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

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

June 19, 2021

***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 an ICCCM[2] and EWMH[3] compliant top-level reparenting, tiling window manager, built on top of the X Window System[4]: one written in Rust[5], and the other in C++.**

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

I. Introduction

Both Rust and C++ are *system programming* languages, as they offer high performance and ease of access to the underlying hardware. On top of that, Rust also offers novel memory safety guarantees that cater well to low-level system programming needs, as memory safety errors have historically been the main cause of security exploits. In this seminar, we will be discussing the *pragmatics* of Rust and C++. A language's pragmatics pertains to the *purpose* of its language constructs, and the usability of those constructs to achieve certain objectives. In general, the pragmatics of a language construct can be boundless. Take, for instance, the assignment statement. Its common purposes could be modifying part of a compound data structure, or using a temporary variable to swap two values. To properly compare the pragmatics of Rust and C++, instead of scrutinizing the purpose of *all* language constructs in turn, we will be evaluating both languages in their ability to specifically achieve a *single, common objective*: the implementation of a window manager. In particular, we identify four core phases in the design of such software. In section II, we will discuss each language's capacity in dealing with external dependencies. Particularly, we will look at which language features aid the programmer in managing external bodies of code, and how these dependencies can be abstracted upon and decoupled from. In section III, we will compare the languages' capabilities in wrapping external events, and in delegating work based on the specific kind of events generated. Section IV will cover language constructs that facilitate the storage and retrieval of callable objects, such as to allow for input bindings and event dispatch. Finally, in section V, each language's ability to allow for the mutation of distributed state is discussed.

II. External Dependency Management

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. As a result, it is traditionally not considered an aspect of that language's feature set per se. Notwithstanding, external dependency management is more and more becoming an appreciated addition to the *ecosystem* around a language, especially so for compiled languages. 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[4]. Although WMRS and WMCPP each use a different library to achieve this (XCB over `libxcb`[6] and Xlib over `libX11`[7], respectively), the methodology is the same, as they both require the importing of an external body of code.

A. C++: Ad Hoc Dependency Management

C++ does not come with an external dependency manager. That is to say, the official ISO standard C++ specification[8] 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 a Make script that manually encodes which files must be compiled.

B. *Rust: Automated Dependency Management using Cargo*

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[9]. 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*[9]. 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)[9]. A package's manifest file describes each of its target crates by specifying their type (library or binary executable), their metadata, and how to build them[9]. WMRS's manifest file defines a package that consists of a single library crate (`winsys`), along with three binary executable crates (`wmRS`, `wmRSbar`, and `wmRSclient`).

C. *Rust: Decoupling External Dependencies using Traits*

WMRS's `winsys` library is an abstraction above and wrapper around the API into 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*[10] collection of methods that is declared for some (at declare-time) unknown type `Self` (which at implementation-time becomes the implementing type), most often used with the intention to implement shared behavior[11,12]. A comparable concept from other languages that most closely resembles traits are *interfaces*, although there are differences still[11,13]. A small snippet from WMRS's `winsys` library `Connection` trait looks as follows:

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

Here, we have 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 have not 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[11,13]. 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). It 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[14] 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:

```
use x11rb::connection;
pub struct XConnection
  <'xconn, XConn: connection::Connection>
{
  xconn: &'xconn XConn,
  // ...
}
impl<'xconn, XConn: connection::Connection>
  Connection for XConnection<'xconn, XConn>
{
  fn step(&self) -> Option<Event> { /* ... */ }
  // ...
}
```

The `XConnection` structure introduces two generic type arguments, one being a lifetype parameter (`'xconn`), and the other being the type of the structure's `xconn` field (`XConn`), 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` structure implements our previously defined `Connection` trait, allowing it to be used by our window manager by means of composition. The listing below shows the structure that represents and encapsulates WMRS's core window manager logic. It *contains* a reference to *some* type that implements our `Connection` trait (i.e. a wrapper around *some* windowing system).

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

Since we are 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 spe-

cific version it should actually run (known as *dispatch*)[10,11]. 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[10,11]. 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[10,11]. 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.

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

This structure is used to cycle through both the workspace structures managed by the window manager, as well as the clients (discussed in section V) within a workspace. Two specific versions of this structure are consequently 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*[11,15]. A trait object is an opaque value of a type that implements some set of traits[16]. It is opaque, as one cannot know which concrete type is behind a trait object's pointer up front[16]. Whereas the size of each monomorphized type is always known, trait objects are therefore dynamically sized[16]. To resolve a call to one of such an opaque type's methods at runtime, *virtual method tables* (*vtables*) are used[16]. 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)[16]. Our `Model` structure's `conn` field is a reference to such a trait object. Behind the reference, we can have any implementation of the `Connection` trait. Which of the trait's implementors' methods are called, is determined at runtime.

