Zero-Cost Abstractions in C++20

Vincent Reverdy
[vincent.reverdy@lapp.in2p3.fr]
Researcher in Computer Science and Numerical Cosmology
CNRS - French National Centre for Scientific Research
LAPP - Laboratoire d'Annecy de Physique des Particules

July 11th, 2023

Table of contents

Code complexity

Why do we want better and better supercomputers?

Keep the same physics

- Better resolution
- Better accuracy
- Better statistics

Keep the same resolution

- Improve physical modeling
- Multiphysics

Usual code limitations

Runtime performance

- Limitations on execution time or energy consumption

Memory

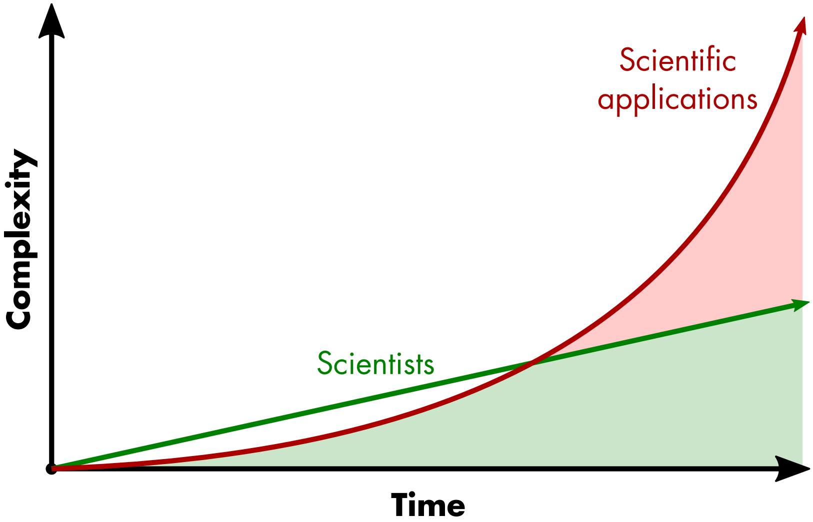
- Limitations on memory usage

The most problematic code limitation

Structural code complexity

- Codes only exist if humans can write them in the first place

The wall of software complexity



Corollary

Side-effect for the perfect software

If code complexity grows faster than the availability of better CPU and memory... $\Rightarrow$ one can design code as if computational resources were infinite

In practice

If you expect your code to be in full production in 10 years, design it with the computational resources available at that time in mind, as well as the availability of better compiler optimizations.

What was special about this game?

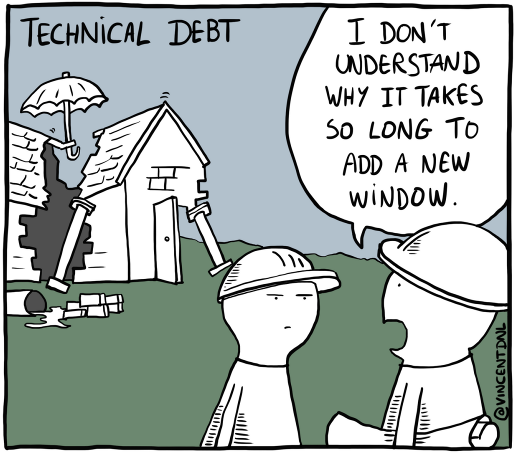
The role of abstraction

Complexity reduction

$$\mathcal{C} = \mathcal{O}\left(\prod_i \alpha_i\right) \quad \Rightarrow \quad \mathcal{C} = \mathcal{O}\left(\sum_i \alpha_i\right)$$

- $\mathcal{C}$: structural complexity
- i: concept
- α_i: number of instances of that concept

The critical role of software architecture



Software Architecture

Starting from an example

A navigation code used to actually fly airplanes

```
void xXY_Brg_Rng(double X_1, double Y_1, double X_2, double Y_2, double *Bearing, double *Range);

void DistanceBearing(double lat1, double lon1,
                     double lat2, double lon2,
                     double *Distance, double *Bearing);

double DoubleDistance(double lat1, double lon1,
                      double lat2, double lon2,
                      double lat3, double lon3);

void FindLatitudeLongitude(double Lat, double Lon,
                           double Bearing, double Distance,
                           double *lat_out, double *lon_out);

double CrossTrackError(double lon1, double lat1,
                       double lon2, double lat2,
                       double lon3, double lat3,
                       double *lon4, double *lat4);

double ProjectedDistance(double lon1, double lat1,
                         double lon2, double lat2,
                         double lon3, double lat3,
                         double *xtd, double *crs);

void LatLon2Flat(double lon, double lat, int *scx, int *scy);
```

A few guiding principles

Small functions

- Write small functions when possible (less than 30 lines)

Few parameters

- Try to minimize the number of parameters (less than 4 most of the time)

Keep the same pattern

- Keep the same pattern of parameters for similar functions

Type everything!

