From Metaprogramming Tricks to Elegance

Custom Overload Sets and Inline SFINAE for Truly Generic Interfaces

Vincent Reverdy
(vreverdy@illinois.edu)

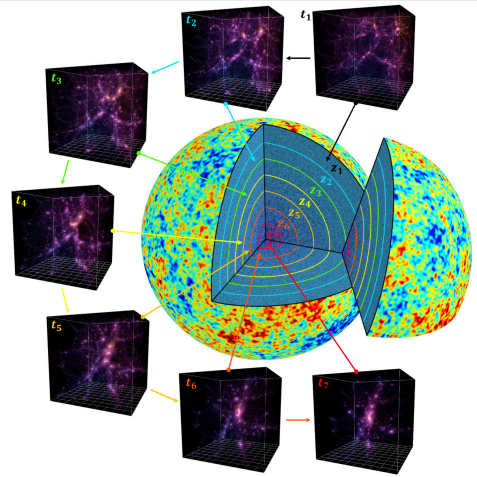
Department of Astronomy, University of Illinois at Urbana-Champaign
Laboratory Universe and Theories, Paris Observatory

September 27th, 2018

Introduction

Once upon a time...

...everything started with the Big Bang



Doing high performance numerical astrophysics. . .

. . . while exploring template metaprogramming (back in 2011)

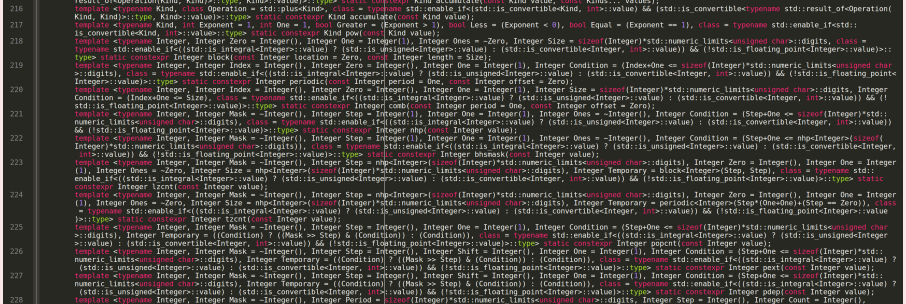
```
207     // Helpers
208     /// \name           Helpers
209     //@{
210     public:
211         template <class Template, class Function, class Tuple, typename... Kinds, class = typename std::enable_if<(sizeof...(Kinds)+1 <= std::tuple_size<Template>::value)>::type> static constexpr Templa
212         template <class Template, class Function, class Tuple, typename... Kinds, class = typename std::enable_if<(sizeof...(Kinds) == std::tuple_size<Template>::value)>::type> static constexpr Template
213         template <class Template, class Function, typename Kind, class = typename std::enable_if<std::is_convertible<Template, int>::value>::type> static constexpr Template apply(Function&& f, Kind&& va
214         template <typename Kind, class Operation = std::plus<Kind>, unsigned int Index = 0, class Tuple, bool Condition = (Index+2 <= std::tuple_size<typename std::remove_reference<
215         template <typename Kind, class Operation = std::plus<Kind>, typename... Kinds, class = typename std::enable_if<(std::is_convertible<Kind, int>::value) && (std::is_convertible<typename std::resul
216         template <typename Kind, class Operation = std::plus<Kind>, class = typename std::enable_if<(std::is_convertible<Kind, int>::value) && (std::is_convertible<typename std::result_of<Operation(Kind
217         template <typename Kind, int Exponent = 1, int One = 1, bool Greater = (Exponent > 1), bool Less = (Exponent < 0), bool Equal = (Exponent == 1), class = typename std::enable_if<std::is_convertib
218         template <typename Integer, Integer Zero = Integer(), Integer One = Integer(1), Integer Ones = ~Zero, Integer Size = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, class = typename
219         template <typename Integer, Integer Index = Integer(), Integer Zero = Integer(), Integer One = Integer(1), Integer Condition = (Index+One <= sizeof(Integer)*std::numeric_limits<unsigned char>::d
220         template <typename Integer, Integer Index = Integer(), Integer Zero = Integer(), Integer One = Integer(1), Integer Size = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer Cons
221         template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(1), Integer One = Integer(1), Integer Ones = ~Integer(), Integer Condition = (Step+One <= sizeof(Integer)*std::numer
222         template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(1), Integer One = Integer(1), Integer Ones = ~Integer(), Integer Condition = (Step+One <= nhp<Integer>(sizeof(Intege
223         template <typename Integer, Integer Mask = ~Integer(), Integer Step = nhp<Integer>(sizeof(Integer)*std::numeric_limits<unsigned char>::digits), Integer Zero = Integer(), Integer One = Integer(1
224         template <typename Integer, Integer Mask = ~Integer(), Integer Step = nhp<Integer>(sizeof(Integer)*std::numeric_limits<unsigned char>::digits), Integer Zero = Integer(), Integer One = Integer(1
225         template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(), Integer One = Integer(1), Integer Condition = (Step+One <= sizeof(Integer)*std::numeric_limits<unsigned char>::di
226         template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(), Integer Shift = Integer(), Integer One = Integer(1), Integer Condition = (Step+One <= sizeof(Integer)*std::numer
227         template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(), Integer Shift = Integer(), Integer One = Integer(1), Integer Condition = (Step+One <= sizeof(Integer)*std::numer
228         template <typename Integer, Integer Mask = ~Integer(), Integer Period = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer Step = Integer(), Integer Count = Integer(), Integer Z
229         template <typename Integer, Integer Mask = ~Integer(), Integer Period = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer Step = Integer(), Integer Count = Integer(), Integer Z
230         template <typename Integer, Integer Mask = ~Integer(), Integer Length = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, bool Msb = false, std::size_t Step = Integer(), Integer Zero =
231         template <typename Integer, Integer Mask = ~Integer(), Integer Length = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, bool Msb = false, std::size_t Step = Integer(), Integer Zero =
232         template <typename Integer, Integer Mask = ~Integer(), Integer Length = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, bool Msb = false, std::size_t Step = Integer(), Integer Zero =
233     //@}
```

