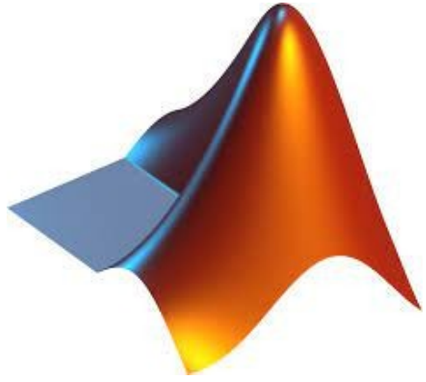
Numerical computing in Rust

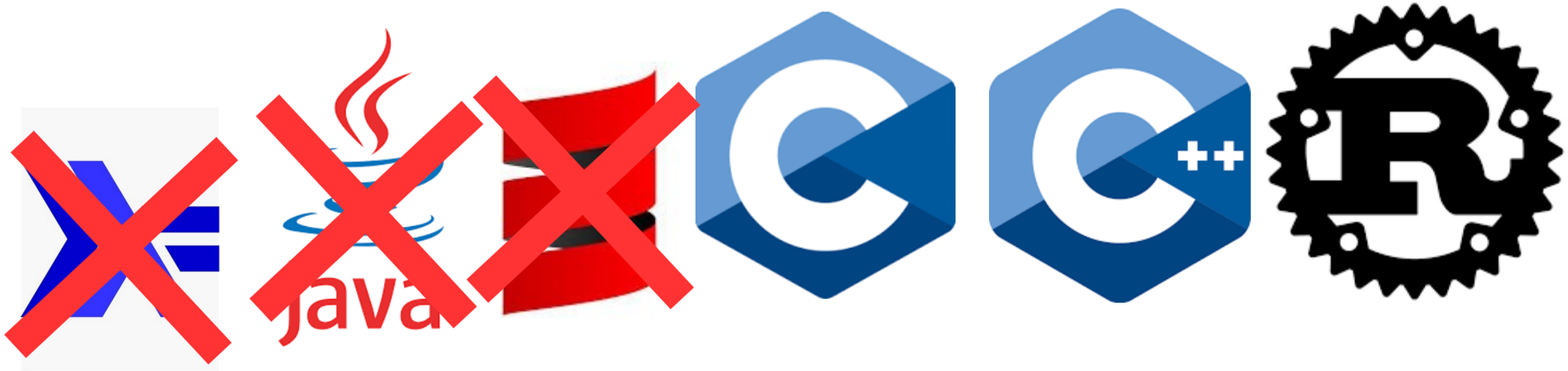
Hadrien Grasland 2024-11-25

My ideal compute language?

