

Department I - C Plus Plus

Modern and Lucid C++ Advanced  
for Professional Programmers

Week 14 – Rust and C++

Thomas Corbat / Felix Morgner  
Rapperswil, 06.06.2022  
FS2022



- **Recap Week 13**
- **Intro to Rust**
- **Rust-like Features in C++**

- **Participants should ...**
  - ... know how Rust provides increased lifetime safety
  - ... know how to apply similar safety features using the Clang C++ compiler

Recap Week 13



- **Five libraries**
  - All depending on a common infrastructure library
- **Two executables**
  - Depend on some or all of the libraries
- **Two target-platforms**
  - Linux on x86\_64 and armv7
  - OS X
- **Code will change owners**
- **4 months time-frame**

- **Write a script that...**

- Compiles each source file
- Links all object files together
- Repeats that for every target

- **DON'T! Because...**

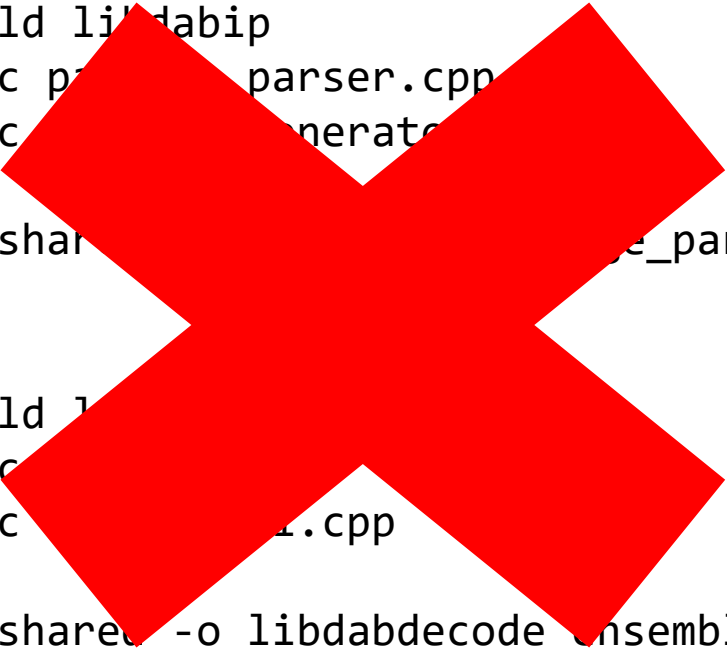
- ... every source file get built every time!
- ... the commands tend to be platform specific
- ... build order must be managed manually
- ... scripts tend to become messy over time

```
#!/bin/bash

# Build libdabip
gcc -c parser.cpp
gcc -c generator.cpp
...
gcc -shared -o libdabip.o parser.o ...

# Build 1
gcc -c 1.cpp
gcc -c 2.cpp
...
gcc -shared -o libdabdecode ensemble.o ...

# Build <you get the idea>
```



- **Make-style Build Tools**

- Run build scripts
- Produce your final products
- Often verbose
- Use a language agnostic configuration language

- **Build Script Generators**

- Generate configurations for Make-style Build Systems or Build Scripts
- Configuration independent of actual build tool
- Advanced features (download dependencies, etc.)

