Pragmatics of Rust and C++: The implementation of a window manager

Max van Deurzen
Technische Universität München
Munich, Germany
m.deurzen@tum.de

***Abstract*—In comparing and discussing programming languages (and natural languages, for that matter), not only *syntax* and *semantics* are of importance. *Pragmatics* is the third general area of language description, which deals with the practical aspects of how language constructs and features allow its users to achieve various objectives[1]. In this paper, we will look at the pragmatics of both Rust and C++. Specifically, we will be comparing the languages in their ability to be used as a tool to write medium to large *system programs*. As a case study, we will be discussing two implementations of a complete ICCCM and EWMH compliant top-level reparenting, tiling window manager, built on top of the X Window System: one written in Rust, and the other in C++.**

***Keywords*—Pragmatics, Rust, C++, Window Manager**

I. Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

II. External Dependency Management

As a programming language's ability to aid the programmer in managing external dependencies—by, for instance, providing various tools that come installed with its compiler or development environment—is generally not incorporated into that language's syntax or, by extension, its semantics, it is traditionally not considered an aspect of that language's feature set per se. Notwithstanding, it is more and more becoming an appreciated addition to the *ecosystem* around a language, especially so for compiled languages. In fact, many would consider automated external dependency management to be a must for any modern programming language. As it directly affects both the portability of code, as well as the (ease of) managing different versions of a dependency, a language's ability to unburden the programmer from the manual management of external dependencies can greatly improve the maintainability of a project, and can therefore indeed be viewed as a feature of the language in light of its *pragmatics*. In this section, we will be discussing the practicalities of working with external code in both Rust and C++. We will do this by means of a comparison between two window manager implementations, one written in Rust, and the other written in C++, which we will henceforth be referring to as wmRS and wmCPP, respectively.

Both window managers are built on top of the X Window System, which means they rely on an external library to interface with the X server. Although wmRS and wmCPP each use a different library to achieve this (XCB over `libxcb` and Xlib over `libX11`, respectively), the concept is the same, as they both require the importing of an external body of code.

C++ does not come with an external dependency manager. That is to say, the official ISO standard C++ specification[2] does not outline the functionality or otherwise existence of any package manager. As a result, developers have no other choice than to resort to third-party tools to manage dependencies, to reinvent the wheel themselves and hack together configure and build scripts that take external dependencies into account, or to disregard package management altogether and let other developers and end users resolve dependencies on their own. In any case, C++ dependency management is complex, and the lack of a standardized tool inhibits the adoption of C++ projects, as well as their portability to other platforms and development environments.

Since the only external dependency wmCPP is reliant on is `libX11`, which is fairly ubiquitous and readily available through distributions' own package managers, it merely links with the X11 development libraries without the use of any form of own dependency management. Specifically, building the project is done using the following Make script (large parts altered or redacted for clarity).

```
CC = g++
CXXFLAGS = -std=c++17 -march=native -O3
LDFLAGS = `pkg-config --libs x11` -flto

obj/%.o: src/%.cc
  ${CC} ${CXXFLAGS} -MMD -c $< -o $@
release: obj/%.o
  ${CC} $< ${LDFLAGS} -o bin/wmCPP
```

Assuring the availability of an external body of code is not the only aspect of external dependecy management that affects the pragmatics of C++. Writing a library that is to be imported as an external dependency and including code from an external dependency both require special care to be taken to make sure no symbol collisions occur. In header files, this is usually done by making use of so-called *header guards*, a set of preprocessor directives that render including a header an idempotent operation, preventing double inclusion errors. If incorrect double inclusion does unexpectedly occur, for instance due to collisions in the header guards themselves, vague—as header guards are handled by the preprocessor, not the compiler—and hard-to-trace compiler errors arise, subjecting the programmer to unnecessary mental strain, and wasting time. Although not part of the official standard[2], many C++ compilers support the `#pragma once` preprocessor directive, providing the same functionality as header guards, but preventing the occurrence of double inclusion errors.

Another source of issues is the order of include directives in C++. The ordering of included files relative to one another can significantly influence the outcome of the compilation stage, possibly causing errors, one of which being the introduction of circular dependencies while there actually aren't any there (e.g. a forward declaration that was included only after the circular dependency was already introduced).