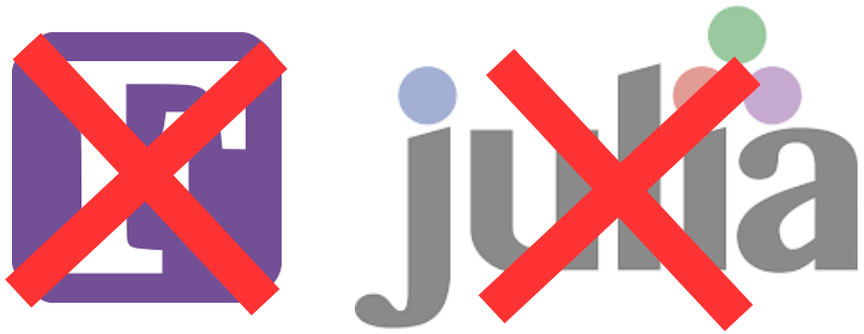


General-purpose

Easy to optimize

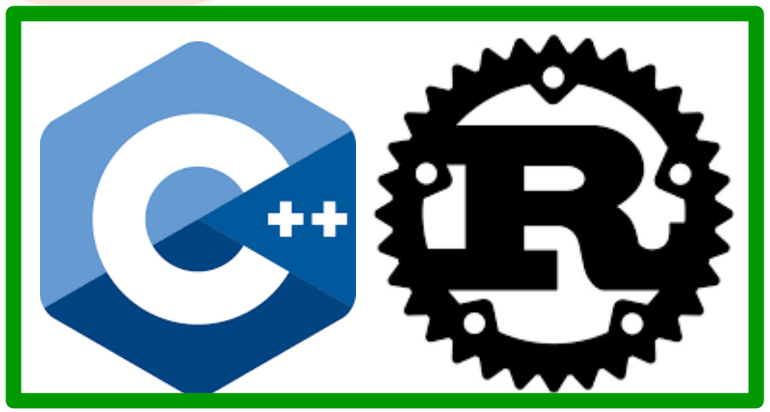


Well-equipped for larger projects*



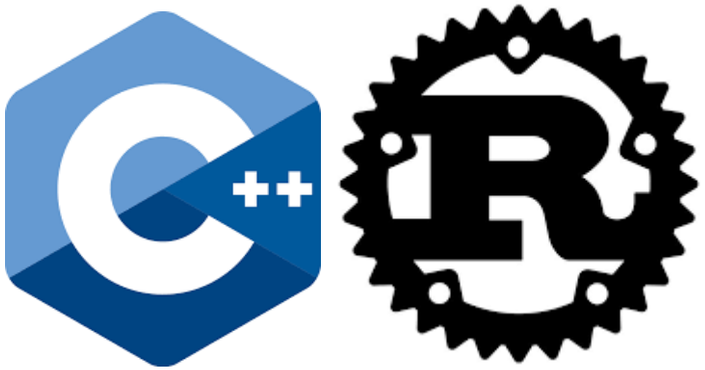
* When we start appreciating encapsulation, generics, polymorphism, code generation, a hierarchized API…

Not that much of a choice!



Many common ideas

- (Normally) AoT compiled
- No mandatory GC
- Strict/explicit typing
- Low-level control
- Metaprogramming
- Rich & zero-cost abstractions
- Price to pay: takes a while to master



- **So, what does Rust do differently?**

Undefined behavior (UB) in C++

- The optimizer assumes there is none → Unpredictable effect
- **Arithmetics:** Signed int overflow, shift > bits, -INT_MIN, casts…
- **Arrays:** Out of bounds accesses, iterator invalidation…
- **Pointers/references:** Null, misaligned, invalid, strict aliasing…
- **Uninitialized memory:** Merely reading its value is UB (beware destructors, assignment, exceptions…)
- **Infinite loops** may violate Fermat's last theorem
- **Multi-threading:** Concurent access to data being written
- Many more → Unavoidable in real-world code…

Consequence: Security problems*

- Share of memory safety vulnerabilities in C/++ projects:
  - 65% in Android (90% of media & bluetooth vulns)
  - 65% in the Linux kernel (according to Ubuntu)
  - 66% in iOS, 72% in macOS
  - 70% in Google Chrome
  - 70% in Microsoft products
  - 74% in Firefox's CSS engine
- …and that's just one kind of undefined behavior!

* For sure, your compute code may not be exposed to attackers today.
But are you 100% sure no one will ever try to build a web visualization on top of it?

Rust's answer: A safety pledge

- Outside of unsafe blocks*, Rust compiler proves UB-**safety**:
  - **Type:** Values will honor type invariants (e.g. str is UTF-8)
  - **Memory:** References will point to valid, initialized memory
  - **Thread:** Writes to shared data will be synchronized
- A **good tradeoff** in practice
  - Need unsafe: compile-time proof may be impossible, frequent run-time checks may become expensive
  - BUT can do most work without it
  - Used rarely & localized → Easier to audit than C/++

* Unsafe code benefits from the normal proofs, but can also use un-proven primitives.

C++ type system weaknesses

- **Unexpected behavior and incomprehensible errors**
  often caused by interactions between…
  - Implicit conversions (inc. non-explicit constructors)
  - Function overloading + default arguments
  - Templates + spécializations thereof
  - Virtual methods + inheritance
- Templates extra hard to write due to **minimal type checking***
  - Instantiation errors feel much like Python/JS runtime errors

* I know about C++20 concepts, you'll see how they fail to solve the problem in a few slides.