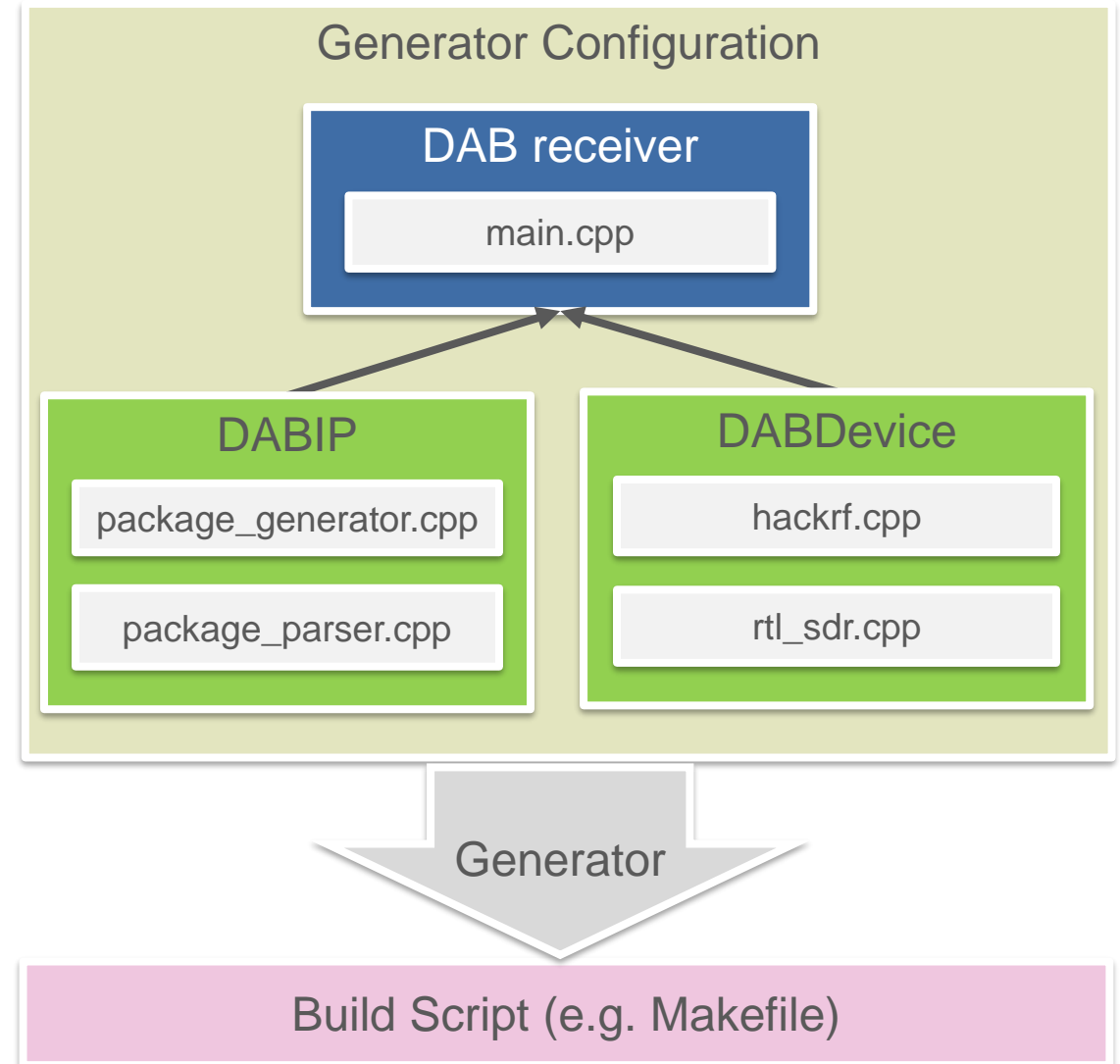
- **Idea: Take a step back**

- Define what we want to achieve, not how to do it
- Work on a higher level
- Let the create the actual build configurations

- **Platform independent build specification**

- **Tool Independent**

- Often can generate IDE projects
- Support multiple build tools





- **CMake includes CTest**

- Enable CTest using `enable_testing()`
- Create a “Test Runner” executable
  - Make sure to include your suite sources!

