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

Max van Deurzen

June 21, 2021

Technische Universität München

Agenda

Agenda

1. What is *Pragmatics*?
2. The *Common Objective*
3. External Dependency Management
4. Main Event Loop
5. Input Bindings
6. Clients
7. Results
8. Discussion

Pragmatics

1. **Syntax**
   Set of rules that define the *structure* and *composition* of allowable symbols into correct statements or expressions in the language

2. **Semantics**
   The *meaning* of these syntactically valid statements or expressions

3. **Pragmatics**
   "...[T]he third general area of language description, referring to practical aspects of how constructs and features of a language may be used to achieve various objectives."

   Robert D. Cameron, 2002

Example: Assignment Statements

1. **Syntax** (*structure*)

   $x = y * 3;$

2. **Semantics** (*meaning*)

   - x

     Location in memory

   - $y * 3$

     Computation of a value based on an expression

   - $x = y * 3;$

     Store result of expression evaluation in location in memory

3. **Pragmatics** (*purpose*)

   Which objectives are assignment statements used *for*?

   - Setting up a temporary variable used to swap the values of two variables
   - Modifying some part of a compound data structure
   - ...

The Common Objective

Case Study: The implementation of a window manager

- **System software**
  - Low-level
  - Platform-specific
- **Medium to large-sized**
  - Increased risk of *code smells*
    - Monolithic classes
    - Global data
    - High interdependence (coupling)
    - ...
- **Event-driven**
  - Reacts to windowing system events
  - Deterministic event dispatch

Case Study: The implementation of a window manager

- **External Dependency Management**
  - Package management
  - Abstracting and decoupling
- **Main Event Loop**
  - Windowing system events
  - Internal events
  - Event dispatch
- **Input Bindings**
  - Storing and retrieving callable objects
- **Clients**
  - Distributed, mutable state

Case Study: The implementation of **two** window managers

- **Same structure**
    - Built on top of the X Window System
        - Library to communicate with the X server as external dependency
- **Same behavior**
    - ICCCM and EWMH compliant
    - Reparenting, tiling
- **Different languages**
    - One implemented in C++: WMCPP
    - One implemented in Rust: WMRS

External Dependency Management

External Dependency Management

Practicalities of working with external code

1. **Package management**
   - *Availability* of external code
2. **Decoupling dependencies**
   - *Maintainability* of external code

Managing the availability of external code

- The ability to *aid* the programmer in assuring availability
  - Automatically download and compile source code
  - Built-in version control
  - Conflict detection
- Part of the *ecosystem* of a language
  - Installed with its compiler or development environment
- A *must* for any modern programming language

- *No* official package manager
- *Ad hoc* package management
  - Third-party package management tools
    - *Conan*
    - *Vcpkg*
    - *build2*
  - Custom configure and build scripts
  - Let the user manage the dependencies themselves (e.g. through their distribution's package manager)
- Example: Make script

```
CXXFLAGS  := -std=c++20 -march=native -O3
LDFLAGS   := `pkg-config --libs x11 xrandr` -flto
SRC_FILES := $(wildcard src/*.cc)
OBJ_FILES := $(patsubst src/%.cc,obj/%.o,${SRC_FILES})
all: ${OBJ_FILES}
	g++ ${OBJ_FILES} ${LDFLAGS} -o bin/wmCPP
obj/%.o: src/%.cc
	g++ ${CXXFLAGS} -MMD -c $< -o $@
```

- *Cargo*, Rust's official package manager
  - Automatically downloads and compiles dependencies
  - A Rust project is a Cargo *package*
  - A package is a collection of *source files* plus a *manifest* file
  - The manifest file describes the package's *meta-information*, *dependencies*, and a set of *target crates*
  - A crate represents a *library* or *binary executable* program
- Example: `Cargo.toml` manifest file

```
[package]
name = "wmRS"
version = "0.1.0"
edition = "2018"
license = "BSD3"
default-run = "core"
description = """
An ICCCM & EWMH compliant X11
reparenting, tiling window manager,
written in Rust
"""
```

```
[lib]
name = "winsys"
path = "src/winsys/mod.rs"
[[bin]]
name = "core"
path = "src/core/main.rs"
[[bin]]
name = "client"
path = "src/client/main.rs"
[dependencies]
x11rb = "0.8.0"
```