A simple C++ program

```cpp
#include <algorithm>
#include <cstddef>
#include <concepts>
#include <iostream>
#include <vector>

template<std::floating_point T>
T median(const std::vector<T>& input) { /* ... */ }

int main()
{
    std::cout << median(std::vector<float>{ 1.2, 3.4, 5.6 }) << std::endl;
}
```

Imagine that's from a third-party library

Helpful compiler output

```
      |      ^~~~~~~~
/usr/include/c++/13/ostream:694:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   cannot convert ‘input’ (type ‘const std::vector<float>’) to type ‘const char8_t*’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~
/usr/include/c++/13/ostream:699:5: note: candidate: ‘template<class _Traits> std::basic_ostream<char, _Traits>& std::operator<<(basic_ostream<char, _Traits>&, const char16_t*)’ (deleted)
  699 |      operator<<(basic_ostream<char, _Traits>&, const char16_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:699:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   cannot convert ‘input’ (type ‘const std::vector<float>’) to type ‘const char16_t*’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~
/usr/include/c++/13/ostream:703:5: note: candidate: ‘template<class _Traits> std::basic_ostream<char, _Traits>& std::operator<<(basic_ostream<char, _Traits>&, const char32_t*)’ (deleted)
  703 |      operator<<(basic_ostream<char, _Traits>&, const char32_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:703:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   cannot convert ‘input’ (type ‘const std::vector<float>’) to type ‘const char32_t*’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~
/usr/include/c++/13/ostream:709:5: note: candidate: ‘template<class _Traits> std::basic_ostream<wchar_t, _Traits>& std::operator<<(basic_ostream<wchar_t, _Traits>&, const char8_t*)’ (deleted)
  709 |      operator<<(basic_ostream<wchar_t, _Traits>&, const char8_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:709:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   mismatched types ‘wchar_t’ and ‘char’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~
/usr/include/c++/13/ostream:714:5: note: candidate: ‘template<class _Traits> std::basic_ostream<wchar_t, _Traits>& std::operator<<(basic_ostream<wchar_t, _Traits>&, const char16_t*)’ (deleted)
  714 |      operator<<(basic_ostream<wchar_t, _Traits>&, const char16_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:714:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   mismatched types ‘wchar_t’ and ‘char’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~
/usr/include/c++/13/ostream:718:5: note: candidate: ‘template<class _Traits> std::basic_ostream<wchar_t, _Traits>& std::operator<<(basic_ostream<wchar_t, _Traits>&, const char32_t*)’ (deleted)
  718 |      operator<<(basic_ostream<wchar_t, _Traits>&, const char32_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:718:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   mismatched types ‘wchar_t’ and ‘char’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~
/usr/include/c++/13/ostream:801:5: note: candidate: ‘template<class _Ostream, class _Tp> _Ostream&& std::operator<<(_Ostream&&, const _Tp&)’
  801 |      operator<<(_Ostream&& __os, const _Tp& __x)
      |      ^~~~~~~~
/usr/include/c++/13/ostream:801:5: note:   template argument deduction/substitution failed:
/usr/include/c++/13/ostream: In substitution of ‘template<class _Ostream, class _Tp> _Ostream&& std::operator<<(_Ostream&&, const _Tp&) [with _Ostream = std::basic_ostream<char>&; _Tp = std::vector<float>]’:
concept.cpp:9:50:   required from ‘T median(const std::vector<T>&) [with T = float]’
concept.cpp:22:24:   required from here
/usr/include/c++/13/ostream:801:5: error: template constraint failure for ‘template<class _Os, class _Tp>  requires (__derived_from_ios_base<_Os>) && requires(_Os& __os, const _Tp& __t) {__os << __t;} using std::__rvalue_stream_insertion_t = _Os&&’
/usr/include/c++/13/ostream:801:5: note: constraints not satisfied
/usr/include/c++/13/ostream: In substitution of ‘template<class _Os, class _Tp>  requires (__derived_from_ios_base<_Os>) && requires(_Os& __os, const _Tp& __t) {__os << __t;} using std::__rvalue_stream_insertion_t = _Os&& [with _Os = std::basic_ostream<char>&; _Tp = std::vector<float>]’:
/usr/include/c++/13/ostream:801:5:   required by substitution of ‘template<class _Ostream, class _Tp> _Ostream&& std::operator<<(_Ostream&&, const _Tp&) [with _Ostream = std::basic_ostream<char>&; _Tp = std::vector<float>]’
concept.cpp:9:50:   required from ‘T median(const std::vector<T>&) [with T = float]’
concept.cpp:22:24:   required from here
/usr/include/c++/13/ostream:768:13:   required for the satisfaction of ‘__derived_from_ios_base<_Os>’ [with _Os = std::basic_ostream<char, std::char_traits<char> >&]
/usr/include/c++/13/ostream:768:39: note: the expression ‘is_class_v<_Tp> [with _Tp = std::basic_ostream<char, std::char_traits<char> >&]’ evaluated to ‘false’
  768 |      concept __derived_from_ios_base = is_class_v<_Tp>
      |                                        ^~~~~~~~~~~~~~~
hadrien@silent-graloufotron:~/Bureau/concept>
```

