

Concept-based programming for HPC

Vincent Reverdy

April 7th, 2022



Table of contents

1 Introduction

2 Metaprogramming

3 High Performance Software Architecture

4 Concepts

Introduction

1 Introduction

2 Metaprogramming

3 High Performance Software Architecture

4 Concepts

The common problem of generic code

Handling types, once at a time: cannot possibly scale

```
1 // A character
2 void print(char x) {std::cout << x << std::endl;}
3
4 // An integer
5 void print(int x) {std::cout << x << std::endl;}
6
7 // A floating-point number
8 void print(float x) {std::cout << x << std::endl;}
9
10 // A vector of integers
11 void print(const std::vector<int>& v) {
12     for (std::size_t i = 0; i < v.size(); ++i) {
13         std::cout << v[i] << " ";
14     } std::cout << std::endl;
15 }
16
17 // A vector of floating-point numbers
18 void print(const std::vector<float>& v) {
19     for (std::size_t i = 0; i < v.size(); ++i) {
20         std::cout << v[i] << " ";
21     } std::cout << std::endl;
22 }
23
24 // A list of integers
25 void print(const std::list<int>& l) {
26     for (auto it = l.begin(); it != l.end(); ++it) {
27         std::cout << *it << " ";
28     } std::cout << std::endl;
29 }
```

Using generics to handle the problem: templates in C++

For fundamental types

```
1 // Characters, integers, floating-point numbers...
2 template <class T>
3 void print(T x) {
4     std::cout << x << std::endl;
5 }
```

For vectors

```
1 // Vectors of characters, of integers, of floating-point numbers
2 template <class T>
3 void print(const std::vector<T>& v) {
4     for (std::size_t i = 0; i < v.size(); ++i) {
5         std::cout << v[i] << " ";
6     } std::cout << std::endl;
7 }
```

For lists

```
1 // Lists of characters, of integers, of floating-point numbers
2 template <class T>
3 void print(const std::list<T>& l) {
4     for (auto it = l.begin(); it != l.end(); ++it) {
5         std::cout << *it << " ";
6     } std::cout << std::endl;
7 }
```

Generalizing for containers

For fundamental types

```
1 // Characters, integers, floating-point numbers...
2 template <class T>
3 void print(T x) {
4     std::cout << x << std::endl;
5 }
```

For containers

```
1 // Containers of characters, of integers, of floating-point numbers
2 template <template <class> class C, class T>
3 void print(const C<T>& c) {
4     for (auto it = c.begin(); it != c.end(); ++it) {
5         std::cout << *it << " ";
6     } std::cout << std::endl;
7 }
```

Yay! That works: everything is perfect, full of rainbows and unicorns

```
1 int main(int argc, char* argv[]) {
2     std::vector<int> v = {0, 1, 2, 3, 4};
3     std::list<double> l = {0.1, 1.2, 2.3, 3.4, 4.5};
4     print(1);
5     print(3.14);
6     print(v);
7     print(l);
8 }
```

What could possibly go wrong?!?

Introducing a simple type

```
1 struct type {};
2
3 int main(int argc, char* argv[]) {
4     print(type{});
5     print(std::vector<int>{});
6     return 0;
7 }
```

Congratulations you just broke the Universe

GCC is so happy that it outputs 810 lines of warnings, errors, and comments

What did actually go wrong?

- The templated functions accepted nearly everything
- What makes the content printable was never actually checked
- The pattern C<T> did not provide any guarantee that the type was a container

Let's use Object-Oriented Programming!

First and foremost

No. Just no.

The problems with Object-Oriented Programming

- Enforce a mental model where everything has to be a hierarchy
- Once a hierarchy has been set, it's almost fixed forever
- Moving away from the hierarchy ⇒ multiplication of abstract classes
- ... and encourage runtime overhead through virtual calls

Important note

- Object-Oriented Programming mostly relies on a **top-down** approach
- Abstraction comes first
- Concrete objects are forced into fixed hierarchies

Metaprogramming

1 Introduction

2 Metaprogramming

3 High Performance Software Architecture

4 Concepts

Facing the original problem

Goals

- Keep generic interfaces
- Guide the user through the right function
- Guide the compiler through the right function (right = functionality, optimization...)