D. C++: Abstraction through Abstract Classes

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 Event step() = 0;
  virtual void move_window(Window, Pos) = 0;
  virtual void resize_window(Window, Dim) = 0;
  virtual void close_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::close_window(Window window)
{
  Logger::log_info("Closing 0x%#08x.", window);
}
// ...
```

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 close_window(Window window) override {
    // non-virtual call
    Connection::close_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 virtual method table mechanism).

E. External Dependency Management: Remarks

C++'s ad hoc approach to package management and its ineffectiveness in assuring the availability of an external body of code are not the only aspects of external dependecy management that affect 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[8], 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). Rust's *module* system has neither of the issues C++ has with its include directives.

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*[10,11]. Traits allow for the central declaration (and possibly definition with their default implementations) of common behavior[10,11]. The use of generics on top of traits introduces the possibility for abstractions over different types[10,11]. Trait bounds then *constrain* these type abstractions, imposing restrictions on what exactly those types are to provide[10,11,16]. 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:

```
impl Identify for u32 {
  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. The window managers 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 bundled event information to perform window management actions, *delegating work* to different parts of the program.

A. Rust and C++: Windowing System Event Decoupling

As seen in the examples from the previous section, WMRS and WMCPP stipulate the existence of a `step` method in their respective `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 `Event step()` (WMCPP). In both cases, `Event` is some internally defined

structure of data that decouples low-level windowing system event specifics from that which is consumed at the higher level.

B. Rust: Internal Event Structuring using Enumerations

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*[11]. Part of WMRS's `Event` enumeration looks as follows:

```
pub enum Event {
  Mouse { event: MouseEvent },
  Key { event: KeyEvent },
  CloseRequest { window: Window },
  ScreenChange,
  // ...
}
```

The first thee 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[16]. Behind the scenes, each `enum` instance has an associated integer that is used to determine the underlying concrete variant, known as a *discriminant*[16]. 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[16].

C. C++: Internal Event Encoding with std::variant

A similar construct in C++ is not *its* `enum` (or `enum class`), but rather `std::variant`, a class template available since `C++17` that represents a type-safe tagged union[8]. Just as with Rust's `enum`, and similar to regular unions, the object representation of the concrete alternative held inside a `std::variant` instance is allocated entirely within that instance's object representation[8]. That is, the alternative may not allocate any additional dynamic memory[8]. As a result, as with Rust's `enum`, the size of a `std::variant` instance is dependent on its largest alternative[8]. WMCPP partly presents its `Event` type as follows:

```
typedef std::variant<
  std::monostate,
  Mouse,
  Key,
  CloseRequest,
  ScreenChange,
  // ...
> Event;
```

Here, each concrete alternative is a C++ `struct` containing information about the specific event that is being represented (e.g. `struct Mouse { MouseEvent event; };`). 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, since C++ does not have pattern matching), and therefore less structured and more tedious to work with.

D. Internal Events: Remarks

An astute reader may have noticed that the `Connection` interfaces of WMRS and WMCPP do not exactly match. In particular, their respective `step` functions' signatures seem to communicate different things. In WMRS, that function *optionally* yields an `Event`, wheres in WMCPP, it simply returns an `Event`. In C++, when using a `std::variant`, it is encouraged to encode an empty alternative using `std::monostate`[8]. The same behavior *could* instead have been encoded as `std::optional<Event>`, where in this case, the first alternative of `Event` is omitted. The difference in usage mainly lies in the *visiting* of `Event` alternatives, which we will discuss in the next section.

E. Rust: Event Dispatch through Pattern Matching

Once an event has been generated at the windowing system and filtered into a structure that is to be consumed by the window manager, work must be delegated to various handlers that deal with specific types of events. An `Event` consequently encodes not only the *information* used by these handlers, but also which handler is responsible for that event. WMRS uses Rust's principle pattern matching construct, `match`, for event dispatch. Its main event loop, defined within a method belonging to its `Model` structure, partly looks as follows:

```
while self.running {
  if let Some(event) = self.conn.step() {
    match event {
      Event::Mouse { event, }
        => self.handle_mouse(event, /*...*/),
      Event::Key { keycode, }
        => self.handle_key(keycode, /*...*/),
      Event::CloseRequest { window, }
        => self.handle_close_request(window),
      Event::ScreenChange
        => self.handle_screen_change(),
      // ...
    }
  }
}
```

The `step` method returns an `Option<Event>`, so the first thing to do is check whether a valid `Event` was retrieved from the windowing system connection (i.e., whether the event that was generated is one the window manager is interested in). Rust's `if let` construct is syntactic sugar for a match statement that runs code when the value matches one specific pattern, and ignores all other values[11]. Converting the above `if let` to an equivalent `match` results in the following:

```
match self.conn.step() {
  Some(event) => { /*...*/ },
  _ => (),
}
```

Within the body of the first match *arm*, the `event` identifier has been *bound* to the value that was passed in, *if* that value matches the pattern[11]. Here, that means that *if* a valid event is returned from the `step` method, that concrete `Event` instance will be bound to the `event` identifier within the body of that match arm. The `_` placeholder is used to denote all possible *remaining* values[11]. The Rust compiler will assert

that all possible cases are handled within a match statement, as matches are exhaustive[11].