Find the problem

```
 7 template<std::floating_point T>
 8 T median(const std::vector<T>& input) {
 9     std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
10     std::vector<T> sorted = input;
11     std::sort(sorted.begin(), sorted.end());
12     std::size_t midpoint = sorted.size() / 2;
13     if (sorted.size() % 2 == 0) {
14         return (sorted[midpoint] + sorted[midpoint + 1]) / 2.0;
15     } else {
16         return sorted[midpoint];
17     }
18 }
```

Find all the problems

C++20 concepts don't reliably prevent instantiation errors
(only work if manually kept in sync with implementation)

Illegal in C++
(no alternative)

```
 7 template<std::floating_point T>
 8 T median(const std::vector<T>& input) {
 9     std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
10     std::vector<T> sorted = input;
11     std::sort(sorted.begin(), sorted.end());
12     std::size_t midpoint = sorted.size() / 2;
13     if (sorted.size() % 2 == 0) {
14         return (sorted[midpoint] + sorted[midpoint + 1]) / 2.0;
15     } else {
16         return sorted[midpoint];
17     }
18 }
```

UB if input.size() == 0

float → double → float round trip if T is float

Dubious result if there is a NaN in « input »
(UB likely with less careful third party sort)

Find all the problems

I've been writing C++ for **~20 years**

The first code I write remains full of these « little gotchas »

Only a tiny fraction is detected by usual compiler lints (-Wall -Wextra)

It takes hours of proofreading, testing… to get to a correct result for all inputs

Rust's answer: Stronger typing

- All Rust polymorphism comes from **constrained generics**:
  - Types can implement traits, e.g. operator overloads
  - Generic code must tell what traits it needs in its API
  - Using ~anything else causes a clear compiler error*
- Consequence: Rust is a lot more **predictable**
  - No conversion/overloading/template/virtual/… interactions
  - Generics fail early and clearly, neither at instantiation time nor deep inside of the implementation

* This is the check that C++20 concepts lack, likely for backcompat with old templates. Thus any change to generic C++ code may silently invalidate its concept API contract…

Let's translate my code to Rust

```rust
use num_traits::Float;

fn median<T: Float>(input: &Vec<T>) -> T {
    println!("DEBUG: Finding the median of {input}...");
    let mut sorted = input.clone();
    sorted.sort_unstable();
    let midpoint = sorted.len() / 2;
    if sorted.len() % 2 == 0 {
        (sorted[midpoint] + sorted[midpoint + 1]) / 2.0
    } else {
        sorted[midpoint]
    }
}

fn main() {
    println!("{}", median(&vec![1.2, 3.4, 5.6]));
}
```