Using SFINAE: Substitution Failure is not an Error

```
1 // For numbers
2 template <class T, class = std::enable_if_t<std::is_arithmetic_v<T>>
3 void print(const T& x) {
4     std::cout << x << std::endl;
5 }
6
7 // For container of numbers
8 template <class Container, class = std::enable_if_t<std::is_arithmetic_v<
9     typename std::iterator_traits<typename Container::const_iterator>::value_type
10 >>>
11 void print(const Container& container) {
12     for (auto it = std::begin(container); it != std::end(container); ++it) {
13         std::cout << *it << " ";
14     } std::cout << std::endl;
15 }
```

More problems: signatures

Signature conflict

- `template <class T, class> void print(const T&)`
- `template <class Container, class> void print(const Container&)`

Mixing std::enable_if with Non-Type Template Parameters (NTTPs)

```
1 // For numbers
2 template <class T, bool = std::enable_if_t<std::is_arithmetic_v<T>, bool>{}>
3 void print(const T& x) {
4     std::cout << x << std::endl;
5 }
6
7 // For container of numbers
8 template <class Container, int = std::enable_if_t<std::is_arithmetic_v<
9     typename std::iterator_traits<typename Container::const_iterator>::value_type
10 >, int>{}>
11 void print(const Container& container) {
12     for (auto it = std::begin(container); it != std::end(container); ++it) {
13         std::cout << *it << " ";
14     } std::cout << std::endl;
15 }
```

More problems: ordering

What if a type T satisfies

- None of the `enable_if` ⇒ OK, clean error code, no function found
- One of the `enable_if` ⇒ OK, the right function is chosen
- Several `enable_if` ⇒ AMBIGUITY

Using variadic templates to create an artificial ordering (different signatures for free)

```
1 // For numbers
2 template <class T, class = std::enable_if_t<std::is_arithmetic_v<T>>>
3 void print(const T& x) {
4     std::cout << x << std::endl;
5 }
6
7 // For container of numbers
8 template <
9     class Container,
10    class... Dummy,
11    class = std::enable_if_t<std::is_arithmetic_v<
12        typename std::iterator_traits<typename Container::const_iterator
13        >::value_type>>,
14    class = std::enable_if_t<sizeof...(Dummy) == 0>
15 >
16 void print(const Container& container, Dummy...) {
17     for (auto it = std::begin(container); it != std::end(container); ++it) {
18         std::cout << *it << " ";
19     } std::cout << std::endl;
20 }
```

Summary on metaprogramming techniques

Metaprogramming

- It works
- Allows to guide the compiler “by hand”

Too restrictive (but can be extended with more time)

- In the first case, printing should not be limited to arithmetic type
- In the second case, C-arrays are not taken into account

Limitations

- Lots of template trickery involved
- Difficult to read and debug
- Complicated error messages

Important note

- Start from the bottom: concrete types and shared properties

High Performance Software Architecture

1 Introduction

2 Metaprogramming

3 High Performance Software Architecture

4 Concepts

Using types to guide the compiler to the most performant algorithm

The example of the size of a container

Create a function `element_count(x)` so that:

- if x has a known size with a $\mathcal{O}(1)$ access, retrieve it
- otherwise, if x can be iterated through, iterate through it to compute the size in $\mathcal{O}(N)$
- otherwise, assume that x is only one element, and returns 1

In C++ terms

Create a function `template <class T> element_count(T&&)` so that:

- if T has a `::size` member function, call it
- otherwise, if `std::begin` and `std::end` can be called on T , then compute the size using `std::distance`
- otherwise, assume that T has only one element, and returns 1

Checking code validity



Goal

```
1 template <class T>
2 constexpr std::size_t element_count(T&& x) {
3     std::size_t count = 0;
4     if constexpr (/* x.size() is valid code */) {
5         count = x.size();
6     } else if constexpr (/* std::distance(std::begin(x), std::end(x)) is valid code*/) {
7         count = std::distance(std::begin(x), std::end(x));
8     } else {
9         count = 1;
10    }
11    /* ... */
12    return count;
13 }
```

An observation about lambdas

Checking lambda invocability: using lambdas

```
1 template <class F, class... Args>
2 bool check(F&&, Args&&...) {
3     return std::is_invocable_v<F, Args...>;
4 }

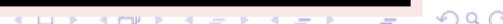
1 int main(int argc, char* argv[]) {
2     check([](auto x){return x.size();}, 42);
3     check([](auto x){return x.size();}, std::vector<int>{});
4     return 0;
5 }
```

```
#> g++ -std=c++17 lambda.cpp -o lambda && ./lambda
error: request for member 'size' in 'x', which is of non-class type 'int'
check([](auto x){return x.size();}, 42);
```