. . . leads to great performances. . .

. . . and great genericity . . .

Except. . .

. . . when you unfold the code. . .



You realize...

...that no sane human...

...can read your code...

Aside notes

- The code works great and deliver amazing performance
- It's very modular as long as you don't dig into the details
- It's just completely unmaintainable
- And the minute you need to dig into the details... you want to run away and `std::abort`
- (and by the way, the expression "sane human" probably does not apply to C++ developers)

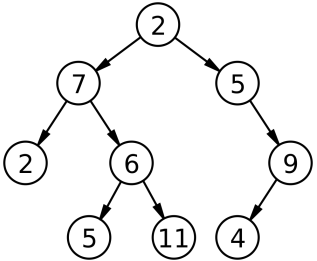
That being said...

...the code is available at https://github.com/vreverdy/magrathea-pathfinder, it is still running on petascale supercomputers, and delivers great scientific analyses on relativistic light propagation.

So now what?

Since then...

- Learned how to write readable template metaprogramming code
- Have been working on a number of proposals to the C++ committee (including bit manipulation)
- Have been working on a generic high performance tree library for simulations and machine learning
- The C++17 standard has been published



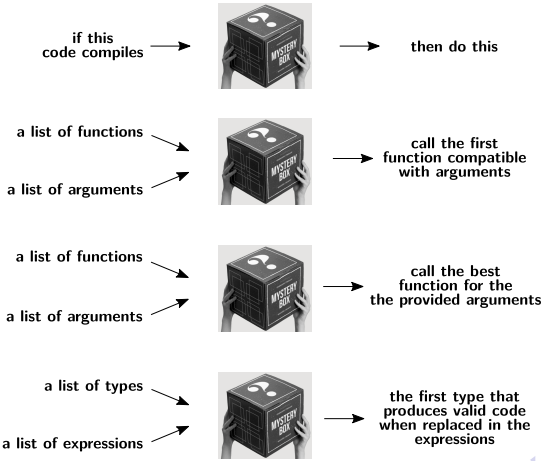
Main question

How to write a super generic tree library in a very readable manner?

What is this presentation about?

Main question

How to improve code genericity in C++17 without paying the price of code complexity and readability?



Substitution Failure Is Not An Error

Substitution Failure Is Not An Error (SFINAE)

According to Wikipedia

Substitution failure is not an error (SFINAE) refers to a situation in C++ where an invalid substitution of template parameters is not in itself an error.