Additionally matching on this `event`, all *variants* that can possibly exist for its type are enumerated, binding identifiers to each of the fields in the struct-like variants defined in the `Event` enum, which are then passed on to handler methods within the `Model` (e.g. `fn handle_close_request(window: Window)`).

F. C++: Delegation by Visiting std::variant Alternatives

C++ does not consist of a similar pattern matching construct. Its `switch` statement merely allows an identifier to be tested for *equality* against a list of values. Specifically, that means that `std::variant` alternatives cannot simply be `switch`ed on. Instead, the `std::visit` companion function exists to allow for the *visiting*[17] of a `std::variant` instance based on its possible alternatives[8,18]. It takes a *visitor* object that implements *function-call operator* overloads for each of the alternatives' types, along with the `std::variant` instance to visit. An example visitor for the `Event` variant, defined within WMCPP's `Model` class, partly looks as follows:

```
class Model;
class Model final
{
public:
  // ...
private:
  // ...
  struct EventVisitor
  {
    EventVisitor(Model& model): model(model) {}
    void operator()(std::monostate) {}
    void operator()(Mouse event) {
      model.handle_mouse(event);
    }
    void operator()(Key event) {
      model.handle_key(event);
    }
    void operator()(CloseRequest event) {
      model.handle_close_request(event);
    }
    void operator()(ScreenChange) {
      model.handle_screen_change();
    }
    // ...
  private:
    Model& model;
  } event_visitor = EventVisitor(*this);
  // ...
};
```

Whereas WMRS explicitly assured that a valid event was received, WMCPP *implicitly* does so through its `event_visitor`. The `std::monostate` variant represents the *no valid event* case, and is analogous to a *no-op* event. Within the same class, a method exists that contains WMCPP's main event loop which does the following:

```
while (running)
  std::visit(event_visitor, conn.step());
```

Here, we rely on the visitor to deduce the type of the event retrieved from the underlying connection's `step` method, subsequently delegating work to the relevant handler within WMCPP's model.

G. Event Dispatch: Remarks

While in this example both WMRS and WMCPP are able to express the desired behavior with relatively little code, visiting tagged union variants in C++ is clearly more verbose and less able in communicating intent than in Rust, and the discrepancies only worsen as more complex (or *pattern-reliant*) situations arise[18].

IV. Input Bindings

Key and mouse bindings are the central point of interaction between the user and the window manager. At a high level, as the name would suggest, they work by *binding* to sets of *input states*. These input states typically comprise keyboard (i.e., *which keys are pressed down?*) and mouse (*which mouse buttons are pressed down?*) inputs, but can in principle come from any kind of peripheral input source, such as sensors. Since input readings are hardware and, transitively, platform dependent, both WMRS and WMCPP rely on the connection with the windowing system to facilitate these bindings. That is—as can be seen from the `Connection` interfaces given in the examples above—both window managers receive specific events (e.g. `Key` and `Mouse`) that communicate that a specific binding has been activated.

A. Rust: Storing and Retrieving Callable Objects

To attach window management actions to these key and mouse bindings, WMRS uses a Rust `HashMap` to map mouse and key input state abstractions to *closures* that call into part of the `Model`. Mouse input abstractions look as follows:

```
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
pub enum MouseEventKind {
  Press, Release, Motion,
}
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
pub enum Button {
  Left, Middle, Right, ScrollUp, // ...
}
#[repr(u8)]
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
pub enum Modifier {
  Ctrl = 1 << 0, Shift = 1 << 1,
  Super = 1 << 2, Alt = 1 << 3, // ...
}
#[derive(PartialEq, Eq)]
pub struct MouseInput {
  pub button: Button,
  pub modifiers: HashSet<Modifier>,
}
#[derive(PartialEq, Eq, Hash)]
pub struct MouseEvent {
  pub kind: MouseEventKind,
  pub input: MouseInput,
  pub window: Option<Window>,
}
```

Since we are dealing with hardware and platform dependencies, WMRS again relies on the connection with the windowing system to establish the mapping from concrete underlying window system input encodings to the abstractions above. In particular, we expect a `MouseEvent` to be fired

(and propagated up through `Connection::step`) whenever a button is pressed. The event's input *modifiers* are contained within a Rust `HashSet`, as ordering is irrelevant and possible duplicates are redundant. Its `window` field is to optionally contain the unique identifier of the window that was clicked on (if any). The representation of key events is similar to the above, but instead of mouse buttons it includes non-modifier key identifiers.