- Encode as much information as possible in types

Bikeshedding

- Finding good names for things is hard, but critical

Core idea

Structure matters

- Encode application domain information in the structure of the program itself

Aside note

- Programs can be seen as mathematical structures on which mathematical metrics can be computed (eg: the abstract shape or the topology of a program)

Software stack

<table>
<tr><td colspan="4">Applications</td></tr>
<tr><td colspan="4">High level libraries</td></tr>
<tr><td>Wrappers and bindings</td><td>Python</td><td>R</td><td>Java</td></tr>
<tr><td>Optimized libraries</td><td colspan="2">Interpreters (Python, R...)</td><td>Virtual machines (JVM)</td></tr>
<tr><td colspan="4">Compiled, native, low level languages (C, C++...)</td></tr>
<tr><td colspan="4">Compilers, mostly written in C and C++ (GCC, LLVM...)</td></tr>
<tr><td colspan="4">Machine layer, assembly instructions</td></tr>
</table>

Conceptual propagation

Conceptual approximations

- Conceptual approximations propagate from the bottom up and gets amplified

Data structures are key

- Data structures generally sit at the bottom
- Worth spending time on it

The GPE Principle

The Holy Triad

- Genericity
- Performance
- Expressivity

Genericity: Optimize for the library's author lines of code

- How many special cases can I cover with my code?

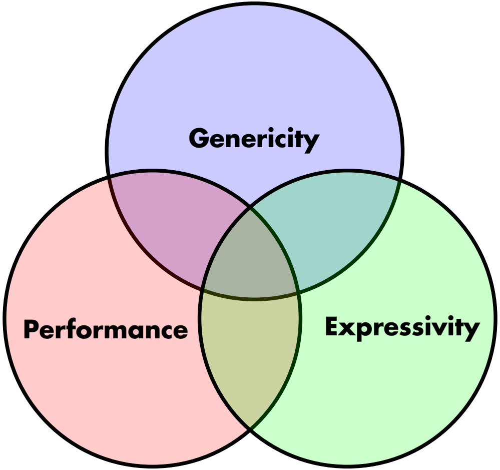
Performance: Optimize for runtime

- How fast my code is?

Expressivity: Optimize for the library's user lines of code

- How much can I express in a single line of code?

The GPE Principle



Zero-cost abstractions

Everything at the same time

- C++ does not make you choose between genericity, performance, and expressivity
- High-levels of abstraction are compatible with high-levels of performance

Warning

- Possible does not necessarily means easy to write

Concept-based programming

Software Architecture in C++20

Concept and constraints

- Concepts as constrained generic programming
- A way to formalize and specify abstractions and software architecture in C++

Concepts and constraints in C++20

Concept

- Named set of requirements
- Must appear at namespace scope

```
template </*template-parameter-list*/>
concept /*concept-name*/ = /*constraint-expression*/;
```

Constraints

- Sequence of logical operations and operands
- Requirements on template arguments
- 3 types: conjunctions / disjunctions / atomic constraints

Example of concepts

A simple arithmetic concept

```
// Concept definition
template <class T>
concept arithmetic = std::is_arithmetic_v<T>;

// Constrained function v1
template <arithmetic T>
void print_v1(T x) {
    std::cout << x << std::endl;
}

// Constrained function v2
template <class T>
requires arithmetic<T>
void print_v2(T x) {
    std::cout << x << std::endl;
}

// Constrained function v3
void print_v3(arithmetic auto x) {
    std::cout << x << std::endl;
}
```

Example of constraints

Constraints on addability

```
template <class T>
concept addable = requires {std::declval<T>() + std::declval<T>();};

template <class T>
concept addable = requires (T x) {x + x;};

template <class T1, class T2>
concept addable2 = requires (T1 x, T2 y) {x + y;};

template <class T>
concept addable_and_multiplicable = addable<T> && requires (T x) {x * x;};

template <class T>
requires requires (T x) {x + x;}
T add(T x, T y) {
    return x + y;
}
```

