaways

- Associated functions are defined in impl blocks and called using StructName::function().
- Variables defined in a nested scope are inaccessible outside of that block, requiring the use of the variable to be hoisted or defined in a common parent scope.
- When conditional logic initializes a variable, both the if and else branches must ensure the variable receives a value for compilation to succeed.

Key Concepts

- **Scoping Resolution:** Ensuring that any use of a variable occurs within the curly braces defining its valid scope.
- **Function Parameter Definition:** Specifying parameters within a function signature using (name: type, ...) .

Lesson 2.8: Module System

The module system is used to organize library code, where lib.rs is the library root and items within it are private by default, requiring the pub keyword for external access. The use statement is crucial for bringing items into scope, making them accessible without specifying their full, absolute path (which starts with the package name).

Key Takeaways

- For functions, the idiomatic style is to bring the containing module into scope (e.g., use hello::english) and call the function through that module (e.g., english::greet()).
- Absolute paths within a library start with the keyword crate instead of the library's name.
- Modules can be defined inline using curly braces or externally by using a semicolon (e.g., mod spanish;), which tells Rust to look for a corresponding .rs file.
- Top-level modules reside in the source directory, while submodules are placed in directories named after their parent module.

Key Concepts

- **Module (mod):** A unit of code organization that creates its own scope; can be public (pub) or private (default).
- **use Statement:** Mechanism for bringing items from a path (e.g., package::module::item) into the current scope for simplified calling.
- **pub Keyword:** Used to set an item's visibility, making it accessible from modules outside of its current scope or package.
- **Crate.io:** Rust's package registry, which serves as the starting point for finding external packages/libraries to use.

Lesson 2.9: Exercise C - Module System

Exercise C involved refactoring constants and functions from main.rs into a library file (lib.rs) and a submodule (sound.rs), ensuring the original output remained consistent. It emphasized that all moved items must be explicitly marked pub to be accessible from the binary (main.rs) and demonstrated importing multiple items concisely using braces in a single use statement.

Key Takeaways

- Constants are made available externally by defining them in lib.rs and preceding them with pub.
- The name of the library/package is taken from the name field in Cargo.toml.
- Multiple items from the same path can be imported efficiently using syntax like use animal::{first, second, third}.
- Submodules (like sound) must also be public and functions within them must be public to be accessed from outside.

Key Concepts

- **Library Root (lib.rs):** The special file that serves as the root of the project's reusable library code.
- **Concise Importing:** Using curly braces and commas within a use statement to group imports from the same module path.
- **Idiomatic Function Call:** Calling a function through its containing module (e.g., sound::dog()) is conventional style, especially when the function is public.