Managing the maintainability of external code

- The ability to *decouple* own code from external code
  - Changes to own code don't affect interface with external code
  - Changes to external code *only* affect inerface with external code
- When external code changes:
  - Only interface with external code needs to be recompiled
- When own code changes:
  - Only own code needs to be recompiled

Decouple window manager from windowing system

1. Hide the connection with the windowing system behind an *interface*
    - Provide *abstraction* and *encapsulation*
    - Describe *common behavior*
    - *Usage* is *agnostic* of concrete implementation
2. Implement the interface for *each* targeted windowing system
    - *X Window System*
    - *Wayland*
    - *Desktop Window Manager* (Windows)
    - *Quartz Compositor* (macOS)
    - ...
3. Have the window management logic call into the interface

1. Hide the connection with the windowing system behind a trait

- *Zero-overhead* collection of methods
  "What you don't use, you don't pay for [Stroustrup, 1994]. And further: What you do use, you couldn't hand code any better."

  Bjarne Stroustrup

- Comparable to, *but not the same as*, the concept of an OOP *interface*
  - Implementation does not require changes to the implementor
    - Traits can be implemented on *external* code
    - No ambiguity when two implemented traits share method name and signature
- Can define *stateless* default implementations

1. Hide the connection with the windowing system behind a trait

- No inheritance, only implementation
  - No downcasting or reference casting
- Declared for some (at declare-time) unknown type `Self`
  - When implemented `Self` becomes the implementing type
- Example: WMRS's `Connection` trait:

```
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);
    // ...
}
```

2. Implement the trait for each targeted windowing system

- Example: WMRS's `XConnection` structure:

```
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> { /* ... */ }
    // ...
}
```

- `x11rb`: Rust library to interact with the X Window System
  - External dependency
  - Rust bindings to interact with the X server

3. **Have the window management logic call into the interface**

- Example: WMRS's core window manager logic:

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

- *Polymorphism* to abstract away from the concrete implementation
- `Model` *contains* a reference to *some* `Connection` implementor
- The trait methods of this implementor are called where needed
  - Static dispatch
    - Concrete method to call is baked into the binary
  - Dynamic dispatch
    - Concrete method to call is looked up *at runtime*

Static dispatch

- Concrete method to call is baked into the binary
    - *Monomorphization* at compile time
    - Generic code is converted into “specific” code
    - One version for each concrete type used as generic argument
    - Size of concrete type is always known
- No additional time overhead at runtime
- Example: WMRS's `Cycle` structure:

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

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

```rust
pub struct Model<'model> {
    // ...
    workspaces: Cycle<Workspace>,
    // ...
}

pub struct Workspace {
    clients: Cycle<Window>,
}
```

Dynamic dispatch

- Concrete method to call is looked up *at runtime*
- *Trait objects* keep instances abstract until concretization is required
  - Opaque value of a type that implements some set of traits
  - Until further inspection, concrete type is unknown
  - *Dynamically sized*: size of underlying concrete type is not known up front
- Under the hood, 2 pointers:
  - 1 pointer to data
  - 1 pointer to *virtual method table* (*vtable*)
- Virtual method table points to that object's concrete method implementations

Dynamic dispatch

- Example: WMRS's `XConnection`'s `xconn` reference:

```
x11rb::connection;

pub struct XConnection<'xconn, XConn: connection::Connection> {
    xconn: &'xconn XConn,
    // ...
}
```

- Example: WMRS's `conn` trait object:

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

Dynamic dispatch

- ~~Example: WMRS's XConnection's xconn reference:~~

```
use x11rb::connection;

pub struct XConnection<'xconn, XConn: connection::Connection> {
    xconn: &'xconn XConn,
    //...
}
```

- Example: WMRS's conn trait object:

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

1. Hide the connection with the windowing system behind an abstract class

- Abstract type that cannot be implemented, only *derived*
- Establish common denominator between types
- Can define *stateful* default implementations
- Same as OOP interface when it *only* contains *pure virtual* methods
  - No associated inline logic
  - *Must* be implemented by inheriting subclasses
- Derived class concrete method invocation *only* through dynamic dispatch