According to cppreference

"Substitution Failure Is Not An Error". This rule applies during overload resolution of function templates: When substituting the deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error. This feature is used in template metaprogramming.

According to the standard [temp.deduct]/8

If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed, with a diagnostic required, if written using the substituted arguments. *[Note: If no diagnostic is required, the program is still ill-formed. Access checking is done as part of the substitution process. - end note]* **Only invalid types and expressions in the immediate context of the function type, its template parameter types, and its explicit-specifier can result in a deduction failure.** *[Note: The substitution into types and expressions can result in effects such as the instantiation of class template specializations and/or function template specializations, the generation of implicitly-defined functions, etc. Such effects are not in the "immediate context" and can result in the program being ill-formed. - end note]*

Introductory example

Example of SFINAE from cppreference

```cpp
// Preamble
#include <iostream>

// This overload is always in the set of overloads ellipsis parameter has the lowest ranking for overload resolution
void test(...) {std::cout << "Catch-all overload called";}

// This overload is added to the set of overloads if C is a ref-to-class type and F is a ptr to member function of C
template <class C, class F>
auto test(C c, F f) -> decltype((void)(c.*f)(), void()) {std::cout << "Reference overload called";}

// This overload is added to the set of overloads if C is a ptr-to-class type and F is a ptr to member function of C
template <class C, class F>
auto test(C c, F f) -> decltype((void)((c->*f)()), void()) {std::cout << "Pointer overload called";}

// Structure with a function f
struct X {void f() {}};

// Main
int main(int argc, char* argv[]) {
    X x;
    test( x, &X::f);
    test(&x, &X::f);
    test(42, 1337);
    return 0;
}
```

```
#> g++ -std=c++17 sfinae.cpp -o sfinae && ./sfinae
Reference overload called
Pointer overload called
Catch-all overload called
```

std::enable_if (C++11)

Example of use of std::enable_if

```cpp
// Preamble
#include <iostream>
#include <type_traits>

// Function for integers
template <class T, class = std::enable_if_t<std::is_integral_v<T>>>
void function(T x) {std::cout << "integer" << std::endl;}

// Function for floats: remark the function signature needs to be different
template <class T, class = std::enable_if_t<std::is_floating_point_v<T>>>
void function(const T& x) {std::cout << "floating point" << std::endl;}

// Default structure
template <class T, class = void>
struct structure {
    void operator()() const {std::cout << "structure<T>" << std::endl;}
};

// Specialization for integers
template <class T>
struct structure<T, std::enable_if_t<std::is_integral_v<T>>> {
    void operator()() const {std::cout << "structure<integer>" << std::endl;}
};

// Main
int main(int argc, char* argv[]) {
    function(5);                // integer
    function(0.5);              // floating point
    structure<double>{}();      // structure<T>
    structure<int>{}();         // structure<integer>
    return 0;
}
```

std::void_t (C++17)

Example of use of std::void_t from cppreference

```cpp
// Preamble
#include <iostream>
#include <type_traits>
#include <vector>
#include <map>

// An empty class
class A {};

// A generic trait to detected if a type is iterable
template <class T, class = void>
struct is_iterable: std::false_type {};

// The true specialization when begin and end can be called
template <typename T>
struct is_iterable<T, std::void_t<
    decltype(std::begin(std::declval<T>())),
    decltype(std::end(std::declval<T>()))>
>: std::true_type {};

// Main
int main(int argc, char* argv[]) {
    std::cout << std::boolalpha;
    std::cout << is_iterable<std::vector<double>>::value << std::endl;      // true
    std::cout << is_iterable<std::map<int, double>>::value << std::endl;    // true
    std::cout << is_iterable<double>::value << std::endl;                   // false
    std::cout << is_iterable<A>::value << std::endl;                        // false
    return 0;
}
```

Remarks on SFINAE, `std::enable_if` and `std::void_t`

Notes

- Amazing tools for all sorts of metaprogramming tricks and code genericity
- Sometimes requires a lot of trickery to make it work
- Difficult to clearly specify the intent: bad code expressivity
- Requires the definition of functions or classes for each SFINAE trick
- Leads to a lot of boilerplate code and can negatively impact code locality
- Even if `concepts` will solve some of these problems, SFINAE will remain useful in many situations

Also...