Checking lambda invocability: using lambdas and decltype(auto)

```
1 int main(int argc, char* argv[]) {
2     check([](auto x) -> decltype(auto) {return x.size();}, 42);
3     check([](auto x) -> decltype(auto) {return x.size();}, std::vector<int>{});
4     return 0;
5 }
```

```
#> g++ -std=c++17 lambda.cpp -o lambda && ./lambda
error: request for member 'size' in 'x', which is of non-class type 'int'
check([](auto x){return x.size();}, 42);
```



A second observation about lambdas

Checking lambda invocability: using lambdas and decltype(/*...*/)

```
1 int main(int argc, char* argv[]) {
2     std::cout << std::boolalpha;
3     std::cout << check([](auto x) -> decltype(x.size()) {return x.size();}, 42) << '\n';
4     std::cout << check([](auto x) -> decltype(x.size()) {return x.size();}, std::vector<int>{})
5     std::cout << std::endl;
6     return 0;
7 }
```

```
#> g++ -std=c++17 lambda.cpp -o lambda && ./lambda
false
true
```

Checking lambda invocability: removing the return statement

```
1 int main(int argc, char* argv[]) {
2     std::cout << std::boolalpha;
3     std::cout << check([](auto x) -> decltype(x.size()) {}, 42) << std::endl;
4     std::cout << check([](auto x) -> decltype(x.size()) {}, std::vector<int>{}) << std::endl;
5     return 0;
6 }
```

```
#> g++ -std=c++17 lambda.cpp -o lambda && ./lambda
false
true
```

Inline SFINAE through lambdas

Mixing SFINAE and lambdas to achieve inline SFINAE: the basic idea

```
1 // Preamble
2 #include <vector>
3 #include <iostream>
4 #include <type_traits>
5
6 // Check
7 template <class F, class... Args>
8 constexpr std::is_invocable<F, Args...> check(F&&, Args&&...) noexcept {
9     return std::is_invocable<F, Args...>();
10 }
11
12 // Main
13 int main(int argc, char* argv[]) {
14     std::cout << std::boolalpha;
15     std::cout << check([](auto x) -> decltype(x.size()) {}, 42) << std::endl;
16     std::cout << check([](auto x) -> decltype(x.size()) {}, std::vector<int>{}) << std::endl;
17     return 0;
18 }
```

```
#> g++ -std=c++17 lambda.cpp -o lambda && ./lambda
false
true
```

Leveraging the idea: validator

The validator class

```
1 // Validator class
2 template <class... Callables>
3 struct validator
4 {
5     // Types and constants
6     template <class... Args>
7     static constexpr bool is_invocable_v = (std::is_invocable_v<Callables, Args...> && ...);
8     template <class... Args>
9     using is_invocable = std::bool_constant<is_invocable_v<Args...>>;
10
11    // Constructors
12    constexpr validator() noexcept = default;
13    template <
14        class... F,
15        class = std::enable_if_t<sizeof...(F) == sizeof...(Callables)>,
16        class = std::enable_if_t<std::is_constructible_v<std::tuple<Callables...>, F...>
17    >
18    constexpr validator(F&&...) noexcept {}
19
20    // Function call operator
21    template <class... Args>
22    constexpr is_invocable<Args...> operator()(Args&&...) const noexcept {
23        return is_invocable<Args...>();
24    }
25};
26 // Deduction guide
27 template <class... Callables> validator(Callables&&...) -> validator<F...>;
28 // Type traits
29 template <class T> struct is_validator: std::false_type {};
30 template <class... Callables> struct is_validator<validator<Callables...>>: std::true_type {};
31 template <class T> inline constexpr bool is_validator_v = is_validator<T>::value;
```

Leveraging the idea: validate

The validate function

```
1 // Helper struct remove_cvref (part of C++20)
2 template <class T>
3 struct remove_cvref: std::remove_cv<std::remove_reference<T>> {};
4 template <class T>
5 using remove_cvref_t = typename remove_cvref<T>::type;
6
7 // Validate function
8 template <
9     class... Args,
10    class... Callables,
11    class = std::enable_if_t<(!is_validator_v<std::remove_cvref_t<Callables>>) && ...>
12 >
13 constexpr std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>
14 validate(Callables&&...) noexcept {
15     return std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>();
16 }
17
18 // Overload for validator
19 template <class... Args, class... Callables>
20 typename validator<Callables...>::template is_invocable<Args...>
21 validate(const validator<Callables...>&)
22 {
23     return typename validator<Callables...>::template is_invocable<Args...>();
24 }
```