1. Hide the connection with the windowing system behind an abstract class

- Example: WMCPP's `Connection` abstract class interface:

```
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;
    // ...
};
```

- Connection contains *at least* 1 virtual method
  - Connection is an abstract class
- Connection has 0 inline method implementations
  - Connection is a proper OOP interface

1. Hide the connection with the windowing system behind an abstract class

- Pure virtual methods *can* be defined to be called *statically*
- Example: WMCPP's `Connection`'s implementation:

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

void
Connection::close_window(Window window)
{
    Logger::log_info("Closing 0x%#08x.", window);
}

// ...
```

2. Derive the abstract class for each targeted windowing system

- Example: WMCPP's `XConnection` derived class:

```
#include "connection.hh"
extern "C" {
#include <X11/Xlib.h>
// ...
}
class XConnection final: public Connection
{
public:
    void close_window(Window window) override {
        Connection::close_window(window); // non-virtual call
        // ...
    }
    // ...
};
```

- `<X11/...>`: Xlib library to interact with the X Window System
  - External dependency

3. Have the window management logic call into the interface

- Example: WMCPP's core window manager logic:

```
#include "connection.hh"
class Model final
{
public:
    Model(Connection& conn): conn(conn) { /* ... */  }
    // ...
private:
    Connection& conn;
    // ...
};
```

- *Polymorphism* to abstract away from the concrete implementation
- `Model` *contains* a reference to *some* `Connection` implementor
- The overridden methods of this implementor are dynamically called where needed

Additional C++ external dependency management difficulties:

- Problem: *double inclusion*
- Possible solution: **header guards**
  - Preprocessor directives
  - Include idempotence
  - Not fail-safe
  - Hard-to-trace symbol collision errors
  - `#pragma once` as unofficial solution

- Problem: *includes are non-commutative*
- Possible solution: none

- Rust's *module* system does not have these issues

Main Event Loop

Main Event Loop

Three core stages:

1. **Listen for windowing system events**
   - *Block* until an event has been generated
2. **Create windowing system agnostic event abstraction**
   - *Extract* and *bundle* concrete information into abstract window manager consumable
3. **Delegate work to different parts of the program**
   - Perform window management actions based on the type of the concrete event

1. **Listen for windowing system events**
    1. Concrete `Connection`'s external dependency generates *events*
        - Input events
        - Map notification events
        - ...
    2. Convert windowing system specific event information into higher-level event abstraction
        - Decouple *windowing system event* from *window manager event*
    3. `Connection::step` method propagates event abstraction up to window manager logic
        - wmRS: `fn step(&self) -> Option<Event>;`
        - wmCPP: `Event step();`

2. Create windowing system agnostic event `enum`

- Definition of a type by *enumerating* its *variants*
- Encodes *meaning*
  - Associated integer called *discriminant*
  - Tagged union
- May attach *data*
  - Data can be *directly* associated with a variant
- Size as large as its largest variant
- Example: WMRS's `Event` enumeration:

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

2. Create windowing system agnostic event `std::variant`

- Definition of a type by *enumerating* its *alternatives*
- Type-safe tagged union class template
- Encodes *meaning*
- Contains *data*
  - Data can only *indirectly* be associated with an alternative
  - Strong type alias required for same-type alternatives
- Size as large as its largest alternative
- Example: WMCPP's `Event` type:

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

```
struct Mouse { MouseEvent event; };
struct Key { KeyEvent event; };
struct CloseRequest { Window window; };
struct ScreenChange {};
// ...
```

Event Abstraction: Remarks

WMRS:

```
fn step(&self) -> Option<Event>;
```

WMCPP:

```
Event step();
```

WMRS:

```
fn step(&self) -> Option<Event>;
```

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

WMCPP:

```
Event step();
```

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

3. Delegate work to different parts of the program

- `match` on specific *type* of event
  - Call appropriate *handler*
  - Pass encoded information to handler
- Example: WMRS's main event loop:

```
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(window),
            Event::ScreenChange
                => self.handle_screen_change(),
            // ...
        }
    }
}
```

3. **Delegate work to different parts of the program**

- Example: WMRS's main event loop:

```
while self.running {
    if let Some(event) = self.conn.step() {
        // ...
    }
}
```

- Equivalent to:

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

3. **Delegate work to different parts of the program**

- *Visit* the alternatives in `std::variant` using `std::visit`
    - *Visitor* object implementing *function-call operator* overloads
    - `std::variant` instance to visit

3. Delegate work to different parts of the program

- Example: WMCPP's visitor object:

```
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(event); }
    void operator()(ScreenChange)       { model.handle_screen_change(); }
    // ...
private:
    Model& model;
} event_visitor = EventVisitor(*this);
```

3. Delegate work to different parts of the program

- Example: WMCPP's visitor object:

```
struct EventVisitor
{
    EventVisitor(Model& model): model(model) {}
    void operator()(std::monostate)     {}
    // ...
private:
    Model& model;
} event_visitor = EventVisitor(*this);
```

- *Implicit* "**no valid event**" encoding
- Analogous to *no-op* event

3. **Delegate work to different parts of the program**

- Example: WMCPP's main event loop:

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

1. Retrieve generated event from windowing system connection
2. Rely on visitor to deduce type of event
3. Call associated handler

Event Dispatch: Remarks

- Rust and C++ both achieve desired behavior
- Visiting tagged unions in C++ is more verbose, less clear in communicating intent
- Difference worsens as more complex (or pattern-reliant) situations arise

Input Bindings

Input Bindings

- **Bind functionality to sets of peripheral input states**
  - *Mouse* bindings
  - *Keyboard* bindings
  - *Sensors*
  - ...
- **Hardware and platform dependent**
  - Initiated by the connection with the windowing system
- **Concrete input information to abstract window manager events**
  - Mouse event variant
  - Key event variant
  - ...

Input Bindings

Three-step process:

1. **Establish abstract notion of input**
    - Convert concrete input states to abstract input events
        - *Mouse* input abstractions, *keyboard* input abstractions, ...
        - *Windowing system specifics* to *window manager abstractions*
2. **Map input to window management actions**
    - Input abstractions to *closures*
3. **Retrieve and perform window management actions**

1. Establish abstract notion of input

- Example: WMRS's *mouse input* abstractions:

```
#[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,
    // ...
}
// ...
```

1. **Establish abstract notion of input**

- Example: WMRS's *mouse input* abstractions (cont.):

```
// ...
#[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>,
}
```

1. Establish abstract notion of input

- Example: WMCPP's *mouse input* abstractions:

```
enum class MouseEventKind { Press, Release, Motion };
enum class Button { Left, Middle, Right, ScrollUp, /* ... */ };
enum Modifier {
    Ctrl  = 1 << 0,
    Shift = 1 << 1,
    Super = 1 << 2,
    Alt   = 1 << 3,
    // ...
};
// ...
```

1. Establish abstract notion of input

- Example: WMCPP's *mouse input* abstractions (cont.):

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

2. **Map input to window management actions in a `HashMap`**

- Example: WMRS's *input mappings*:

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

2. Map input to window management actions in a `HashMap`

- Example: WMRS's mouse input mapping:

```
pub type MouseAction = Box<
    dyn FnMut(&mut Model<'_>, Option<Window>) -> bool
>;
// ...
```

  - Box<T>: store value of type T on the *heap*
    - Constant size: pointer to heap address
  - FnMut: closure trait that describes calling of function that mutates state
    - `dyn FnMut(...) -> ...`: *trait object* (dynamic dispatch)
    - Hooks into main window manager logic
    - Operates on clicked-on window (if any)

2. **Map input to window management actions in a `HashMap`**

- Example: WMRS's mouse input mapping:

```
#[derive(PartialEq, Eq)]
pub struct MouseInput {
    pub button: Button,
    pub modifiers: HashSet<Modifier>,
}
pub type MouseBindings = HashMap<
    MouseInput, MouseAction
>;
```

  - `MouseInput` used as *key* to `HashMap`
    - `PartialEq` and `Eq`

2. Map input to window management actions in a HashMap

- Example: WMRS's mouse input mapping:

```
#[derive(PartialEq, Eq)]
pub struct MouseInput {
    pub button: Button,
    pub modifiers: HashSet<Modifier>,
}
pub type MouseBindings = HashMap<
    MouseInput, MouseAction
>;
```

  - MouseInput used as *key* to HashMap
    - ~~PartialEq and Eq~~ (#[derive(PartialEq, Eq)])

2. **Map input to window management actions in a `HashMap`**

- Example: WMRS's mouse input mapping:

```
#[derive(PartialEq, Eq)]
pub struct MouseInput {
    pub button: Button,
    pub modifiers: HashSet<Modifier>,
}
pub type MouseBindings = HashMap<
    MouseInput, MouseAction
>;
```

  - MouseInput used as *key* to HashMap
    - ~~PartialEq and Eq~~ (#[derive(PartialEq, Eq)])
    - Hash

2. Map input to window management actions in a `HashMap`

- Example: WMRS's mouse input mapping:

```
#[derive(PartialEq, Eq)]
pub struct MouseInput {
    pub button: Button,
    pub modifiers: HashSet<Modifier>,
}
pub type MouseBindings = HashMap<
    MouseInput, MouseAction
>;
```

  - `MouseInput` used as *key* to `HashMap`
    - ~~PartialEq and Eq~~ (`#[derive(PartialEq, Eq)]`)
    - Hash (not automatically derivable)

2. Map input to window management actions in a `HashMap`

- Example: WMRS's mouse input mapping:

```
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
pub enum Button { Left, Middle, Right, ScrollUp, /* ... */ }
#[derive(PartialEq, Eq)]
pub struct MouseInput {
    pub button: Button,
    pub modifiers: HashSet<Modifier>,
}
```

  - Hash (not automatically derivable)
    - `HashSet` not automatically derivable
    - *Manual* implementation

Rust: Storing Window Management Actions

2. Map input to window management actions in a `HashMap`

- Example: WMRS's `MouseInput`'s `Hash` implementation:

```
#[derive(PartialEq, Eq)]
pub struct MouseInput {
    pub button: Button,
    pub modifiers: HashSet<Modifier>,
}
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);
    }
}
```

Rust: Storing Window Management Actions

2. Map input to window management actions in a `HashMap`

- Example: WMRS's `MouseInput`'s `Hash` implementation:

```
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);
    }
}
```

  - `<H: Hasher>`: hashing function's logic
    - *Streaming* hasher
    - *State* changes as data is being hashed
    - Final state is hashed value

2. **Map input to window management actions in a HashMap**

- Example: Registering a mouse binding in WMRS:

```
let mut mouse_bindings = HashMap::new();
mouse_bindings.insert(
    MouseInput {
        button: Button::Right,
        modifiers: {
            let mut modifiers = HashSet::with_capacity(2);
            modifiers.insert(Modifier::Ctrl);
            modifiers.insert(Modifier::Super);
            modifiers
        },
    },
    Box::new(|model: &mut Model, win: Option<Window>| -> bool {
        if let Some(window) = win {
            model.set_floating_window(window);
            true
        }
    })
);
```

2. Map input to window management actions in a std::unordered_map

- Example: wmCPP's *input mappings*:

```
typedef
    std::function<void(Model&)>
    KeyAction;
typedef
    std::function<bool(Model&, std::optional<Window>)>
    MouseAction;
typedef
    std::unordered_map<KeyInput, KeyAction>
    KeyBindings;
typedef
    std::unordered_map<MouseInput, MouseAction>
    MouseBindings;
```

2. Map input to window management actions in a `std::unordered_map`

- Example: WMCPP's mouse input mapping:

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

  - `std::function<T>`: *class template*
    - Polymorphic function wrapper
    - Callable *target*
    - C++ *type erasure*
    - Dynamic dispatch
    - *Small Buffer Optimization*

2. Map input to window management actions in a `std::unordered_map`

- Example: WMCPP's mouse input mapping:

```
struct MouseInput final {
    Button button;
    std::unordered_set<Modifier> modifiers;
};
typedef
    std::unordered_map<MouseInput, MouseAction>
    MouseBindings;
```

  - `MouseInput` used as *key* to `std::unordered_map`
    - `operator==`
    - `std::hash<MouseInput>::operator()`

2. Map input to window management actions in a `std::unordered_map`

- Example: WMCPP's `MouseInput`'s `operator==` overload:

```
struct MouseInput final {
    Button button;
    std::unordered_set<Modifier> modifiers;
};
inline bool
operator==(const MouseInput& lhs, const MouseInput& rhs)
{
    return lhs.button == rhs.button
        && lhs.modifiers == rhs.modifiers;
}
```

2. Map input to window management actions in a std::unordered_map

- Example: WMCPP's `std::hash<MouseInput>::operator()` implementation:

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

            return button_hash ^ modifiers_hash;
        }
    };
}
```

2. Map input to window management actions in a std::unordered_map

- Example: wmCPP's std::hash<MouseInput>::operator() implementation:

```
size_t
operator()(const MouseInput& input) const
{
    size_t button_hash = hash<Button>()(input.button);
    size_t modifiers_hash = hash<size_t>()(
        accumulate(input.modifiers.begin(), input.modifiers.end(),
            static_cast<Modifier>(0), bit_or<Modifier>())
    );

    return button_hash ^ modifiers_hash;
}
```

  - std::hash<T>::operator(): T's hashing function's logic
    - *Hash coding* hasher
    - Final state within function

2. Map input to window management actions in a `std::unordered_map`

- *No* automatic deriving of operator overloads
- Example: WMCPP's `Modifier`'s `operator|` implementation:

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

2. Map input to window management actions in a std::unordered_map

- Example: Registering mouse bindings in WMCPP:

```
Model(/* ... */)
  : mouse_bindings({
      { { Button::Right, { Ctrl, Super } },
          [](Model& model, std::optional<Window> window) {
              if (window)
                  model.set_fullscreen_window(*window);
          }
      },
      // more mouse bindings
  })
{ /* ... */ }
```

3. Retrieve and perform window management actions

- Example: WMRS's mouse event handler:

```
pub struct MouseEvent {
    pub kind: MouseEventKind,
    pub input: MouseInput,
    pub window: Option<Window>,
}

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

```
pub type MouseBindings = HashMap<
    MouseInput, MouseAction
>;
```

3. Retrieve and perform window management actions

- Example: WMCPP's mouse event handler:

```
struct MouseEvent final {
    MouseEventKind kind;
    MouseInput input;
    Option<Window> window;
};
```

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

```
void
Model::handle_mouse(MouseEvent& event)
{
    MouseBindings& mb = this->mouse_bindings;
    if (mb.count(event.input) > 0)
        mb.at(event.input)(*this, event.window);
}
```

Input Bindings: Remarks

- Rust and C++ both achieve desired behavior
- Rust: *streaming* hasher
  - Arbitrary amount of data
  - Byte-per-byte
  - Central hash function (`Hasher` trait)
- C++: *hash coding* hasher
  - Reduced into single integer
  - Local hash function

Input Bindings: Remarks

- Rust: operator overloading through *traits*
  - Trait bounds
- C++: non-uniform operator overloading
  - *Class methods*
  - *Non-method functions*
  - *Non-method friend functions*
  - *Concrete templated class implementations*

Input Bindings: Remarks

- Rust: *verbose* `HashMap` construction
  - *Macro metaprogramming*
    - *Declarative*: eliminate duplicate code
    - *Procedural*: operate on AST
- Example: Constructing mouse bindings in WMRS's using *declarative macros*:

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

- C++: no such feature

Clients

Clients

- **Window as main windowing system entity**
  - Application GUI
  - Integer representation
  - Unique identification
  - *Stateless*
- **Client as main window manager entity**
  - Window coupled with *state*
    - Title
    - Floating, tiled region
    - Process identifier (PID)
    - *Fullscreen?*
    - *Floating?*
    - *Iconified?*
    - ...

- Major memory safety aspect: **mutability**
  - `mut`: two *types* of semantics
    - Patterns: *changeable underlying value*
    - References: *changeable underlying value*, *aliasing* not allowed
- *aliasing* XOR *mutability*
  - Either *distribute* or *mutate*
  - Safe to mutate only if *no other* references exist
- Example: WMRS's `Client` structure:

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

Distributing references to client structures

- **Option 1**: distribute `&mut Client`
    - Client-mutating methods
    - No more than a single reference outstanding

Option 1: distribute `&mut Client`

- Clients must be centrally stored
  - Map `Window` to its `Client`
- Example: WMRS's `Client` instance storage:

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

- Mutability *bubbles up*
  - Surrounding structures *also* mutable
    - *Inherited, exterior mutability*
    - Mutating single client field: `&mut Client`
    - Mutating `&mut Client`: `fn ...(&mut self, ...)`

Option 1: distribute `&mut Client`

- Example: mutating client state in WMRS:

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

Option 1: distribute `&mut Client`

- Example: mutating client state in WMRS:

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