Wouldn't it be great to be able to write inline SFINAE without extra boilerplate code?

Illustration of some of the non-trivialities

Immediate context or not?

```
1  // Preamble
2  #include <type_traits>
3
4  // Trait
5  template<class T>
6  struct trait{
7      using type = typename T::type;
8  };
9
10 // Function immediate_context
11 void immediate_context(...) {}
12 template<class T, class = typename T::type>
13 void immediate_context(T = 0) {}
14
15 // Function nonimmediate_context
16 void nonimmediate_context(...) {}
17 template<class T, class = typename trait<T>::type>
18 void nonimmediate_context(T = 0) {}
19
20 // Structure
21 template<class T>
22 struct structure {
23     using type = void;
24 };
25
26 // Main
27 int main(int argc, char* argv[]){
28     immediate_context(0);                       // Works!
29     immediate_context(structure<int>{});        // Works!
30     nonimmediate_context(0);                    // ERROR: error: 'int' is not a class, struct, or union type
31     nonimmediate_context(structure<int>{});     // Works!
32 }
```

Implementing a ranked overload set using the variadic template trick and SFINAE

Goal

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`

Trick

Non-variadic template parameters are always preferred against variadic template parameters.

Strategy

We will apply the trick recursively.

Implementation of element_count (1/2)

First, when T has a ::size or is iterable

```
// Non-variadic version: when T has ::size
template <
    class T,
    class = decltype(std::declval<T>().size())
>
std::size_t container_element_count(T&& value) {
    return std::forward<T>(value).size();
}

// Variadic version: when T is iterable
template <
    class T,
    class... X,
    class = std::enable_if_t<sizeof...(X) == 0>,
    class = decltype(std::begin(std::declval<T>())),
    class = decltype(std::end(std::declval<T>()))
>
std::size_t container_element_count(T&& value, X...) {
    return std::distance(
        std::begin(std::forward<T>(value)),
        std::end(std::forward<T>(value))
    );
}
```

Implementation of element_count (2/2)

Second, when T has a neither ::size nor is iterable

```cpp
// Special case for initializer list
template <class T>
std::size_t element_count(const std::initializer_list<T>& value) {
    return std::distance(std::begin(value), std::end(value));
}

// Non-variadic case: call container_element_count
template <
    class T,
    class = decltype(container_element_count(std::declval<T>()))
>
std::size_t element_count(T&& value) {
    return container_element_count(std::forward<T>(value));
}

// Variadic case: when T has neither ::size nor is iterable
template <
    class T,
    class... X,
    class = std::enable_if_t<sizeof...(X) == 0>
>
std::size_t element_count(T&& value, X...) {
    return 1;
}
```

Use of element_count

Result

```cpp
int main(int argc, char* argv[])
{
    element_count(42);                              // Returns 1
    element_count(std::forward_list({1, 2, 3}));    // Uses std::distance
    element_count(std::vector({1, 2, 3, 4}));       // Uses .size
    element_count({1, 2, 3});                       // Uses the initializer_list overload
    return 0;
}
```

Using an overload_rank to rank functions

Introducing overload_rank

```
1 template <std::size_t N> struct overload_rank: overload_rank<N - 1>  {};
2 template <> struct overload_rank<0> {};
```

Alternative implementation of element_count

```
1  // When T is an initializer_list
2  template <class T>
3  std::size_t element_count(const std::initializer_list<T>& value) {
4      return std::distance(std::begin(value), std::end(value));
5  }
6  // When T has size
7  template <class T, class = decltype(std::declval<T>().size())>
8  std::size_t element_count(overload_rank<2>, T&& x) {
9      return std::forward<T>(x).size();
10 }
11 // When T is iterable
12 template <class T, class = decltype(std::begin(std::declval<T>())), class = decltype(std::end(std::declval<T>()))>
13 std::size_t element_count(overload_rank<1>, T&& x) {
14     return std::distance(std::begin(std::forward<T>(x)), std::end(std::forward<T>(x)));
15 }
16 // When T neither has size nor is iterable
17 template <class T>
18 std::size_t element_count(overload_rank<0>, T&& x) {
19     return 1;
20 }
21 // Caller
22 template <class T>
23 std::size_t element_count(T&& x) {
24     return element_count(overload_rank<2>(), std::forward<T>(x));
25 }
```

Concluding on SFINAE

Result with overload_rank

```
1 int main(int argc, char* argv[]) {
2     element_count(42);                              // Calls overload_rank<0> version
3     element_count(std::forward_list({1, 2, 3}));    // Calls overload_rank<1> version
4     element_count(std::vector({1, 2, 3, 4}));       // Calls overload_rank<2> version
5     element_count({1, 2, 3});                       // Calls initializer_list version
6     return 0;
7 }
```

Remark

overload_rank improves things a little bit, but still complex and with a lot of boilerplate code.