The Rust programming language has its own package manager, called Cargo. Cargo can automatically download a project's external dependencies (and, transatively, *their* dependencies), compile them, and install them locally, such that they can be used during the linking stage of the build process[3]. To make proper use of Cargo within a Rust project, that project must be turned into a so-called *package*. Rust packages are simply a collection of source files along with a manifest file (named `Cargo.toml`) in the project's root directory. This manifest file describes the package's meta-information (such as its name and version), and a set of *target crates*[3]. A crate is the source code or compiled artifact of either a library or executable program, or possibly a compressed package that is grabbed from a registry (a service that provides a collection of downloadable crates)[3]. A package's manifest file describes each of its target crates by specifying their type (binary executable or library), their metadata, and how to build them[3].

The following manifest file describes the package that represents our Rust window manager, WMRS (parts redacted for clarity).

```
[package]
name = "wmRS"
version = "0.1.0"
authors = ["deurzen <m.deurzen@tum.de>"]
edition = "2018"
license = "BSD3"
documentation = "https://docs.rs/wmRS"
readme = "README.md"
default-run = "wmRS"
description = """
An ICCCM & EWMH compliant X11 reparenting,
tiling window manager, written in Rust
"""
[profile.release]
lto = true
[lib]
name = "winsys"
path = "src/winsys/mod.rs"
[[bin]]
name = "wmRS"
path = "src/core/main.rs"
[[bin]]
name = "wmRSbar"
path = "src/bar/main.rs"
[[bin]]
name = "wmRSclient"
path = "src/client/main.rs"
[dependencies]
x11rb = {
  version = "0.8.0",
  features = [
    "cursor",
    "xinerama",
    "randr",
    "res"
  ]
}
```

Our package consists of a single library crate (`winsys`), along with three binary executable crates (`wmRS`, `wmRSbar`, and `wmRSclient`). The library is an abstraction above and wrapper around the interface with the underlying windowing system. It defines a single Rust *trait* that represents the connection between the window manager and some windowing system. A trait is a *zero-overhead*[4] collection of methods that is defined for some (at define-time) unknown type `Self` (which at implementation-time becomes the implementing type), most often used with the intention to implement shared behavior[5,6]. A concept from other languages that most closely resembles traits are *interfaces*, although there are differences still[5,7]. A small snippet from WMRS's `winsys` library `Connection` trait looks as follows.

```
pub trait Connection {
  fn step(&self) -> Option<Event>;
  fn close_window(&self, window: Window) -> bool;
  fn move_window(&self, window: Window, pos: Pos);
  fn resize_window(&self, window: Window, dim: Dim);
  fn focus_window(&self, window: Window);
  // ...
}
```

Here, we've supplied a set of function prototypes that eventually become the methods that every type that implements this trait will be required to define. Although we haven't done so here, traits themselves are also allowed to define their functions with some default behavior, that may or may not be overwritten by its implementors[5,7]. Our `Connection` trait represents shared behavior that every windowing system wrapper is required to contain. This allows for the decoupling of the implementation of the higher-level window management functionality from that of the interface with the windowing system (e.g. the explicit drawing of primitives to the screen, or the management and manipulation of windowing system specific window representations), and additionally allows for the seamless transition from one windowing system to another, as multiple windowing systems can be targeted by implementing the trait, effectively creating a new wrapper around an external library that directly interfaces that windowing system. Currently, WMRS only implements the interface with the X Window System, but interfacing with the newer and more modern Wayland is as easy as implementing a new connection to it. WMRS's implementation for this trait, targeting the X Window System, is partly given as follows.

```rust
use x11rb::connection;

pub struct XConnection
  <'conn, Conn: connection::Connection>
{
  conn: &'conn Conn,
  // ...
}

impl<'conn, Conn: connection::Connection>
  Connection for XConnection<'conn, Conn>
{
  #[inline]
  fn step(&self) -> Option<Event> {
    // ...
  }
  // ...
}
```