```
error[E0502]: cannot borrow `*self` as immutable
because it is also borrowed as mutable

if let Some(client) = self.client_map.get_mut(&win) {
                      --------------- mutable borrow occurs here
    self.apply_layout(client.workspace());
    ^^^^              ------ mutable borrow later used here
    |
    immutable borrow occurs here
```

Distributing references to client structures

- ~~**Option 1**: distribute `&mut Client`~~
  - ~~Client-mutating methods~~
  - ~~No more than a single reference outstanding~~
- **Option 2**: distribute `&Client`
  - *More* than a single reference may be outstanding

Option 2: distribute `&Client`

- Shared *mutable* references with `Cell` and `RefCell`
- Example: WMRS's `Client` structure:

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

  - `Cell<...>` and `RefCell<...>`: *mutable* fields
    - *Interior mutability* constructs
  - Other fields: `Client` constants

Interior mutability: `Cell` and `RefCell`

- `Cell`: move or copy values *in* and *out*
  - *No* reference to contained value
- `RefCell`: references instead of values
  - Acquire a *lock* before mutating
  - *Dynamic borrowing*
- *Move* mutability to individual fields
  - *Interior* borrow checking
  - Mutate non-exclusive reference
  - *Mutate* **and** *alias*

Option 2: distribute `&Client`

- Example: mutating client state in WMRS:

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

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

- **All** mutability is interior mutability
    - *Every* reference can be aliased
    - *No* restriction on mutability of references
- Example: WMCPP's `Client` structure:

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

- Example: WMCPP's `Client` structure:

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

  - `Client` *pointers* instead of window representations
    - Main event loop invariant: window destruction gets reported
    - No need for *smart pointers*
    - We know *exactly* when to deallocate memory
  - Other fields: `Client` constants

Distributing pointers to client structures

- Heap-allocated `Client` structures are centrally stored
- Example: WMCPP's `Client` instance storage:

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

Distributing pointers to client structures

- Example: mutating client state in WMCPP:

```
void
Model::set_fullscreen_window(Window window)
{
    if (client_map.count(window) > 0) {
        Client_ptr client = client_map.at(window);
        client->fullscreen = true;
        apply_layout(client->workspace);
    }
}
```

Clients: Remarks

- Rust: borrow checker
  - More restrictive than strictly necessary
    - Language simplicity
    - Compilation speed
    - Need for interior mutability
  - Adds significant programmer effort
  - Renders programming patterns impossible
    - Self-referential structures
    - Adds runtime overhead

Results

Results

- **Compared C++ and Rust in their pragmatics**
  - *How* each language achieves common objectives
  - The implementation of a window manager
- **Rust**
  - Standard package manager and build system
  - Guarantees to provide memory safety and prevent data races
  - *Concise* and *eloquent* syntax
  - Expressively powerful
    - Pattern matching
    - Tagged union enumerations
    - Derivable traits
    - Macros

Results

- **Rust (cont.)**
  - Consistent
    - Operator overloading *only* through traits
  - Error mitigation
    - More programming effort
    - Runtime cost
    - Difficult to work with references
    - Programming patterns impossible
    - Interior mutability

Results

- **C++**
  - Lack of standardized package manager and build system
  - Leaky include system
  - No safety guarantees
  - Complex syntax
  - Inconsistencies
    - Operator overloads
  - Easy to work with references, pointers

Results

- **Rust**
  - Learnt how *not* to do it
  - Borrows from other languages
  - Restrictive but safe
  - Simple yet expressive
- **C++**
  - Lacks Rust's *hindsight*
  - Much more complex
  - Unrestrictive but unprotective
  - Decades of revisions
  - Initial core design goals