- Configure build environment:

```
$ cmake ..
```

- Build the project:

```
$ cmake --build .
```

- Run ctest

```
$ ctest --output-on-failure
```

```
cmake_minimum_required(VERSION "3.12.0")

project("answer" LANGUAGES CXX)

enable_testing()

add_library("${PROJECT_NAME}"
  "answer.cpp"
)

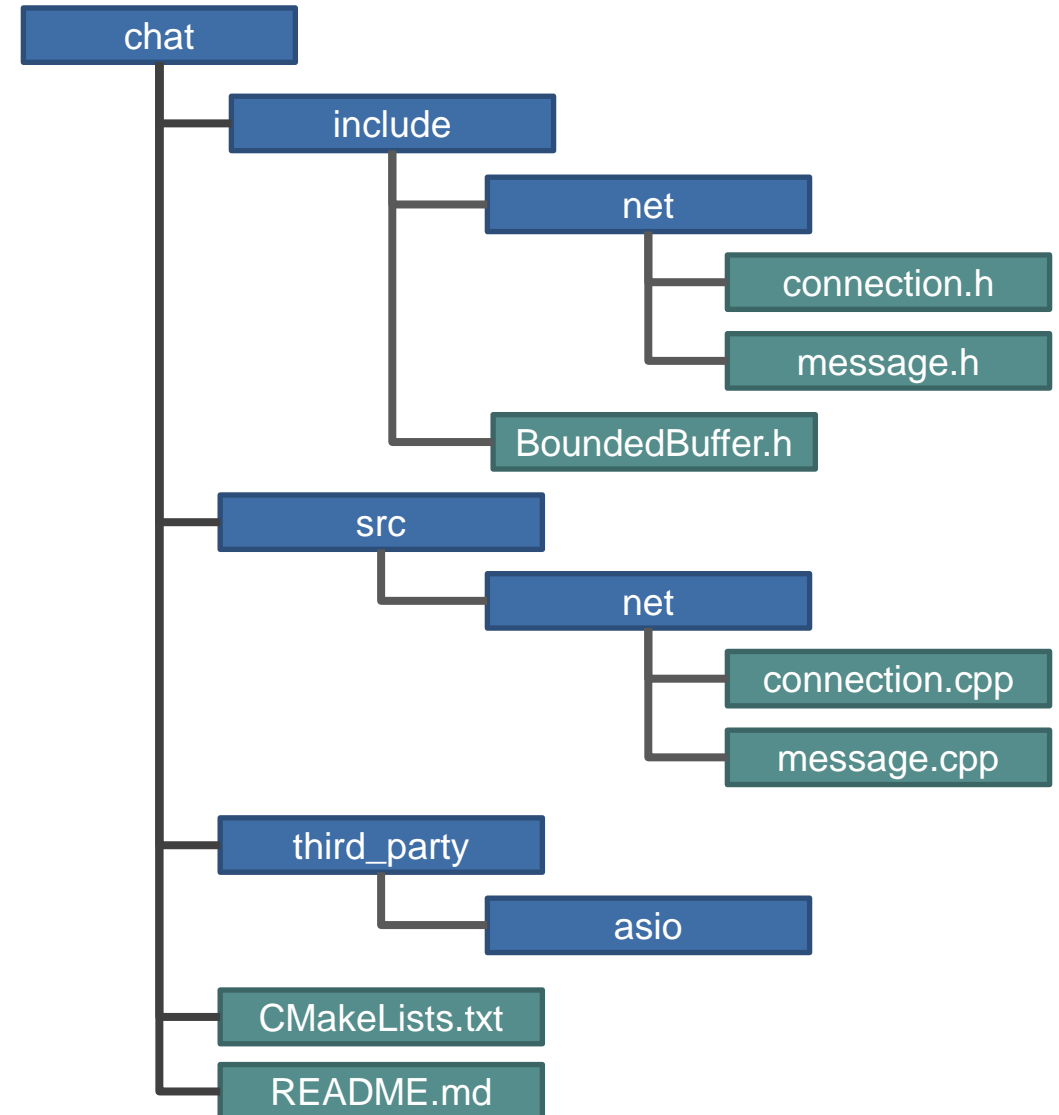
add_executable("test_runner"
  "Test.cpp"
)

target_link_libraries("test_runner" PRIVATE
  "answer"
)

target_include_directories("test_runner" SYSTEM PRIVATE
  "cute"
)

add_test("tests" "test_runner")
```

- **Headers live in the “include” folder**
  - Add subfolders for separate subsystems if needed
- **Implementation files live in the “src” folder**
  - Make sure that subfolder layout matches the “include” folder (**consistency**)
- **Put third-party projects/sources in a “third\_party” or “lib” folder**
- **Test resource live in the “test” folder**
  - The test folder will have src, include, and third\_party subfolders if required
- **Build configuration files should live in the root of your project**



- **Libraries may benefit from a slightly different layout**

- You will need to ship your headers
- Your headers might have very generic names