The `XConnection` structure introduces two generic type arguments, one being a lifetype parameter (`'conn`), and the other being the type of the struct's `conn` field (`Conn`), which *requires* the `connection::Connection` trait to be implemented. This `connection::Connection` trait is imported from the external package `x11rb`, which we supplied as a project dependency in our manifest file. The `x11rb` package serves as a Rust API to the X Window System, essentially providing us with Rust bindings to interact with the X server. Finally, the `XConnection` struct implements our previously defined `Connection` trait, allowing it to be used by our window manager by means of composition. Given below is the structure that represents and encapsulates the core window manager logic. It *contains* a reference to *some* type that implements our `Connection` trait (i.e. a wrapper around *some* windowing system).

```rust
pub struct Model<'model> {
  conn: &'model mut dyn Connection,
  // ...
}
```

Since we're using polymorphism here to abstract away from the actual, concrete implementation of the connection type, Rust needs to be able to determine at runtime which specific version it should actually run (known as *dispatch*)[4,5]. It can do so either *statically* or *dynamically*. Static dispatch in Rust makes use of *monomorphization* at compile time to convert generic code into "specific" code: one version for each concrete type that is used as a generics argument[4,5]. That means that at runtime, no (time) overhead is incurred when running generic code, as the specific versions of the code to run are baked into the binary[4,5]. An example of static dispatch in WMRS is the following `Cycle` struct, which allows for cycling forward and backward through a collection of (generic) items.

```rust
pub struct Cycle<T>
where
  T: Identify + Debug,
{
  index: Cell<Index>,
  elements: VecDeque<T>,
  indices: HashMap<Ident, Index, BuildIdHasher>,
  unwindable: bool,
  stack: RefCell<HistoryStack>,
}

impl<T> Cycle<T>
where
    T: Identify + Debug,
{
  // ...
}
```

This struct is used to cycle through both the workspace structures managed by the window manager, as well as the clients within a workspace. That means two specific versions of this struct are constructed at compile-time: one for `Cycle<Workspace>`, and one for `Cycle<Client>`, where `Workspace` and `Client` in turn both implement the `Identify` and `Debug` traits (as per the constraint on `T`).

Dynamic dispatch defers resolving generic code until it is required at runtime, making use of so-called *trait objects*[5,8]. A trait object is an opaque value of a type that implements some set of traits[9]. It's opaque, as one cannot know which concrete type is behind a trait object's pointer up front[9]. Whereas the size of each monomorphized type is always known, trait objects are therefore dynamically sized[9]. To resolve a call to one of such an opaque type's methods at runtime, *virtual method tables* (*vtables*) are used[9]. Each instance of a pointer to a trait object consists of a pointer to an instance of some type `T` that implements the set of traits, as well as a pointer to a vtable that contains a function pointer to `T`'s implementation of each method of the implemented set of traits (and their supertraits)[9]. Our `Model` struct's `conn` field is a pointer to such a trait object. Behind the pointer, we can have any implementation of the `Connection` trait. Which of the trait's implementors' methods are called, is determined at runtime.

Looking back at WMRS's manifest file, the three binary executable crates in its package respectively represent the window manager itself, a client program to communicate with the window manager (to control various window management activities, such as closing the currently focused window, from scripts or the command line), and a status bar that displays information about the state of the window manager, such as the currently activated workspace. Each make use of the `winsys` library to communicate with the underlying windowing system.

Our C++ window manager implementation, WMCPP, attempts to achieve the same flexibility by making use of *abstract classes*. Abstract classes define abstract types that cannot themselves be implemented, but are instead used to establish a common denominator between types that should present shared behavior. Abstract classes can mimick the behavior of interface constructs in languages such as Java by declaring only *pure virtual* methods. Pure virtual methods are methods that do not expose any associated inline logic, and as such *must* be implemented by any inheriting subclasses. Consider WMCPP's `Connection` abstract class.

```
class Connection
{
public:
  virtual ~Connection() {}

  virtual void step(Event&) = 0;
  virtual bool close_window(Window) = 0;
  virtual void move_window(Window, Pos) = 0;
  virtual void resize_window(Window, Dim) = 0;
  virtual void focus_window(Window) = 0;
  // ...
};
```

The fact that this class contains *at least* a single virtual method declaration without an inline implementation (i.e., its declaration appears to be assigned to zero), makes it an abstract class. When not a single method (except for possibly its constructor or destructor) has an inline implementation, that class is considered to be a proper interface.