The actual mapping between these input abstractions and hooks into the window manager's `Model` then looks as follows:

```
pub type KeyAction = Box<
  dyn FnMut(&mut Model<'_>)
>;
pub type MouseAction = Box<
  dyn FnMut(&mut Model<'_>, Option<Window>)
>;
pub type KeyBindings = HashMap<
  KeyInput, KeyAction
>;
pub type MouseBindings = HashMap<
  MouseInput, MouseAction
>;
```

These are simple *type alias* declarations, giving existing (in this case compound) types a new name. The first two represent closures that hook into (i.e. take as parameter) a *mutable*—or, more semantically explicative: *exclusive*—reference to the model instance[11]. `FnMut` is a trait that describes the calling of a function that is allowed to change (*mutate*) state; it permits its implementors to use call syntax, such that they can act and look like regular function pointers. `Box` *boxes* up a value, storing it on the heap and yielding a pointer to it (under the hood). Specifically, for `KeyAction`, that means that *some* type that implements the `FnMut` trait and takes a mutable reference to a model instance (`dyn FnMut(&mut Model<'_>)`) is boxed up and stored on the heap for future reference. We require the use of `Box`, because we want to be able to store *any implementor* of the `FnMut` trait. Since the size of an arbitrary implementor is not fixed (in Rust terms: `FnMut` does not implement the `Sized` trait), we require this indirection through `Box`, which will point to a virtual table. To retrieve the exact function to call, Rust will again rely on dynamic dispatch[19]. The advantage of this approach is that it facilitates the changing of input bindings at runtime, allowing for multi-mode hierarchical input bindings, as well as dynamic user-initiated binding customizations.

To be able to use `KeyInput` and `MouseInput` as keys to the `KeyBindings` and `MouseBindings` `HashMaps`, they must satisfy several conditions. These conditions are expressed through the trait bounds `PartialEq`, `Eq`, and `Hash`, respectively defining partial equivalence and equivalence relations, enabling the comparison for (partial) equality, and a hash function, allowing instances of its implementors to be *hashed* with an implementation of `Hasher`[11,16]. `PartialEq` and `Eq` can be automatically implemented for `KeyInput` and `MouseInput` by making use of Rust's `derive` attribute. This attribute asks the compiler to provide basic implementations for built-in (and even self-written, if an implementation is first defined using *procedural macros*) traits, given that the trait and the type requiring the implementation are *derivable*[11,16]. For `PartialEq` and `Eq` in particular, all fields of a to-be-derived compound type *must* already implement `PartialEq` and `Eq`, which is the case for `MouseEvent`'s fields given in the example above. The same condition holds for `Hash`, although `HashSet`, the type of `MouseInput`'s `modifiers` field, is not automatically derivable[11,16]. To satisfy the `Hash` trait bound for `MouseInput`, it must therefore be defined manually. To this end, each variant of WMRS's `Modifier` enumeration is represented by an unsigned 8-bit integer value (`#[repr(u8)]`). The values are assigned in such a manner that the `enum` acts as a collection of *bit flags*, resolving each variant *and* the logical disjunction of any combination of variants into a unique value. `MouseInput`'s `Hash` implementation then looks as follows:

```
impl Hash for MouseInput {
  fn hash<H: Hasher>(&self, state: &mut H) {
    self.button.hash(state);
    self.modifiers.iter()
      .fold(0u8, |acc, &m| acc | m as u8)
      .hash(state);
  }
}
```

Hashing works relative to an object that implements the `Hasher` trait (`<H: Hasher>`)[11]. The methods required by `Hasher` are to contain the logic of the hashing function in use, and an instance of an implementor of the trait will usually represent some *state* that changes as data is being hashed[11,20]. The final state of such an instance then represents the hashed value[11,20]. Hashing structures that expose this behavior are called *streaming* hashers[21]. To obtain the hashed value of a `MouseInput` instance, we change, in ordered succession, the state of the `Hasher` object by supplying the hasher the `button` field's value, followed by the logical disjunction of all of the `Modifier` variants contained in the `modifiers` field. When looking for the matching value of a supplied key in the `HashMap`, Rust will use the key instance's `hashed` value to compute the *"bucket"* from which to start the search, and `eq` (the comparison method implemented through `PartialEq` and `Eq`) to actually perform the search.

To finally add a new mouse binding, we insert an input/action pair into the `MouseBindings` `HashMap`, as follows:

```
let mut mouse_bindings = HashMap::new();
mouse_bindings.insert(
  MouseInput {
    button: Button::Middle,
    modifiers: {
      let mut modifiers = HashSet::with_capacity(3);
      modifiers.insert(Modifier::Ctrl);
      modifiers.insert(Modifier::Shift);
      modifiers.insert(Modifier::Super);
      modifiers
    },
  },
  Box::new(|model: &mut Model, win: Option<Window>| {
    if let Some(window) = win {
      model.set_fullscreen_window(window);
    }
  })
);
```

To retrieve a mouse binding and call its function, a handler method within WMRS's `Model` gets passed in a `MouseEvent` that contains information about the button and modifiers that had been pressed. The contained logic looks somewhat as follows:

```
fn handle_mouse(
  &mut self,
  event: MouseEvent,
  mouse_bindings: &MouseBindings,
) {
  // ...
  if let Some(action)
    = mouse_bindings.get(&event.input)
  {
    action(self, event.window);
  }
  // ...
}
```

Population of and retrieval from WMRS's `KeyBindings` `HashMap` is done in similar fashion.