Subsumption of concepts (1/3)

Subsumption as partial ordering

```
template <class T>
concept addable = requires (T x) {x + x;};

template <class T>
concept shiftable = requires (T x) {x << 1;};

template <class T>
concept addable_and_shiftable = addable<T> && shiftable<T>;

template <addable T>
void f(T x) {std::cout << "only addable" << std::endl;}

template <addable_and_shiftable T>
void f(T x) {std::cout << "addable and shiftable" << std::endl;}

f(5.1); // only addable
f(5); // addable and shiftable
```

Subsumption of concepts (2/3)

Works only with concepts

```cpp
template <class T, class = void>
struct is_addable: std::false_type {};
template <class T>
struct is_addable<T, std::void_t<decltype(
    std::declval<T>() + std::declval<T>()
)>>: std::true_type {};
template <class T>
inline constexpr bool is_addable_v = is_addable<T>::value;

template <class T, class = void>
struct is_shiftable: std::false_type {};
template <class T>
struct is_shiftable<T, std::void_t<decltype(
    std::declval<T>() << std::declval<std::size_t>()
)>>: std::true_type {};
template <class T>
inline constexpr bool is_shiftable_v = is_shiftable<T>::value;
```

Subsumption of concepts (3/3)

Works only with concepts

```
template <class T>
concept addable = is_addable_v<T>;

template <class T>
concept shiftable = is_shiftable_v<T>;

template <class T>
concept addable_and_shiftable = is_addable_v<T> && is_shiftable_v<T>;

template <addable T>
void f(T x) {std::cout << "only addable" << std::endl;}

template <addable_and_shiftable T>
void f(T x) {std::cout << "addable and shiftable" << std::endl;}

f(5.1); // ambiguous
f(5); // ambiguous
```

C++ concepts vs Rust traits

C++ concepts

- Structural typing
- A type may accidentally satisfy a concept
- No coupling between concepts (architecture) and types (implementation)
- Concepts are optional
- Constraints work on allowed expression for the whole language
- Subsumption and logical expressions of constraints (`&&`, `||`, `!`)

Rust traits

- Nominal typing
- `impl Trait for Type` explicitly indicates that a type satisfy a trait
- Coupling between traits (architecture) and types (implementation)
- Traits are mandatory
- Traits can only check for a subset of the language

C++ concepts can do nominal typing

Nominal typing implementation

```cpp
template <class Trait>
struct implements_trait {};

template <class T, class Trait, class = void>
struct is_implementing: std::false_type {};

template <class T, class Trait>
struct is_implementing<T, Trait, std::enable_if_t<
    std::is_base_of_v<implements_trait<Trait>, T>
>>: std::true_type {};

template <class T, class Trait>
concept implements = is_implementing<T, Trait>::value;

struct mytrait {};

struct mytype: implements_trait<mytrait> {};

template <implements<mytrait> T>
void f(T x) {}

f(mytype{}); // OK
f(3); // ERROR
```

Combining concepts and contraints with if constexpr

With an external concept

```
template <class T>
concept shiftable = requires {std::declval<T>() << std::declval<int>();};

template <class T>
void is_shiftable() {
    if constexpr (shiftable<T>) {std::cout << "shiftable" << std::endl;}
    else {std::cout << "not shiftable" << std::endl;}
}

// is_shiftable<int>() —> "shiftable"
// is_shiftable<double>() —> "not shiftable"
```

With an inline **requires** clause

```
template <class T>
void is_shiftable() {
    if constexpr (requires {std::declval<T>() << std::declval<int>();}) {std::cout << "shiftable" << std::endl;}
    else {std::cout << "not shiftable" << std::endl;}
}
```

With an inline **requires** clause with a parameter list

```
template <class T>
void is_shiftable() {
    if constexpr (requires (T x, int y) {x << y;}) {std::cout << "shiftable" << std::endl;}
    else {std::cout << "not shiftable" << std::endl;}
}
```

Checking if a function exists

The traditional way: the preprocessor

```
#ifdef __SUPPORTS_THEFUNCTION
/* Doing something here */
#endif
```