Although an abstract class's pure virtual methods cannot be called *dynamically* (i.e., using virtual dispatch), it may still implement associated logic that can subsequently be called *statically*. Consider part of `Connection`'s implementation.

```
#include "connection.hh"
#include "log.hh"

// ...
void
Connection::focus_window(Window window)
{
  Logger::log_info("Focusing window %s.",
    window.to_string());
  // ...
}
// ...
```

Calling a virtual function statically (non-virtually) is done by using its qualified name in the call. This is especially useful for implementors (derived classes) that all share an identical portion of code. In our example above, regardless of the underlying windowing system, the logging of an event is a fixed procedure, and can thus be part of the abstract class's implementation. Performing the call from WMCPP's `XConnection` class would look as follows.

```
#include "connection.hh"

// ...
class XConnection final: public Connection
{
public:
  // ...
  void focus_window(Window window) override {
    // non-virtual call
    Connection::focus_window(window);
    // ...
  }
  // ...
};
// ...
```

As can be seen in the above example, providing a concrete implementation for our abstract notion of a connection is done through *inheritance*. Each wrapper around the connection with *some* windowing system will be represented as a class that inherits (derives) from the `Connection` abstract class, providing an implementation specific to that windowing system.

WMCPP's `Model` class will then *contain* a reference to *some* implementation of `Connection`, as follows.

```
#include "connection.hh"

// ...
class Model final
{
public:
  Model(Connection& conn): conn(conn) {
    // ...
  }
  // ...
private:
  Connection& conn;
  // ...
};
```

Just as in our Rust implementation, `conn`'s implementation details will be resolved only at runtime, when they are needed, through dynamic dispatch (making use of a similar vtable mechanism).

While in our window manager implementations both Rust's traits and C++'s abstract classes appear to achieve the same objective in similar fashion, the constructs themselves are very different. For one, implementation (traits) is not the same as inheritance (abstract classes). By design, Rust does not allow for any type of inheritance (i.e. one object can inherit from another object's definition). The traditional benefits to inheritance are twofold: code reuse and polymorphism. Rust's traits attain the same functionality by combining implementation with *generics* and *trait bounds*. Traits allow for the central declaration (and possibly definition with their default implementations) of common behavior. The use of generics on top of traits introduces the possibility for abstractions over different possible types. Trait bounds then constrain these type abstractions, imposing restrictions on what exactly those types are to provide. WMRS's `Cycle` trait introduces a generic type, `T`, that is *bound* by the restriction `Identify + Debug`, stipulating that whatever concrete `T` gets passed in *must* implement both `Identify` and `Debug`.

Another useful feature of traits is the ability to implement crate-owned traits on external (and even built-in) types. Take our `Identify` trait as an example, which declares an `id` function that serves to uniquely identify an implementing object, and which is additionally a bound on the types passed into the `Cycle` struct. If we want our `Cycle` to work on the built-in 32-bit unsigned integer type, it would first need to implement `Identify`, which can be done as follows.

```rust
impl Identify for u32 {
  #[inline(always)]
  fn id(&self) -> Ident {
    *self
  }
}
```

C++'s abstract classes, on the other hand, do not allow for (re)derivation by already existing (external) types, without first creating some kind of wrapper object around them. This makes working with external code a lot more convenient in Rust compared to C++.

As inheritance often leads to the sharing of more code than necessary (due to the fact that *all* of a parent class's characteristics are inherited), many would consider it to be a suboptimal solution. Rust, with its trait objects, allows for tighter encapsulation, and altogether presents a more modern and flexible approach in that regard.

III. Main Event Loop

The main event loop in both WMRS and WMCPP comprises three core stages. They first rely on the underlying windowing system to report certain *events* they are interested in. This is done synchronously, as without any hardware interrupts or changes to the environment, nothing is to be done. That is, both window managers *block* until a new event is received. The second stage is the *extraction* and *bundling* of useful information from underlying windowing system events into a structure that can be consumed by the various components of each window manager. The last stage is using that windowing system event information to perform window management actions, *delegating work* to different parts of the program.