B. C++: Bindings with std::function and std::unordered_map

To achieve an analogous mouse and key binding setup in WMCPP, we first create C++-equivalent scoped enumerations (`enum class`) for `MouseEventKind` and `Button`, and an unscoped enumeration (C-style `enum`) for `Modifier`. Unfortunately, an `enum class` cannot be used to represent modifier keys, as that construct is not effective in encoding bit flags. The reason for this is that the logical disjunction of two scoped bit flag enumerators would not resolve into another valid scoped enumerator. Additionally, the following `struct`s are constructed to represent `MouseInput` and `MouseEvent`:

```
struct MouseInput final {
  const Button button;
  const std::unordered_set<Modifier> modifiers;
};
struct MouseEvent final {
  MouseEventKind kind;
  MouseInput input;
  std::optional<Window> window;
};
```

Just as in WMRS, a hash map construct (*associative container* in C++ parlance) is used to map input abstractions to closures that call into the `Model` object. Specifically, `std::unordered_map`—and not `std::map`, which *orders* its entries by key and is therefore (in this case needlessly) less efficient—will take as key arbitrary objects of type `T` that provide overloads for `operator==` and `std::hash<T>::operator()`. The first is the equality comparison operator, comparable to Rust's `PartialEq::eq` trait method. The `std::hash<T>` template's `operator()` overload implements a hash function that, when called, converts instances of type `T` into an unsigned integer (`std::size_t`) that represents the hashed value. WMCPP's implementation for both overloads looks as follows:

```
inline bool
operator==(const MouseInput& lhs, const MouseInput& rhs)
{
  return lhs.button == rhs.button
    && lhs.modifiers == rhs.modifiers;
}

namespace std
{
  template <>
  struct hash<MouseInput>
  {
    std::size_t
    operator()(const MouseInput& input) const
    {
      std::size_t button_hash
        = std::hash<Button>()(input.button);
      std::size_t modifiers_hash
        = std::hash<std::size_t>()(
          std::accumulate(input.modifiers.begin(),
            input.modifiers.end(),
            static_cast<Modifier>(0),
            std::bit_or<Modifier>()
          )
        );
      return button_hash ^ modifiers_hash;
    }
  };
}
```

Conceptually, the same is happening here as in WMRS's overloads for `PartialEq::eq` and `Hash::hash`. Comparison of `MouseInput` objects is simply done based on their constituent fields. To obtain the hashed value of such an object, we take the *exclusive or* of the hashed values of both the `button` field and the logical disjunction of the values in the `modifiers` `std::unordered_set`. The `std::accumulate` function is similar to Rust's `Iterator::fold` method that *folds* every element of a collection into an accumulator by applying some operation. In both cases, this operation is the logical disjunction (bitwise or), and the accumulator is initialized to zero. The reason `Modifier` needed to be an unscoped enumeration is due to this particular use of the `std::bit_or` operation. However, something is still missing. Since, out of the box, the bitwise or operator is not defined on `Modifier`'s enumerators, we must first define it ourselves:

```
inline Modifier
operator|(Modifier lhs, Modifier rhs)
{
  return static_cast<Modifier>(
    static_cast<std::size_t>(lhs)
      | static_cast<std::size_t>(rhs)
  );
}
```

Now, to capture and store hooks into the `Model` instance in the `unordered_map`, the `std::function` class template is used, a polymorphic function wrapper that encapsulates a callable *target*. Particularly, as in WMRS, `MouseAction` is simply a type alias:

```
typedef
  std::function<void(Model&, std::optional<Window>)>
  MouseAction;
```

Just like Rust's `FnMut`, calling a `std::function` instance will make use of dynamic dispatch to retrieve the logic to execute[8]. The reason this virtual lookup is required is because, under the hood, the callable object gets wrapped inside a structure that additionally contains possibly bound data[8]. As a result, the exact type and, by extension, its size, will not be

known to callers of the `std::function` instance[22]. A benefit to C++'s `std::function` over Rust's `FnMut` is that memory will not be dynamically allocated when storing lambda functions without any bound parameters, thanks to C++'s *Small Buffer Optimization*[22,23]. That implies that retrieving such instances from a growable collection, such as a hash map, saves us one level of indirection.

The `MouseBindings` type that represents the map between input abstractions and actions is also a simple type alias:

```
typedef
  std::unordered_map<MouseInput, MouseAction>
  MouseBindings;
```

An instance of this type is stored as a member field inside of the `Model` instance. Registering the initial bindings happens through `Model`'s constructor, and can be done rather compactly using *aggregate initialization* as follows[?]:

```
Model(/* ... */)
  : mouse_bindings({
    { { Button::Middle, { Ctrl, Shift, Super } },
      [](Model& model, std::optional<Window> win) {
        if (win) model.set_fullscreen_window(*win);
      }
    },
    // ...
  })
{ /* ... */ }
```

To subsequently retrieve a mouse binding and call its target, the relevant `MouseEvent` event dispatches work to a handler method that partly comprises the following:

```
void
Model::handle_mouse(MouseEvent& event)
{
  MouseBindings& mb = this->mouse_bindings;
  if (mb.find(event.input) != mb.end())
    mb.at(event.input)(*this, event.window);
}
```

C. Input Bindings: Remarks

Whereas Rust uses streaming hashers, consuming an arbitrary amount of data byte-per-byte until it is told to yield a hashed summary, C++'s `std::hash<T>::operator()` relies on so-called *hash coding*[21]. Hash coding is a lot less convenient—as can clearly be seen in the examples provided above—as complex data types are expected to be reduced into a single integer within the body of a single function[21]. Moreover, C++ does not allow for a central (e.g. on a per `HashMap` instance basis) hashing algorithm to be defined, which Rust does allow for through its `Hasher` trait[11,16].