Reminder on SFINAE

- Amazing tools for all sorts of metaprogramming tricks and code genericity
- Sometimes requires a lot of trickery to make it work
- Difficult to clearly specify the intent: bad code expressivity
- Requires the definition of functions or classes for each SFINAE trick
- Leads to a lot of boilerplate code and can negatively impact code locality
- Even if concepts will solve some of these problems, SFINAE will remain useful in many situations

Again...

Wouldn't it be great to be able to write inline SFINAE without extra boilerplate code?

Checking code validity

Checking code validity

if this code compiles → 

→ then do this

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

```cpp
// Preamble
#include <vector>
#include <iostream>
#include <type_traits>

// Check
template <class F, class... Args>
constexpr std::is_invocable<F, Args...> check(F&&, Args&&...) noexcept {
    return std::is_invocable<F, Args...>();
}

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

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

Leveraging the idea: validator

The validator class

```cpp
// Validator class
template <class... Callables>
struct validator
{
    // Types and constants
    template <class... Args>
    static constexpr bool is_invocable_v = (std::is_invocable_v<Callables, Args...> && ...);
    template <class... Args>
    using is_invocable = std::bool_constant<is_invocable_v<Args...>>;

    // Constructors
    constexpr validator() noexcept = default;
    template <
        class... F,
        class = std::enable_if_t<sizeof...(F) == sizeof...(Callables)>,
        class = std::enable_if_t<std::is_constructible_v<std::tuple<Callables...>, F...>
    >
    constexpr validator(F&&...) noexcept {}

    // Function call operator
    template <class... Args>
    constexpr is_invocable<Args...> operator()(Args&&...) const noexcept {
        return is_invocable<Args...>();
    }
};
// Deduction guide
template <class... Callables> validator(Callables&&...) -> validator<F...>;
// Type traits
template <class T> struct is_validator: std::false_type {};
template <class... Callables> struct is_validator<validator<Callables...>>: std::true_type {};
template <class T> inline constexpr bool is_validator_v = is_validator<T>::value;
```

Leveraging the idea: validate

The validate function

```cpp
// Helper struct remove_cvref (part of C++20)
template <class T>
struct remove_cvref: std::remove_cv<std::remove_reference<T>> {};
template <class T>
using remove_cvref_t = typename remove_cvref<T>::type;

// Validate function
template <
    class... Args,
    class... Callables,
    class = std::enable_if_t<((!is_validator_v<std::remove_cvref_t<Callables>>) && ...)>
>
constexpr std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>
validate(Callables&&...) noexcept {
    return std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>();
}

// Overload for validator
template <class... Args, class... Callables>
typename validator<Callables...>::template is_invocable<Args...>
validate(const validator<Callables...>&)
{
    return typename validator<Callables...>::template is_invocable<Args...>();
}
```

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

```cpp
// Specialization for deduction guide: syntax is_valid(std::vector<int>{})([](auto x) -> decltype(x.size()) {})
template <class... Args>
struct is_valid<void, Args...>
{
    // Constructor from arguments
    template <
        class... Types,
        class = std::enable_if_t<std::is_constructible_v<std::tuple<Args...>, Types...>
    >
    constexpr is_valid(Types&&...): {}

    // Caller
    template <
        class... Callables,
        class = std::enable_if_t<((!is_validator_v<remove_cvref_t<Callables>>) && ...)>
    >
    constexpr std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>
    operator()(Callables&&...) const noexcept {
        return std::bool_constant<(std::is_invocable_v<Callables, Args...> && ...)>();
    }

    // Caller using validator
    template <class... Callables>
    constexpr typename validator<Callables...>::template is_invocable<Args...>
    operator()(const validator<Callables...>&) const noexcept {
        return typename validator<Callables...>::template is_invocable<Args...>();
    }
};