- **Idea: Introduce another nesting level for your headers**

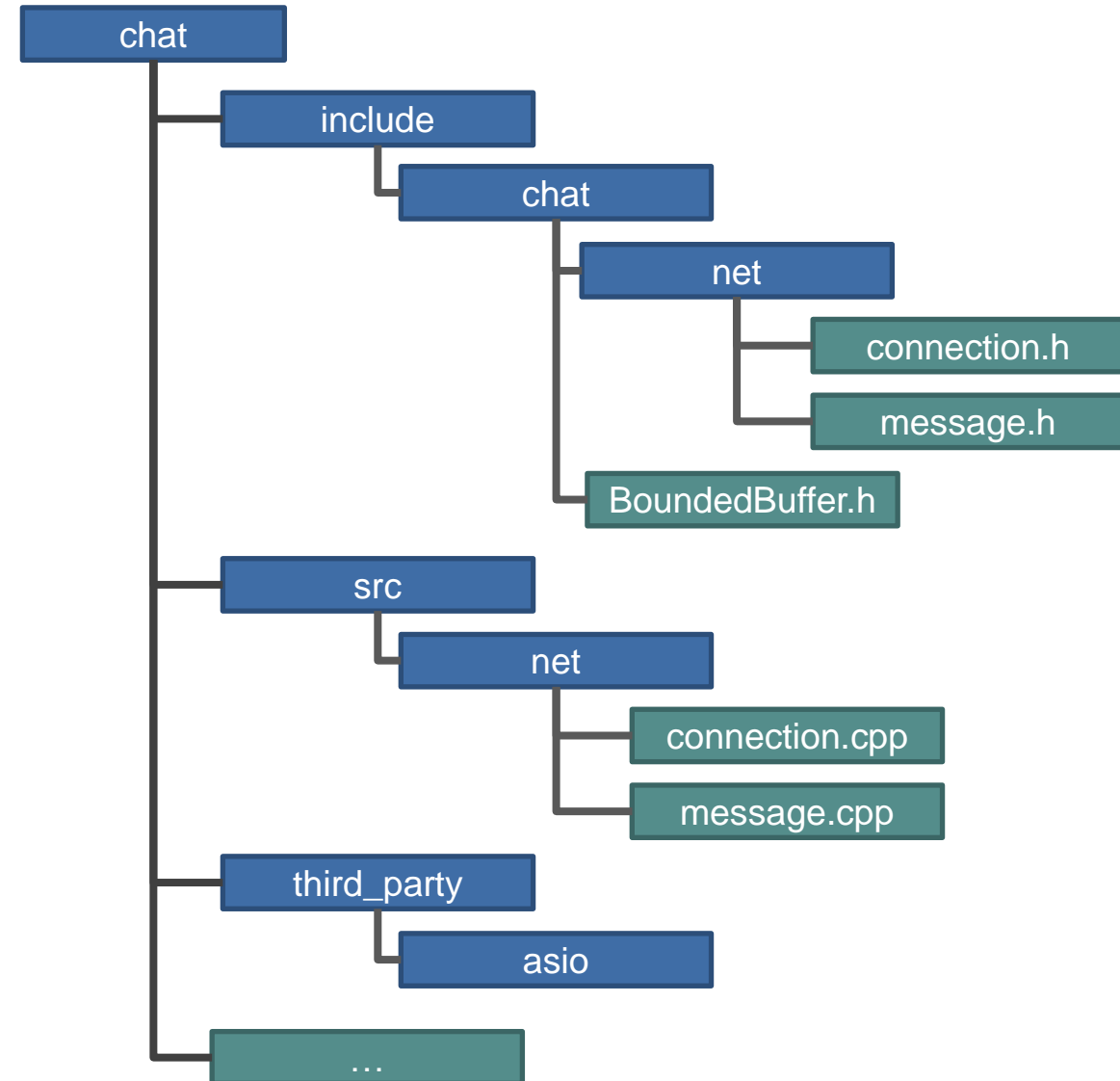
- Use the name of your project

```
#include "message.h"
```

... becomes ...

```
#include "chat/message.h"
```

- Helps avoid filename clashes



# Intro to Rust



- **Multiplayer Game**

- Focused on Survival Gameplay
- First Released in December 2012
- Inspired by S.T.A.L.K.E.R