Rust's way of overloading operators through trait implementations is also a much cleaner approach than that of C++. Depending on the operator, C++ operator overloading is done through class methods, non-method functions, non-member friend functions, or even by providing a concrete implementation to a templated class, such as in our `std::hash<MouseInput>::operator()` example above[8].

In WMRS, simply registering a single mouse binding is quite verbose. To aid in populating the `HashMap`, and to increase readability and programmer convenience, Rust *macros* can be used. In particular, *declarative macros* facilitate a form of *metaprogramming*—a way of *writing code with code*[11,16,24]. Declarative macros allow code to be generated by providing `match`-like syntax, which is particularly useful for elimiting duplicate code. Function-like declarative macros could reduce the code from our example above into the following:

```
let mouse_bindings = init_mouse_bindings!(
  1-C-S-Middle => execute_model_window!(model, win,
    if let Some(window) = win {
      model.set_fullscreen_window(window);
    }
  ),
  // more mouse bindings
);
```

Here, `init_mouse_bindings!` and `execute_model_window!` are macro invocations. Another form of Rust macros, called *procedural macros*, are even more powerful, but also more complex[24]. These macros allow the Rust syntax to be expanded; they take arbitrary input, operate on the abstract syntax tree, and produce valid Rust code[24]. C++ does not have a similarly powerful meta-programming feature[8].

V. Clients

So far, the notion of a *window* was used throughout without defining what exactly it is to entail. In both WMRS and WMCPP, the `Window` type is a unique identifier (e.g., an unsigned integer) for a window that, again, serves to map windowing system specific representations to an abstraction that is to be used within the window manager. A window in and of itself therefore does not contain any *state*. To associate state with a window, we wrap around it a structure that we call a *client*. Clients are objects that contain all relevant information about a window with regard to the window manager: its title, position and size on-screen, process identifier, whether it is in fullscreen mode, and much more.

A. Rust: Mutable State with Cell and RefCell

We have thus far deferred discussing a major aspect of the Rust programming language: *mutability*. Rust's `mut` keyword carries two kinds of semantics, depending on where it is used. In patterns, `mut` indeed indicates that changing the underlying value is allowed[11,25]. In references, however, it rather pertains to *exclusivity*. That is, a `&mut` reference may be mutated, but additionally is not allowed to be *aliased*[11,25]. The reason this rule exist, is because Rust's safety foundation is based on the principle *aliasing* `XOR` *mutability*: An object can only be safely mutated if no other reference exists to its contents[11,16]. Consider part of WMRS's `Client` structure:

```
pub struct Client {
  window: Window,
  workspace: Cell<usize>,
  parent: Option<Window>,
  children: RefCell<Vec<Window>>,
  fullscreen: Cell<bool>,
  // ...
}
```

We want to pass around references to instances of this structure throughout the `Model`, to read and alter state, and to

make window management decisions accordingly. Disregarding the concrete contents of the `Client` structure for a moment, one possibility would be to have the different client-mutating methods that are part of the `Model` operate on a `&mut Client`. While this is a plausible approach, it heavily limits us in using and passing around references as, in concreto, no more than a single reference to a to-be-mutated `Client` may then exist at a time[11]. This becomes particularly troublesome, as we need to store `Client` instances within the `Model`, mapping window representations to their associated `Client` structure, as follows:

```
pub struct Model<'model> {
  client_map: HashMap<Window, Client>,
  // ...
}
```

Since mutability transitively affects all surrounding structures, mutating a single field within a `Client` instance means that the reference to that instance would have to be `&mut`. This additionally requires that the `Model`'s method that retrieves from its `client_map` and operates on such an instance would *also* have to take `&mut self`. For this reason, this kind of mutability is what's known in Rust as *inherited* mutability, also referred to as *external* mutability[11].

Considering that the windowing system generates events in respect to some *window*, whenever we need to alter the state that is associated with that window, we must therefore first retrieve a mutable reference to its client from the `client_map`. A result of this, is that something like the following would not be possible:

```
impl<'model> Model<'model> {
  fn set_fullscreen_window(&mut self, win: Window) {
    if let Some(c) = self.client_map.get_mut(&win) {
      c.set_fullscreen(true);
      self.apply_layout(c.workspace());
    }
  }
  fn apply_layout(&self, index: usize) { /* ... */ }
  // ...
}
```

Even though `apply_layout` can be called on a non-exclusive reference to the model (i.e. it takes `&self`), the above will not compile: An immutable borrow of `self` (`self.apply_layout(c.workspace())`) occurs *after* a mutable one (`self.client_map.get_mut(&win)`) inside `set_fullscreen_window`, which is not allowed due to the exclusivity rules discussed before. Because every method that directly manipulates a client has a structure analogous to the `set_fullscreen_window` method, this approach is not viable.