// Deduction guide
template <class... Args>
is_valid(Args&&...) -> is_valid<void, Args...>;
```

Syntax of validator, validate and is_valid

Syntax summary

```cpp
// Validator
validator([](auto x) -> decltype(x.size()){})(42);                          // false
validator([](auto x) -> decltype(x.size()){})(std::vector<int>{});          // true

// Validate
validate<int>([](auto x) -> decltype(x.size()){});                          // false
validate<std::vector<int>>([](auto x) -> decltype(x.size()){});             // true

// Is valid
is_valid<int>([](auto x) -> decltype(x.size()){});                          // false
is_valid<std::vector<int>>([](auto x) -> decltype(x.size()){});             // true
is_valid(42)([](auto x) -> decltype(x.size()){});                           // false
is_valid(<std::vector<int>>{})([](auto x) -> decltype(x.size()){});         // true
```

More complicated statements

```cpp
// Checks if all expressions are valid
is_valid<int>(
    [](auto i) -> decltype(i + i) {},
    [](auto i) -> decltype(i * i) {}
);

// For all types checks if all expressions are valid
is_valid<int, std::vector<int>>(
    [](auto i, auto) -> decltype(i + i) {},
    [](auto, auto v) -> decltype(v.size()) {}
);
```

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 }
```

Custom overload sequences

Custom overload sequences

Goal

```
1 f(
2     [](auto x) -> decltype(x.size()) {return x.size();},
3     [](auto x){return x;},
4     [](auto x, auto y){return x + y;},
5     [](auto x, auto y, auto z){return x + y + z;}
6 )(10, 42); // should return 52
```

Preamble

Strategy

We will use the same type of tricks as for inline SFINAE.

Helper struct: a standard type list would be nice, but we'll use a `tuple` instead

```cpp
// Index constant
template <std::size_t Value>
using index_constant = std::integral_constant<std::size_t, Value>;
```

```cpp
// Tuple index: returns the first index at which the provided type appears in the tuple
template <class...> struct tuple_index;

// Generic version
template <class Type, class... Types>
struct tuple_index<Type, std::tuple<Types...>>
: tuple_index<index_constant<0>, Type, std::tuple<Types...>> {};

// Implementation detail: recursive call
template <std::size_t Index, class Type, class Head, class... Tail>
struct tuple_index<index_constant<Index>, Type, std::tuple<Head, Tail...>>
: std::conditional_t<
    std::is_same_v<Type, Head>,
    index_constant<Index>,
    tuple_index<index_constant<Index + 1>, Type, std::tuple<Tail...>>
> {};

// Implementation detail: end of recursion
template <std::size_t Index, class Type>
struct tuple_index<index_constant<Index>, Type, std::tuple<>>{};

// Convenience alias
template <class Type, class Tuple>
inline constexpr std::size_t tuple_index_v = tuple_index<Type, Tuple>::value;
```

Use of tuple_index

Example of use of tuple_index

```cpp
int main(int argc, char* argv[]) {
    std::cout << tuple_index_v<int, std::tuple<int, double, double>> << std::endl;
    std::cout << tuple_index_v<double, std::tuple<int, double, double>> << std::endl;
    return 0;
}
```

```
#> g++ -std=c++17 tuple_index.cpp -o tuple_index && ./tuple_index
0
1
```

Will be used on sequence of boolean constants

```cpp
int main(int argc, char* argv[]) {
    using tuple = std::tuple<
        std::false_type, std::false_type, std::false_type,
        std::true_type, std::false_type, std::true_type
    >;
    std::cout << tuple_index_v<std::true_type, tuple> << std::endl;
    return 0;
}
```

```
#> g++ -std=c++17 tuple_index_bool.cpp -o tuple_index_bool && ./tuple_index_bool
3
```

Remark

The same strategy could be used on a `std::integer_sequence<bool, bool...>` instead.