Leveraging the idea: `is_valid`

The validate class

```
1 // Is valid class: syntax is_valid<std::vector<int>>([](auto x) -> decltype(x.size()) {})
2 template <class... Args>
3 struct is_valid
4 {
5     // Constructor from callables
6     template <class... Callables, class = std::enable_if_t<(!is_validator_v<remove_cvref_t<Callables>>) && ...>>
7     constexpr is_valid(Callables...&...) noexcept
8     : value((std::is_invocable_v<Callables, Args...> && ...)) {}
9
10    // Constructor using validator
11    template <class... Callables>
12    constexpr is_valid(const validator<Callables...>&) noexcept
13    : value(validator<Callables...>::template is_invocable_v<Args...>) {}
14
15    // Conversion to bool
16    constexpr operator bool() const {return value;}
17
18    // Implementation details
19    private: bool value;
20};
```

Leveraging the idea: `is_valid`

The validate class

```
1 // Specialization for deduction guide: syntax is_valid(std::vector<int>{})([](auto x) -> decltype(x.size()) {})
2 template <class... Args>
3 struct is_valid<void, Args...>
4 {
5     // Constructor from arguments
6     template <
7         class... Types,
8         class = std::enable_if_t<std::is_constructible_v<std::tuple<Args..., Types...>>
9     >
10    constexpr is_valid(Types&&...): {}
11
12    // Caller
13    template <
14        class... Callables,
15        class = std::enable_if_t<(!is_validator_v<remove_cvref_t<Callables>>) && ...>
16    >
17    constexpr std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>
18    operator()(Callables&&...) const noexcept {
19        return std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>();
20    }
21
22    // Caller using validator
23    template <class... Callables>
24    constexpr typename validator<Callables...>::template is_invocable<Args...>
25    operator()(const validator<Callables...>&) const noexcept {
26        return typename validator<Callables...>::template is_invocable<Args...>();
27    }
28 };
29
30 // Deduction guide
31 template <class... Args>
32 is_valid(Args&&...) -> is_valid<void, Args...>;
```



Syntax of validator, validate and is_valid

Syntax summary

```
1 // Validator
2 validator([](auto x) -> decltype(x.size()){})(42);           // false
3 validator([](auto x) -> decltype(x.size()){})(std::vector<int>{}); // true
4
5 // Validate
6 validate<int>([](auto x) -> decltype(x.size()){});           // false
7 validate<std::vector<int>>([](auto x) -> decltype(x.size()){}); // true
8
9 // Is valid
10 is_valid<int>([](auto x) -> decltype(x.size()){});           // false
11 is_valid<std::vector<int>>([](auto x) -> decltype(x.size()){}); // true
12 is_valid(42)([](auto x) -> decltype(x.size()){});             // false
13 is_valid(std::vector<int>{})([](auto x) -> decltype(x.size()){}); // true
```

More complicated statements

```
1 // Checks if all expressions are valid
2 is_valid<int>(
3     [](auto i) -> decltype(i + i) {},
4     [](auto i) -> decltype(i * i) {}
5 );
6
7 // For all types checks if all expressions are valid
8 is_valid<int, std::vector<int>>(
9     [](auto i, auto) -> decltype(i + i) {},
10    [](auto, auto v) -> decltype(v.size()) {}
11 );
```

And finally, inline SFINAE

Solving the original problem!

```
1 template <class T>
2 constexpr std::size_t element_count(T&& x) {
3     std::size_t count = 0;
4     if constexpr (is_valid<T>([](auto x) -> decltype(x.size()) {})) {
5         count = x.size();
6     } else if constexpr (is_valid<T>(
7         [](auto x) -> decltype(std::begin(x)) {},
8         [](auto x) -> decltype(std::end(x)) {}
9     )) {
10         count = std::distance(std::begin(x), std::end(x));
11     } else {
12         count = 1;
13     }
14     return count;
15 }
```



Concepts

1 Introduction

2 Metaprogramming

3 High Performance Software Architecture

4 Concepts

Concepts and constraints in C++20

Concept

- Named set of requirements
- Must appear at namespace scope

```
1 template <template-parameter-list*>
2 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