To solve this, we use a technique called *interior mutability*. Interior mutability enables the mutating of an underlying structure of data with a *non-exclusive* reference to it, therein defying the requirement that only exclusive references may be mutated[11]. To facilitate this, four language constructs exist: `Cell` and `RefCell` for single-threaded solutions, and `Mutex` and `RwLock` as their thread-safe alternatives[11,16].

In view of the fact that window managers operate on an *ordered* stream of events, they function well as event-driven, single-threaded applications. To that end, each `Client` field that is expected to mutate throughout program execution—representing state changes instigated by windowing system events—gets wrapped inside either a `Cell` or a `RefCell`. `Cell` deals solely with *values*: values are moved or copied into the cell, or retrieved from it as a whole[11,25]. Retrieving a reference to the contents of a `Cell` is not possible[11,25]. `RefCell`, on the other hand, works exclusively with *references*, and implements so-called *dynamic borrowing*[11,25]. That means that access to mutable references to the contents of a `RefCell` are tracked *at runtime*, which will cause a panic if a second mutable borrow is attempted while another is already outstanding[11,25]. Compared to *static borrowing*, at a slight runtime cost, dynamic borrowing permits much more flexible use of references, since the compiler is only able to perform borrow checks at a rather coarse level of granularity[11,25].

From a practical point of view, what `Cell` and `RefCell` allow us to do, is to instead take a non-exclusive `&Client` as parameter, and mutate individual fields, therein enabling reference aliasing *and* retaining mutability. In a sense, interior mutability makes it possible to *divide* mutability of an entire structure into its individual constituents. Consider the following `Client` method:

```
impl Client {
  pub fn set_fullscreen(&self, bool: value) {
    // copy `value` into the Cell
    self.fullscreen.set(value);
  }
  // ...
}
```

Thanks to interior mutability, to be able to call this method on a `Client` reference, that reference does not need to be exclusive (due to it operating on `&self` instead of `&mut self`). This allows us to rewrite the `set_fullscreen_window` method as follows:

```
fn set_fullscreen_window(&self, win: Window) {
  if let Some(c) = self.client_map.get(&win) {
    c.set_fullscreen(true);
    self.apply_layout(c.workspace());
  }
}
```

Using this technique, *most* `Model` methods will now take `&self`, therein greatly improving flexibility, and consequently allowing for a much better structuring of code.

B. C++: Implicit interior mutability with references

Disregarding the non-standardized `restrict` keyword, in C++, *every* reference (and pointer) can be aliased[8]. Moreover, there is no inherent restriction on the mutability of references to an object. Compared to Rust, all mutability in C++ would therefore be considered interior mutability. Part of WMCPP's `Client` structure looks as follows:

```
typedef struct Client* Client_ptr;
typedef struct Client final
{
  Window window;
  unsigned workspace;
  Client_ptr parent;
  std::vector<Client_ptr> children;
  bool fullscreen;
```

```
  // ...
}* Client_ptr;
```

Instead of encoding relationships with other client objects (i.e. `parent` and `children`) using window representations (`Window`), we store *pointers* to the associated objects directly. Furthermore, we do not need to make use of smart pointers (e.g. `std::shared_ptr`). The reason we can safely store *raw* pointers to other `Clients` within a client structure, is thanks to a main event loop invariant we rely on: For each window that gets destroyed, an event will be generated by the windowing system, and propagated up to the window manager. That is, we know *exactly* when to deallocate a `Client` and all of its `children`. To map window representations to their associated client structures, we again make use of a `std::unoredered_map`:

```
class Model final
{
  // ...
private:
  std::unordered_map<Window, Client_ptr> client_map;
  // ...
};
```

Each `Client_ptr` in the map is a heap-allocated object. To change the state of a window, the `Model` has associated methods that retrieve and mutate client objects, an example of which is the following:

```
void
Model::set_fullscreen_window(Window window)
{
  if (client_map.find(window) != client_map.end()) {
    Client_ptr client = client_map.at(window);
    client->fullscreen = true;
    apply_layout(client->workspace);
  }
}
```

Most client-mutating methods that are part of the `Model` follow the same approach.

C. Clients: Remarks

While Rust's mutability rules make the language much more secure in regard to memory-safety and data races, its borrowing system is actually more restrictive than strictly necessary to ensure these guarantees. That is to say, in a *perfect implementation*, static (i.e. compile-time) borrow checking would be as flexible as dynamic (i.e. runtime) borrow checking. The main reason for the existence of Rust's interior mutability constructs is that this perfect implementation *does not exist*. And, for several good reasons, two of which being language simplicity and compilation speed.