Compiler reports 3 errors

```
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs> cargo check
    Checking concept-rs v0.1.0 (/home/hadrien/Bureau/concept/concept-rs)
error[E0277]: `Vec<T>` doesn't implement `std::fmt::Display`
 --> src/main.rs:4:44
  |
4 |     println!("DEBUG: Finding the median of {input}...");
  |                                            ^^^^^^^ `Vec<T>` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `Vec<T>`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)

error[E0277]: the trait bound `T: Ord` is not satisfied
    --> src/main.rs:6:12
     |
6    |     sorted.sort_unstable();
     |            ^^^^^^^^^^^^^ the trait `Ord` is not implemented for `T`
     |
note: required by a bound in `core::slice::<impl [T]>::sort_unstable`
    --> /home/hadrien/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/slice/mod.rs:2951:12
     |
2949 |     pub fn sort_unstable(&mut self)
     |            ------------- required by a bound in this associated function
2950 |     where
2951 |         T: Ord,
     |            ^^^ required by this bound in `core::slice::<impl [T]>::sort_unstable`
help: consider further restricting this bound
     |
3    | fn median<T: Float + std::cmp::Ord>(input: &Vec<T>) -> T {
     |                    +++++++++++++++

error[E0308]: mismatched types
 --> src/main.rs:9:53
  |
3 | fn median<T: Float>(input: &Vec<T>) -> T {
  |           - expected this type parameter
...
9 |         (sorted[midpoint] + sorted[midpoint + 1]) / 2.0
  |         ----------------------------------------    ^^^ expected type parameter `T`, found floating-point number
  |         |
  |         expected because this is `T`
  |
  = note: expected type parameter `T`
                     found type `{float}`

Some errors have detailed explanations: E0277, E0308.
For more information about an error, try `rustc --explain E0277`.
error: could not compile `concept-rs` (bin "concept-rs") due to 3 previous errors
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs>
```

Error 1: Can’t display Vec<T>

```
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs> cargo check
    Checking concept-rs v0.1.0 (/home/hadrien/Bureau/concept/concept-rs)
error[E0277]: `Vec<T>` doesn't implement `std::fmt::Display`
 --> src/main.rs:4:44
  |
4 |     println!("DEBUG: Finding the median of {input}...");
  |                                            ^^^^^^^ `Vec<T>` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `Vec<T>`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `println`
```

- Problem found **even if generic code not instantiated**
- Suggests an alternative: the Debug output

Error 2: Can’t sort floats

```
error[E0277]: the trait bound `T: Ord` is not satisfied
    --> src/main.rs:6:12
     |
6    |     sorted.sort_unstable();
     |            ^^^^^^^^^^^^^^ the trait `Ord` is not implemented for `T`
     |
note: required by a bound in `core::slice::<impl [T]>::sort_unstable`
    --> /home/hadrien/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/slice/mod.rs
     |
2949 |     pub fn sort_unstable(&mut self)
     |            ------------- required by a bound in this associated function
2950 |     where
2951 |         T: Ord,
     |            ^^^ required by this bound in `core::slice::<impl [T]>::sort_unstable`
help: consider further restricting this bound
     |
3    | fn median<T: Float + std::cmp::Ord>(input: &Vec<T>) -> T {
     |                    +++++++++++++++
```

- Not allowed to sort floats by default: **NaN is not ordered**
  - Can assert absence of NaNs in various ways

Error 3: Can’t divide by a literal

```
error[E0308]: mismatched types
 --> src/main.rs:9:53
  |
3 | fn median<T: Float>(input: &Vec<T>) -> T {
  |           - expected this type parameter
...
9 |         (sorted[midpoint] + sorted[midpoint + 1]) / 2.0
  |         ----------------------------------------   ^^^ expected type parameter `T`, found floating-point number
  |         |
  |         expected because this is `T`
  |
  = note: expected type parameter `T`
                     found type `{float}`

Some errors have detailed explanations: E0277, E0308.
For more information about an error, try `rustc --explain E0277`.
error: could not compile `concept-rs` (bin "concept-rs") due to 3 previous errors
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs>
```

- Float literals untyped, no implicit conversion to arbitrary T
- End of the error message points to more detailed explanations

One error remains undetected*

- Rust version still **wrong if input Vec is empty**
- This will cause a deterministic crash (panic) at runtime
  - No undefined behavior, unlike in C++
- Is this a good error handling strategy?
  - Yes if an empty input is considered to be a user error
    - …but then it should be spelled out in documentation!
  - Otherwise, should return **Option<T>**: Some(T) or None

* Alas, Rust will not save you from writing tests. It will only prevent many test *failures*.