The metaprogramming way

```
//void thefunction(int x);

template <class T, class = decltype(thefunction(std::declval<T>()))>
std::true_type supports_thefunction_for(T);
template <class T, class... X>
std::false_type supports_thefunction_for(T, X...);

inline constexpr bool supports_thefunction
= decltype(supports_thefunction_for(std::declval<int>()))::value;
// true if thefunction(int x) is active
// false if thefunction(int x) is commented out
```

The concept-based way

```
//void thefunction(int x);

template <class T = int>
concept supports_thefunction_for
= requires (T x) {thefunction(x);};
```

Checking if a function exists: forcing template dependency

Leveraging alias templates

```cpp
//void thefunction(int x);

// The concept checks for a particular type provided by the user
template <class T>
concept supports_thefunction_for
= requires (T x) {thefunction(x);};

// Alias template keeping only the first type
template <class T, class...>
using first_type = T;

// The concept ignores its template parameter and tests only the relevant type
template <class... Dummy>
concept supports_thefunction
= requires {thefunction(std::declval<first_type<int, Dummy...>>());};
```

Constexpr if and requires clauses

The problem of undefined symbols

```
/*
void thefunction(int) {
    std::cout << "thefunction" << std::endl;
}*/

template <class T>
void check(T x) {
    if constexpr (requires (T y) {thefunction(y);}) {
        thefunction(x); // OK
        thefunction(3); // ERROR
        []<class U>(U x){thefunction(x);}(3); // OK
    } else {
        std::cout << "not thefunction" << std::endl;
    }
}

check(1); //
```

Coming back to the problem of printing

Better than std::enable_if

```
// For numbers
template <printable T>
void print(const T& x) {
    std::cout << x << std::endl;
}

// For container of numbers
template <range R>
requires printable<decltype(*std::begin(std::declval<R>()))>
void print(const R& range) {
    for (auto it = std::begin(container); it != std::end(container); ++it) {
        std::cout << *it << " ";
    } std::cout << std::endl;
}
```

Software architecture with concepts

Solves the problems of metaprogramming-based approaches

- Easy to read
- Easy to implement
- Nice error messages

Contrast with Object Oriented Programming

- Types are not stuck in a fixed hierarchy
- Types come first, abstractions second
- No runtime overhead, pure compile-time check

Important notes

- A way to guide the compiler in the compilation process
- Bottom-up approach
- Designing concepts can be crazy hard

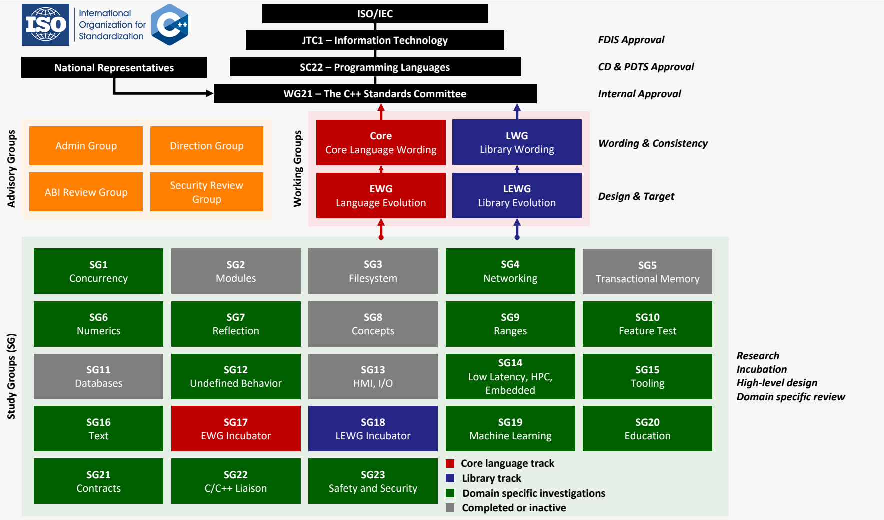
Standardization

The C++ Standard

The standard

- Link: N4950
- Only a specification, not an implementation

The C++ Standards Committee



Going beyond

Why is it so complicated to standardize anything?

- Backward compatibility
- No ABI break
- Bad past decisions
- Insane levels of requirements
- Achieving Genericity, Performance, and Expressivity at the same time

Better than the standard library

- Still possible to do better than the standard and the standard library
- Implementers are not doing black magic and do not have infinite resources

An improved tuple: the overloaded log-tuple trick

Straightforward approach

`get<N>(tuple)` has to iterate over the first `N` types.