A. Windowing System Events

Both WMRS and WMCPP stipulate the existence of a `step` method in their `Connection` interfaces. This method is responsible for converting underlying windowing system information into a higher-level event, as can be gleaned from their signatures: `fn step(&self) -> Option<Event>` (WMRS) and `void step(Event&)` (WMCPP), where in both cases `Event` is some internally defined structure of data.

B. Internal Events

To structure event data, WMRS uses Rust *enumerations*. A Rust **enum** allows for the definition of a type by enumerating its variants, and, aside from encoding *meaning*, is also able to encapsulate *data*[5]. A small part of WMRS's `Event` **enum** looks as follows.

```rust
pub enum Event {
  Mouse { event: MouseEvent },
  Key { key_code: KeyCode },
  MapRequest { window: Window, ignore: bool },
  FocusRequest { window: Window },
  CloseRequest { window: Window },
  Randr,
  // ...
}
```

The first five variants are what are known as *struct-like* variants with named fields, whereas the last one is a fieldless variant that comprises only an identifier[9]. Behind the scenes, each **enum** instance has an associated integer that is used to determine the underlying concrete variant, known as a *discriminant*[9]. As such, Rust **enum**s are *tagged unions*, where the tag is the discriminant. This also means that the size of each instance of an **enum** is determined by that **enum**'s largest variant. Rust's *default representation* stores the discriminant as an **isize**, although the compiler is free to make use of a smaller type if the amount of variants permits it[9].

A similar construct in C++ is not *its* **enum**, but rather `std::variant`, a class template available since C++17 that represents a type-safe union[2]. Just as with Rust's **enum**, and similar to regular unions, the object representation of the concrete variant held inside a `std::variant` instance is allocated entirely within that instance's object representation[2]. That is, the variant may not allocate any additional dynamic memory[2]. As a result, just as with Rust's **enum**, the size of a `std::variant` instance is dependent on its largest variant[2]. WMCPP partly represents its `Event` type as follows.

```cpp
typedef std::variant<
  Mouse,
  Key,
  MapRequest,
  FocusRequest,
  CloseRequest,
  Randr,
  // ...
> Event;
```

Here, each concrete variant is a C++ **struct** containing information about the specific event that is being represented. This is similar to what WMRS does with its Rust **enum**s, only more verbose, non-inlined (as `std::variant` cannot deal with anonymous **struct**s), and therefore less structured.

C. Event Dispatch

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

IV. Communication with the Environment

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

A. *Inter-Process Communication*

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

V. Key Bindings

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

VI. Clients

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

A. *Reference Management*

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

B. *State*

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

VII. Workspaces

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

VIII. Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit

blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

References

[1] Robert D. Cameron. Four Concepts in Programming Language Description: Syntax, Semantics, Pragmatics and Metalanguage, 2002 (accessed May 5, 2021). URL https://www2.cs.sfu.ca/~cameron/Teaching/383/syn-sem-prag-meta.html.

[2] ISO/IEC. ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++. Geneva, Switzerland, 2020. International Organization for Standardization (ISO). URL https://isocpp.org/std/the-standard.

[3] The Cargo Developers. The Cargo Book, 2021 (accessed May 5, 2021). URL https://doc.rust-lang.org/cargo/.

[4] Aaron Turon. Abstraction without overhead: traits in Rust, 2015 (accessed May 5, 2021). URL https://blog.rust-lang.org/2015/05/11/traits.html.

[5] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, San Francisco, CA, USA, 2017. ISBN 1-59327-828-4 (paperback), 1-59327-851-9 (e-pub).

[6] The Rust Developers. Rust by Example, 2021 (accessed May 5, 2021). URL https://doc.rust-lang.org/rust-by-example/.

[7] Thomas Heartman. Rust traits: A deep dive, 2021 (accessed May 6, 2021). URL https://blog.logrocket.com/rust-traits-a-deep-dive/.

[8] Adam Schwalm. Exploring Dynamic Dispatch in Rust, 2017 (accessed May 6, 2021). URL https://alschwalm.com/blog/static/2017/03/07/exploring-dynamic-dispatch-in-rust/.

[9] The Rust Developers. The Rust Reference, 2021 (accessed May 6, 2021). URL https://doc.rust-lang.org/reference/.