```
1 // Concept definition
2 template <class T>
3 concept arithmetic = std::is_arithmetic_v<T>;
4
5 // Constrained function v1
6 template <arithmetic T>
7 void print_v1(T x) {
8     std::cout << x << std::endl;
9 }
10
11 // Constrained function v2
12 template <class T>
13 requires arithmetic<T>
14 void print_v2(T x) {
15     std::cout << x << std::endl;
16 }
17
18 // Constrained function v3
19 void print_v3(arithmetic auto x) {
20     std::cout << x << std::endl;
21 }
```

Example of constraints

Constraints on addability

```
1 template <class T>
2 concept addable = requires {std::declval<T>() + std::declval<T>();}
3
4 template <class T>
5 concept addable = requires (T x) {x + x;}
6
7 template <class T1, class T2>
8 concept addable2 = requires (T1 x, T2 y) {x + y;}
9
10 template <class T>
11 concept addable_and_multiplicable = addable<T> && requires (T x) {x * x;}
12
13 template <class T>
14 requires requires (T x) {x + x;}
15 T add(T x, T y) {
16     return x + y;
17 }
```

Combining concepts and constraints with if constexpr

With an external concept

```
1 template <class T>
2 concept shiftable = requires {std::declval<T>() << std::declval<int>();}
3
4 template <class T>
5 void is_shiftable() {
6     if constexpr (shiftable<T>) {std::cout << "shiftable" << std::endl;}
7     else {std::cout << "not shiftable" << std::endl;}
8 }
9
10 // is_shiftable<int>() -> "shiftable"
11 // is_shiftable<double>() -> "not shiftable"
```

With an inline **requires** clause

```
1 template <class T>
2 void is_shiftable() {
3     if constexpr (requires {std::declval<T>() << std::declval<int>();}) {std::cout << "shiftable" << std::endl;}
4     else {std::cout << "not shiftable" << std::endl;}
5 }
```

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

```
1 template <class T>
2 void is_shiftable() {
3     if constexpr (requires (T x, int y) {x << y;}) {std::cout << "shiftable" << std::endl;}
4     else {std::cout << "not shiftable" << std::endl;}
5 }
```

Checking if a function exists

The traditional way: the preprocessor

```
1 #ifdef __SUPPORTS_THEFUNCTION
2 /* Doing something here */
3 #endif
```

The metaprogramming way

```
1 //void thefunction(int x);
2
3 template <class T, class = decltype(thefunction(std::declval<T>()))>
4 std::true_type supports_thefunction_for(T);
5 template <class T, class... X>
6 std::false_type supports_thefunction_for(T, X...);
7
8 inline constexpr bool supports_thefunction
9 = decltype(supports_thefunction_for(std::declval<int>()))::value;
10 // true if thefunction(int x) is active
11 // false if thefunction(int x) is commented out
```

The concept-based way

```
1 //void thefunction(int x);
2
3 template <class T = int>
4 concept supports_thefunction_for
5 = requires (T x) {thefunction(x);};
```

Checking if a function exists: forcing template dependency

Leveraging alias templates

```
1 //void thefunction(int x);
2
3 // The concept checks for a particular type provided by the user
4 template <class T>
5 concept supports_thefunction_for
6 = requires (T x) {thefunction(x);};
7
8 // Alias template keeping only the first type
9 template <class T, class...>
10 using first_type = T;
11
12 // The concept ignores its template parameter and tests only the relevant type
13 template <class... Dummy>
14 concept supports_thefunction
15 = requires {thefunction(std::declval<first_type<int, Dummy...>>());};
```

Constexpr if and requires clauses

The problem of undefined symbols

```
1  /*
2  void thefunction(int) {
3      std::cout << "thefunction" << std::endl;
4  }*/
5
6  template <class T>
7  void check(T x) {
8      if constexpr (requires (T y) {thefunction(y);}) {
9          thefunction(x); // OK
10         thefunction(3); // ERROR
11         []<class U>(U x){thefunction(x);} (3); // OK
12     } else {
13         std::cout << "not thefunction" << std::endl;
14     }
15 }
16
17 check(1); //
```

Coming back to the problem of printing

Better than std::enable_if

```
1 // For numbers
2 template <printable T>
3 void print(const T& x) {
4     std::cout << x << std::endl;
5 }
6
7 // For container of numbers
8 template <range R>
9 requires printable<decltype(*std::begin(std::declval<R>()))>
10 void print(const R& range) {
11     for (auto it = std::begin(container); it != std::end(container); ++it) {
12         std::cout << *it << " ";
13     } std::cout << std::endl;
14 }
```

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