

# **Pragmatics of Rust and C++:**

## **The implementation of a window manager**

---

Max van Deurzen

June 19, 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

### 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

---

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 interface 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
  - Implement the interface to target the *X Window System*
  - Implement the interface to target *Wayland*
  - Implement the interface to target the *Desktop Window Manager* (Windows)
  - Implement the interface to target the *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 prototype
- 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
  - Contains 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:

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

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

```
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 xconn and conn trait objects:

```
use x11rb::connection;

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

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

## Rust traits vs C++ abstract classes

- Abstract classes: *inheritance*
  1. Describe common behavior
  2. Code reuse
  3. Polymorphism
- Traits: *implementation*
  1. Describe common behavior
  2. Code reuse with **generics**: *abstraction* over different types
  3. Polymorphism with **trait bounds**: *constraints* on these type abstractions

# 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 variant
- Example: WMRS's Event enumeration:

```
typedef std::variant<                                struct Mouse { MouseEvent event; };
    std::monostate,                                struct Key { KeyEvent event; };
    Mouse,                                         struct CloseRequest { Window window; };
    Key,                                           struct ScreenChange {};
    CloseRequest,                                // ...
    ScreenChange,
    // ...
> Event;
```

# Input Bindings

---

## Second Frame

Hello, world!

# Clients

---

## Second Frame

Hello, world!