Finding the first invocable function

Introducing overload_sequence_selector

```cpp
// A list of callables
template <class... Callables> class overload_sequence;

// Selector declaration
template <class...> struct overload_sequence_selector;

// Specialization for overload sequence
template <class... Callables, class... Args>
struct overload_sequence_selector<overload_sequence<Callables...>, Args...>
{
    // Get the index of the first invocable callable
    static constexpr std::size_t value = tuple_index_v<
        std::true_type,
        std::tuple<std::bool_constant<std::is_invocable_v<Callables, Args...>>...>
    >;

    // Get the callable type corresponding to the index
    using type = std::tuple_element_t<value, std::tuple<Callables...>>;
};

// Convenience aliases
template <class... T>
using overload_sequence_selector_t = typename overload_sequence_selector<T...>::type;
template <class... T>
inline constexpr auto overload_sequence_selector_v = overload_sequence_selector<T...>::value;
```

And finally build an overload_sequence

Implementation of overload_sequence

```cpp
template <class... Callables>
struct overload_sequence
{
    // Types
    using sequence_type = std::tuple<Callables...>;
    template <class... Args> using selector = overload_sequence_selector<overload_sequence<Callables...>, Args...>;
    template <class... Args> using selector_t = typename selector<Args...>::type;
    template <class... Args> static constexpr std::size_t selector_v = selector<Args...>::value;

    // Constructors
    constexpr overload_sequence(): _sequence() {}
    template <class... F, class = std::enable_if_t<std::is_constructible_v<std::tuple<Callables...>, F...>>>
    constexpr overload_sequence(F&&... f): _sequence(std::forward<F>(f)...) {}

    // Call operator
    template <class... Args>
    constexpr std::invoke_result_t<selector_t<Args...>, Args...> operator()(Args&&... args) {
        return std::invoke(std::get<selector_v<Args...>>(_sequence), std::forward<Args>(args)...);
    }

    // Implementation details: data members
    private: sequence_type _sequence;
};

// Deduction guide
template <class... Callables> overload_sequence(Callables&&...) -> overload_sequence<Callables...>;
```

Usage of overload_sequence

Introducing type_t: an extension of void_t

```cpp
// Returns the first type and ignores the others
template <class T, class...>
using type_t = T;

// Alternative definition of void_t
template <class... T>
using void_t = type_t<void, T...>
```

Going back to element_count

```cpp
int main(int argc, char* argv[]) {
    overload_sequence element_count(
        [](auto&& x) -> decltype(x.size()) {return std::forward<decltype(x)>(x).size();},
        [](auto&& x) -> type_t<std::size_t, decltype(std::begin(x)), decltype(std::end(x))> {
            return std::distance(
                std::begin(std::forward<decltype(x)>(x)),
                std::end(std::forward<decltype(x)>(x))
            );
        },
        [](auto&&) {return 1;}
    );
    element_count(std::vector({1, 2, 3, 4}));      // Uses .size
    element_count(std::forward_list({1, 2, 3}));  // Uses std::distance
    element_count(42);                            // Returns 1
    return 0;
}
```

An entire family of tools

Basic overload sequences: calls the callables as qualified at the call site

- `overload_sequence`: uses `is_invocable` to pick the right overload
- `overload_sequence_r`: `is_invocable_r` to pick the right overload
- `overload_sequence_nothrow`: `is_nothrow_invocable` to pick the right overload
- `overload_sequence_nothrow_r`: `is_nothrow_invocable_r` to pick the right overload

Forwarding overload sequences: calls the callables as qualified when constructed

- `forwarding_overload_sequence`: uses `is_invocable` to pick the right overload
- `forwarding_overload_sequence_r`: `is_invocable_r` to pick the right overload
- `forwarding_overload_sequence_nothrow`: `is_nothrow_invocable` to pick the right overload
- `forwarding_overload_sequence_nothrow_r`: `is_nothrow_invocable_r` to pick the right overload

Helpers

- `type_t`: generalization of `void_t`
- `overload_sequence_selector`: for the detection of the first compatible overload
- `invoke_result`: automatically works

And to wrap up: it works!

Solving the original problem!

```
1 overload_sequence(
2     [](auto x) -> decltype(x.size()) {return x.size();},
3     [](auto x){return x;},
4     [](auto x, auto y){return x + y;},
5     [](auto x, auto y, auto z){return x + y + z;}
6 )(10, 42); // returns 52
```

Custom overload sets

Custom overload sets

Goal

```
1 f(
2     [](int x) {return x * 10;},
3     [](double x) {return x / 10.;},
4     [](auto x) {return x;}
5 )(10.); // should return 1.
```

This one is easy!