Advanced approach

There is a way to exploit overload resolution to have $\mathcal{O}\left(\log\left(N\right)\right)$ compile-time access.

Indexing

```cpp
// Index constant type
template <std::size_t I>
struct index_constant: std::integral_constant<std::size_t, I> {};

// Index constant variable template
template <std::size_t I>
inline constexpr index_constant<I> index = {};
```

Log-tuple trick: elements

Element wrappers

```
// A basic element wrapper
template <class T>
struct tuple_element_wrapper {
    constexpr tuple_element_wrapper(const T& x): value(x) {}
    // Other constructors to be defined
    T value;
};

// An indexed tuple element
template <std::size_t I, class T>
struct tuple_element: tuple_element_wrapper<T> {
    constexpr tuple_element(const T& x): tuple_element_wrapper<T>(x) {}
    constexpr T& operator[](index_constant<I>) {
        return static_cast<wrapper<T>&>(*this).value;
    }
    constexpr const T& operator[](index_constant<I>) const {
        return static_cast<const wrapper<T>&>(*this).value;
    }
};
```

Log-tuple trick: tuple

Tuple

```cpp
// Base class declaration
template <class Sequence, class... T>
struct tuple_base;

// Base class specialization for index sequence
template <std::size_t... I, class... T>
struct tuple_base<std::index_sequence<I...>, T...>
: tuple_element<I, T>... {
    using index_sequence = std::index_sequence<I...>;
    using tuple_element<I, T>::operator[]...;
    constexpr tuple_base(const T&... x): tuple_element<I, T>(x)... {}
    // Other constructors to be defined
};

// Actual tuple implementation
template <class... T>
struct tuple: tuple_base<std::index_sequence_for<T...>, T...> {
    using base = tuple_base<std::index_sequence_for<T...>, T...>;
    using base::base;
    using base::operator[];
};
template <class... T>
tuple(const T&...) -> tuple<T...>;
```

Result

`mytuple[index<3>]` leverages overload resolution to access the element at compile-time.

Taking C++ to another level of genericity

Genericity in C++

- C++ is type-generic but NOT kind-generic

C++ is type-generic

```
template <class T>
struct wrapper {};

template <>
struct wrapper<int> {};

template <>
struct wrapper<double> {};
```

C++ is NOT kind-generic

```
template <class T>
struct wrapper1 {};

template <auto X>
struct wrapper2 {};

template <template <class...> class F>
struct wrapper3 {};
```

Problem with higher-order metafunctions

The problem

```
// Metafunction hierarchy
template <class T>
struct metafunction_wrapper_0 {};
template <template <class...> class F>
struct metafunction_wrapper_1 {};
template <template <template <class...> class...> class F>
struct metafunction_wrapper_2 {};
template <template <template <template <class...> class...> class...> class F>
struct metafunction_wrapper_3 {};
template <template <template <template <template <class...> class......> class...> class...> class F>
struct metafunction_wrapper_4 {};

// Use cases
metafunction_wrapper_1<metafunction_wrapper_0> x1; // OK
metafunction_wrapper_2<metafunction_wrapper_1> x2; // OK
metafunction_wrapper_3<metafunction_wrapper_2> x3; // OK
metafunction_wrapper_4<metafunction_wrapper_3> x4; // OK
```

Proposal

- Currently no way of collapsing the hierarchy
- Introducing a new mechanism to make C++ kind-generic

Advanced C++

Symbolic calculus in C++

Unique identifiers for symbols

```
template <class T>
struct symbol_id {
    static constexpr auto singleton = []{};
};
```

Symbol definition

```
template <class T = void, auto Id = symbol_id<decltype([]{})>{}>
struct symbol {
    static constexpr auto symbol_id = Id;
};
```

Unique symbols

Symbol definition

```
template <class T = void, auto Id = symbol_id<decltype([]{})>{}>
struct symbol {
    static constexpr auto symbol_id = Id;
};
```

In practice

```
int main() {
    symbol x;
    symbol y;
    std::cout << std::is_same_v<decltype(x), decltype(y)> << std::endl; // 0
}
```