- **Implementation**

- Built on top of Unity
- Most likely C#

- **Online-only**



- **Multi-paradigm Programming Language**

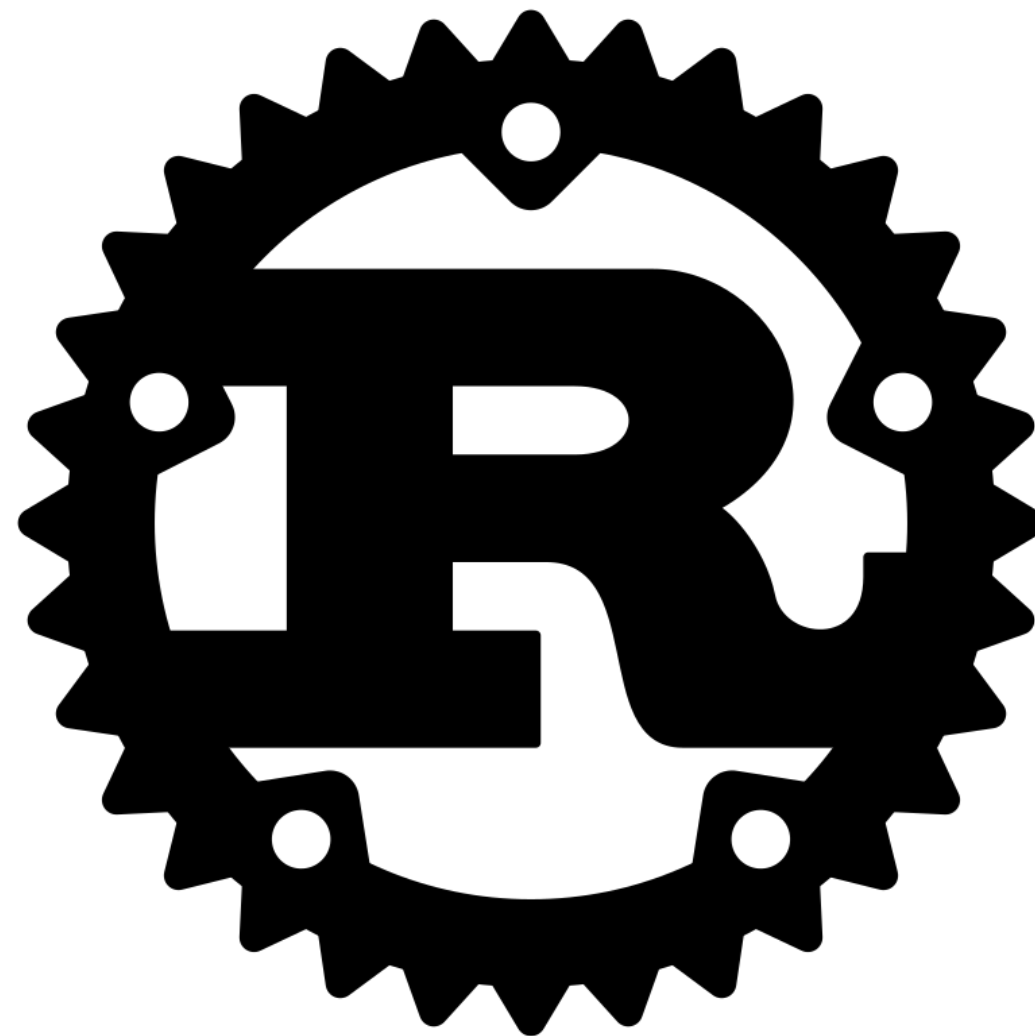
- Functional
- Imperative
- Generic
- ...

- **Strong focus on safety**

- Ownership
- Memory Safety

- **Designed by Mozilla in 2010**

- Now “The Rust Foundation”



- **Rustc**

- Compiler
- Built on top of LLVM

- **Cargo**

- Quick-start for projects
- Build system and package manager
- <https://crates.io> as primary repository

- **Rustfmt**

- Formatter

Live Coding Demo





- **Every Object has an Owner**
  - Ownership can be Transferred
  - Scope is Bound to Owner
- **Ownership has *reader-writer-lock* semantics**
  - Multiple immutable references
  - Only one mutable reference
- **Value semantics still apply when ...**
  - ... copying the object
  - ... moving the object

```
struct Number {  
    value: u32  
}  
  
fn consume(num: Number) -> u32 {  
    num.value  
}  
  
pub fn main() {  
    let num = Number{ value: 32 };  
    consume(num);  
    println!("object.value == {0}", num.value);  
}
```