From a pure pragmatics standpoint, Rust's mutability restrictions add significant additional accounting effort. In fact, they even render the implementation of some programming patterns impossible. For instance, compared to our C++ examples, self-referential structures are not allowed in Rust. For this reason, the only way to encode relationships to other instances is through use of some kind of non-referential indirection. For our `Client` structure in particular, that means we need to store the window representations of `parent` and `children` instead of direct references to their `Client` instances. This adds both a programming and runtime overhead, as we repeatedly need to retrieve their instances through the `client_map` whenever we need to read or alter their state.

VI. Conclusion

In this seminar, we have compared Rust and C++ in their *pragmatics* when writing medium to large system software. Specifically, we have sought to achieve *the same objective* using both languages: the implementing of a window manager. In doing so, we have discussed four core phases in the implementation of such software: external dependency management, the main event loop and event dispatch, input bindings and the storage of callable objects, and, finally, the central storage and distributed mutating of state.

While Rust is able to make certain guarantees in regard to memory safety and data races—making it an eminently safer choice compared to C++—it is additionally often able to capture in its syntax more concisely and eloquently the semantics required to achieve an objective. This has much to do with its more minimal and simplistic, but at the same time more expressively powerful language features, such as, for example, the pattern matching constructs reminiscent of functional languages, or its tagged union enumerations. Another decisive factor in Rust's perceived efficacy is its *consistency*. Whereas C++ offers operator overloading through a multitude of constructs (varying from class methods to concrete template class implementations), Rust does so *solely* through trait implementations.

Rust's safety guarantees indeed make the language demonstrably superior from an *error mitigation* point of view. However, those guarantees can at times still come at a cost in light of its pragmatics. Assuming memory safety assertions are properly taken care of, dealing with references can be much easier in C++ than in Rust. Due to the fact that Rust's borrow checking system is more restrictive than strictly necessary, certain programming patterns are unable to be used, such as, for instance, self-referential structures. In fact, Rust even has *interior mutability* constructs just to improve programming flexibility, something C++ has by default.

All in all, the Rust programming language is the product of learning from years and years of how best *not* to do it, while at the same time borrowing features from languages that *did* get it right. Due to the complexity and strictness of its borrow checking system, the developers have sought to keep the core language as simple yet expressive as possible. C++, on the other hand, lacks in its design the *hindsight* Rust has profited from so much, and is in comparison immensely complex—some would even say convoluted. This results from decades of revisions that add new, powerful language features, which are all required to coexist nicely with older constructs that were initially not designed to cater to these new features, all the while maintaining the design goals of the initial core language.

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] David Rosenthal and Stuart W. Marks. Inter-Client Communication Conventions Manual. 1994. URL https://www.x.org/releases/X11R7.6/doc/xorg-docs/specs/ICCCM/icccm.html.

[3] X Desktop Group. Extended Window Manager Hints, 2005. URL https://specifications.freedesktop.org/wm-spec/wm-spec-1.3.html.

[4] Robert W. Scheifler and Jim Gettys. The X Window System, Version 11. *Software: Practice and Experiment*, 1990.

[5] Max van Deurzen. wzrd: An ICCCM & EWMH compliant X11 (XCB) reparenting, dynamic window manager, written in Rust, 2021 (accessed May 25, 2021). URL https://github.com/deurzen/wzrd.

[6] Bart Massey and Jamey Sharp. XCB: An X Protocol C Binding. 2001.

[7] James Gettys and Robert W. Scheifler. Xlib – C Language X Interface. 1985.

[8] 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.

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

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

[11] 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).

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

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

[14] Kristian Høgsberg. The Wayland Protocol. URL https://wayland.freedesktop.org/docs/html/ch01.html#sect-Motivation.

[15] 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/.

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

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612.

[18] Matt Kline. std::visit is everything wrong with modern C++, 2021 (accessed May 6, 2021). URL https://bitbashing.io/std-visit.html.

[19] Steve Donovan. Why Rust Closures are (Somewhat) Hard, 2018 (accessed May 18, 2021). URL https://stevedonovan.github.io/rustifications/2018/08/18/rust-closures-are-hard.html.

[20] Ivan Dubrov. Tricking the HashMap, 2018 (accessed May 19, 2021). URL http://idubrov.name/rust/2018/06/01/tricking-the-hashmap.html.

[21] Alexis Beingessner. Building a HashMap in Rust - Part 1: What's a HashMap?, accessed May 22, 2021. URL https://cglab.ca/~abeinges/blah/robinhood-part-1/.

[22] Shahar Mike. Under the hood of lambdas and std::function, 2016 (accessed May 22, 2021). URL https://shaharmike.com/cpp/lambdas-and-functions/#std-function.

[23] Andrzej Krzemieński. Common optimizations, 2014 (accessed May 22, 2021). URL https://akrzemi1.wordpress.com/2014/04/14/common-optimizations/.

[24] Anshul Goyal. Macros in rust: A tutorial with examples, 2021 (accessed May 23, 2021). URL https://blog.logrocket.com/macros-in-rust-a-tutorial-with-examples/.

[25] Paul Dicker. Interior mutability patterns, 2019 (accessed June 2, 2021). URL https://pitdicker.github.io/Interior-mutability-patterns/.