Need a total ordering on symbol identifiers

Modifying the definition of the symbol identifiers

```
template <class T>
struct symbol_id {
    static constexpr auto singleton = []{};
    static constexpr const void* address = static_cast<const void*>(&singleton);
};

template <class Lhs, class Rhs>
constexpr std::strong_ordering operator<=>(symbol_id<Lhs>, symbol_id<Rhs>) {
    // Using the standard function object that defines a total order on pointers
    return std::compare_three_way{}(
        symbol_id<Lhs>::address,
        symbol_id<Rhs>::address
    );
}

int main() {
    symbol x;
    symbol y;
    std::cout << (x.id < y.id)= << std::endl;
}
```

Introducing some concepts

Specializable type traits

```
template <class>
struct is_symbolic: std::false_type {};

template <class T, auto Id>
struct is_symbolic<symbol<T, Id>>: std::true_type {};

template <class T>
inline constexpr bool is_symbolic_v = is_symbolic<T>::value;
```

Symbolic concept

```
template <class T>
concept symbolic = is_symbolic_v<T>;
```

Symbolic operators

Assignment

```
struct assignment_operator {
    template <class Rhs, class Lhs>
    constexpr decltype(std::declval<Rhs>() = std::declval<Lhs>())
    operator()(Rhs&& rhs, Lhs&& lhs)
    noexcept(noexcept(std::forward<Rhs>(rhs) = std::forward<Lhs>(lhs))) {
        return std::forward<Rhs>(rhs) = std::forward<Lhs>(lhs);
    }
};
```

Addition

```
struct addition_operator {
    template <class Rhs, class Lhs>
    constexpr decltype(std::declval<Rhs>() + std::declval<Lhs>())
    operator()(Rhs&& rhs, Lhs&& lhs)
    noexcept(noexcept(std::forward<Rhs>(rhs) + std::forward<Lhs>(lhs))) {
        return std::forward<Rhs>(rhs) + std::forward<Lhs>(lhs);
    }
};
```

Symbolic expressions

Expressions

```
template <class... Args>
struct symbolic_expression {
};

template <class... Args>
struct is_symbolic<symbolic_expression<Args...>>: std::true_type {};

template <symbolic Lhs, symbolic Rhs>
constexpr symbolic_expression<decltype(
    []()-> std::tuple<operator_symbol<assignment_operator>, Lhs, Rhs>{return {};}
)>
operator+(Lhs, Rhs) noexcept {
    return {};
}
```

Expression templates are not dead

Application

```cpp
int main(int argc, char* argv[]) {
    symbol x;
    symbol y;
    symbol z;
    auto f = x + y + z; // Contains the AST
}
```

Full symbolic language with AST manipulation

Basic application

```
int main(int argc, char* argv[]) {
    // Real symbols
    symbol<real> a;
    symbol<real> b;
    symbol<real> c;
    symbol<real> d;

    // Symbolic function
    auto f = (a + b) * (c + d);

    // Computation
    f(a = 5., b = 13., c = 50., d = 12.)
}
```

For linear algebra

With matrices

```
int main(int argc, char* argv[]) {
    // Real symbols
    symbol<matrix<real>> a;
    symbol<matrix<real>> b;
    symbol<matrix<real>> c;

    // Symbolic function
    auto f = (a + b) * c;

    // Computation
    f(
        a = std:mdspan(...),
        b = std:mdspan(...),
        c = std:mdspan(...)
    );
}
```

Going beyond

A full symbolic language

- AST manipulation (simplification, . . .)
- Solving equations
- Expressing parallelism
- Custom optimizer

Take-home lesson

Expressivity

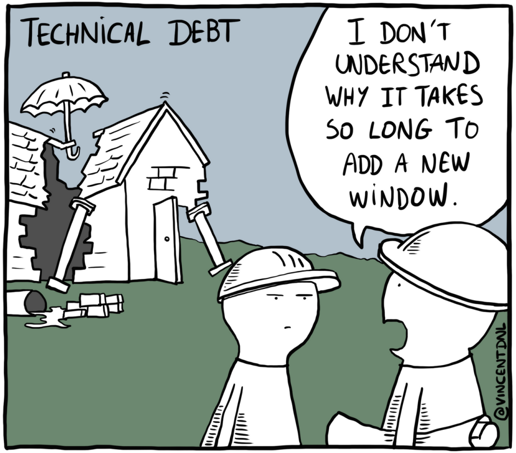
Start from what users would like to write

Another example

```
std::ndarray<double, shape[4]()[3][5]> myarray;
```

Conclusions

Conclusion



Architecture is important. Please abstract things. Thanks.

Thank you for your attention

Any question?