C++ error reporting

- Historically bet everything on **exceptions**
  - Very expensive to throw and catch
  - Hard to write code that’s correct when it happens
  - Discouraged in destructors, but no alternative provided
- Don’t want exceptions? **Welcome to the jungle.**
  - Special return values or « int » that no one checks, as in C
  - Exotic return types specific to each individual project
  - Error case documentation usually incomplete

Rust's answer: A clear strategy

- For recoverable errors, use enumerated type* **Result<T, E>**
  - Contains either valid output Ok(T) or error description Err(E)
  - To get to the output, must specify how errors are handled
- For program bugs (e.g. failed assertions), use **panics**
  - Configurable: unwind (like C++ exceptions) or abort
  - Catching unwinding panics is allowed, but rare/discouraged
- Strong **community consensus** + error documentation culture

* Similar to C++17's std::variant, but with an API that normal people would want to use.

Code generation

- In C++, often done via **template metaprogramming**
  - Error checking bug (SFINAE) that became a key C++ feature
  - Leveraged through unmaintainable expert-only code
  - Very inefficient → Build becomes slow, RAM-hungry
  - Alternatives? Preprocessor macros, parse compiler's debug outputs, add a code generator like ROOT to the build…
- In Rust: Traits/generics, build scripts, or **lexical macros**
  - Operates on AST-like token tree provided by the compiler
  - Much better ergonomics/expressivity tradeoff

Example: serde

- Macro-based generation of ~universal (de)serialization code

```
1 use serde::{Deserialize, Serialize};
2
3 #[derive(Debug, Serialize, Deserialize)]
4 struct Record {
5     idx: u32,
6     data: f64,
7     comment: String,
8 }
```

Doesn't need to be in std

Enables JSON, CSV, Pickle… (de)serialization

Language-provided macro for debug output

Build and dependencies

- In C/++, you get **CMake** and the Linux distribution zoo
  - If you don't hate these yet, you've not faced them enough
  - Outcome: Code reuse aversion → **Wheel reinvention**
- Rust's answer: **cargo** included in standard toolchain*
  - Dual purpose: build system + package manager
  - Easy for small projects, scales well to much larger ones
  - Dependencies are now easy → **Lively libs ecosystem**

* Which also features a bunch of other basic tools: doc generator, unit test harness…

C++ is drowning in its past

- Practicioners (even young) **rarely trained on new revisions**
- **Compilers don't keep up**. Especially on RHEL, embedded…
- **Ghosts from C/++ past** keep influencing C++ future
  - Preprocessor, copy-and-paste macros, includes*
  - Ill-defined primitive types like long (size?) and char (sign?)
  - Typed literals → 42ULL and 1.2f insanity
  - Wrong defaults: switch fallthrough, copy semantics, const
  - Implicit conversions, vector<bool>, numeric_limits::min…

* C++20 modules tried to fix it… with a design so flawed that module mapper madness ensued.
Even without that, not all headers will be rewritten → people must know/deal with both forever.

Rust's answer: Define C++'s future

- **C++17** as seen from **Rust v1 (2015)**
  - **Trying hard to catch up…**
    filesystem, any, string_view, byte, aligned_alloc
  - **…but some copies are pretty defective**
    optional, std::tuple, *structured bindings*, CTAD, std::variant
- **C++20** on a similar trajectory
  - **More decent copies:** <=>, consteval, { .a }, format, span, endian, <bit>, barrier, latch, jthread, assume_aligned
  - **More failed copies:** Ranges, coroutines, modules, concepts

Conclusion

- 2 good reasons to start a C++ projet in 2024
  - Part of a **larger C++ project** that you should not rewrite
  - **C++ libraries** and tools more mature for your problem
- In any other situation, **consider Rust instead**
  - Language now mature enough, rarely the limiting factor
  - Superior ergonomics → Less bugs, more features
  - Easier to learn + better overall support than C++2x

Thanks for your attention !