Live Coding Demo



```
struct Number {  
    value: u32  
}  
  
fn consume(num: Number) -> u32 {  
    num.value  
}  
  
pub fn main() {  
    let num = Number{ value: 32 };  
    consume(num);  
    println!("object.value == {0}", num.value);  
}
```

**error[E0382]: borrow of moved value: `num`**

--> <source>:13:37

```
11 | let num = Number{ value: 32 };  
    | --- move occurs because `num` has type `Number`, which does not implement the `Copy` trait  
12 | consume(num);  
    | --- value moved here  
13 | println!("object.value == {0}", num.value);  
    | ^^^^^^^^^^^ value borrowed here after move
```

Live Coding Demo



```
#[derive(Clone, Copy)]
struct Number {
    value: u32
}

fn consume(num: Number) -> u32 {
    num.value
}

pub fn main() {
    let num = Number{ value: 32 };
    consume(num);
    println!("object.value == {0}", num.value);
}
```

```
struct Number {
    value: u32
}

fn consume(num: & Number) -> u32 {
    num.value
}

pub fn main() {
    let num = Number{ value: 32 };
    consume(&num);
    println!("object.value == {0}", num.value);
}
```

- **Every Object has an Owner**
  - The Compiler Enforces Validity
  - Moved-from values cannot be used anymore
- **Primary use: Multithreading!**
  - Move values into closure
  - All Accesses After Move are Invalid!
  - No data-race possible!

Live Coding Demo





```
use std::thread;

pub fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("{:?}", v);
    });

    handle.join().unwrap();
}
```

**error[E0373]: closure may outlive the current function, but it borrows `v`, which is owned by the current function**

--> <source>:6:32

```
6 | let handle = thread::spawn(|| {
  |   ^^ may outlive borrowed value `v`
7 | println!("{:?}", v);
  | - `v` is borrowed here
```

```
use std::thread;

pub fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("{:?}", v);
    });

    handle.join().unwrap();
}
```

## Rust-like Features in C++



- **Compilers Support Extensions**

- “Playground” for Future Language Features
- Support for Library Specific Features
- GNU C++ Dialect

- **Clang has Rust-like Ownership Extensions**

- Specific Warning Category
  - -Wconsumed

- **Classes Must be Marked Using consumable Attribute**

- `[[clang::consumable(...)]]`

- **Functions and Parameters Are Marked with Specific Attributes**

- `[[clang::callable_when(...)]]`

- `[[clang::return_typestate(...)]]`

- `[[clang::set_typestate(...)]]`

- `[[clang::param_typestate(...)]]`

- **Typestates:**

- `unconsumed, consumed, unknown`

```
struct Number {  
    unsigned value() const {  
        return m_value;  
    }  
  
    unsigned m_value;  
};  
  
auto main() -> int {  
    auto num = Number{32};  
  
    auto fut = std::async(std::launch::async, [num = std::move(num)]{  
        std::cout << num.value() << '\n';  
    });  
  
    std::cout << num.value() << '\n';  
}
```

Live Coding Demo



```
struct [[clang::consumable(unconsumed)]] Number
{
    [[clang::callable_when(unconsumed)]]
    unsigned value() const {
        return m_value;
    }

    unsigned m_value;
};
```

```
<source>:50:22: error: invalid invocation of method 'value' on object 'num' while it is in
the 'consumed' state [-Werror,-Wconsumed]
std::cout << num.value() << '\n';
```



Live Coding Demo



```
auto main() -> int {  
    auto num = Number{32};  
  
    auto fut = std::async(std::launch::async, [num = std::move(num)]{  
        std::cout << num.value() << '\n';  
    });  
  
    num = Number{42};  
  
    std::cout << num.value() << '\n';  
}
```

- **Rust has a powerful mechanism to prevent invalid use of objects**
  - There can only ever be one owner for an object
  - Together with memory safety features makes making mistakes very hard
- **Efforts in the Clang C++ compiler to try and prevent similar guarantees in C++**
  - Currently no effort of standardizing this mechanism
  - Must see the body of the functions