Introducing overload_set

```cpp
// overload_set class definition
template <class... Callables>
struct overload_set: Callables...
{
    // Function call operator
    using Callables::operator()...;

    // Constructor
    template <
        class... F,
        class = std::enable_if_t<std::is_constructible_v<std::tuple<Callables...>, F...>>
    >
    constexpr overload_set(F&&... f)
    : Callables(std::forward<F>(f))... {}
};

// Deduction guide
template<class... Callables> overload_set(Callables...) -> overload_set<Callables...>;
```

A family of tools

- `overload_set`: picks the best overload depending on the qualification at the call site
- `forwarding_overload_set`: picks the best overload depending on the qualification at the construction
- `overload_set_selector`: returns the index of the function that is called in an overload set
- `invoke_result`: automatically works

Problem solved

Solving the original problem!

```
1 overload_set(
2     [](int x) {return x * 10;},
3     [](double x) {return x / 10.;},
4     [](auto x) {return x;}
5 )(10.); // returns 1.
```

Wait really?

What about

- non-lambda classes with an `operator()`: should work without problem
- functions: we cannot inherit from functions, however `overload_set` should wrap them in a class
- pointer to members function?
- references to qualified pointers to member objects?
- references to classes with an `operator()`?
- final classes with an `operator()`?
- unions with an `operator()`?

When taking all that into account...

`overload_set` becomes a total nightmare

To give a glimpse of why, we need to clarify two things

- How to make forwarding classes in C++17?
- What is a callable?

Forwarding classes

An introductory problem on forwarding classes

This is a forwarding function (C++11)

```
1 template <class F, class... Args>
2 constexpr auto call(F&& f, Args&&... args)
3 noexcept(std::forward<F>(f)(std::forward<Args>(args)...))
4 -> decltype(std::forward<F>(f)(std::forward<Args>(args)...)) {
5     return std::forward<F>(f)(std::forward<Args>(args)...);
6 }
7 call(f, 42, "Hello", "World");
```

This is a forwarding function (C++14)

```
1 template <class F, class... Args>
2 constexpr decltype(auto) call(F&& f, Args&&... args)
3 noexcept(std::forward<F>(f)(std::forward<Args>(args)...)) {
4     return std::forward<F>(f)(std::forward<Args>(args)...);
5 }
6 call(f, 42, "Hello", "World");
```

This is a forwarding function (C++17)

```
1 template <class F, class... Args>
2 constexpr decltype(auto) call(F&& f, Args&&... args)
3 noexcept (std::is_nothrow_invocable_v<F, Args...>) {
4     return std::invoke(std::forward<F>(f), std::forward<Args>(args...));
5 }
6 call(f, 42, "Hello", "World");
```

How to do the same with a class?

```
1 caller(f)(42, "Hello", "World");
```

The problem with forwarding and cvref qualifiers

Mixing forwarding references and template deduction guides

```
1  // Structure definition
2  template <class T>
3  struct structure {
4      // Type
5      using type = T;
6      // Data member
7      type value;
8      // Default constructor
9      constexpr structure() = default;
10     // Constructor
11     template <class U, class = std::enable_if_t<std::is_constructible_v<T, U&&>>>
12     constexpr structure(U&& x): value(std::forward<U>(x)) {}
13     // Get value
14     decltype(auto) operator()() {return value;}
15     decltype(auto) operator()(int) {return std::forward<T>(value);}
16 };
```

The type guessing game: sometimes hard to know... What are A_1 and A_2, as well as B_1 and B_2?

```
1  template <class U> structure(U&&) -> structure<U>;
2  using A1 = decltype(structure{42}());
3  using A2 = decltype(structure{42}(0));
```

```
1  template <class U> structure(U&&) -> structure<U&&>;
2  using B1 = decltype(structure{42}());
3  using B2 = decltype(structure{42}(0));
```

Trick: how to print the cv-ref-qualified name of a type?

Printing a type

```
1 template <class T>
2 void print_type() {
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
4 }
```

Printing a type: example

```
1 #include <iostream>
2 int main(int argc, char* argv[]) {
3     print_type<const volatile int&>();
4     return 0;
5 }
```

```
#> g++ print_type.cpp -o print_type && ./print_type
void print_type() [with T = const volatile int&]
#> clang++ print_type.cpp -o print_type && ./print_type
void print_type() [T = const volatile int &]
```

Qualifiers manipulation rules: introduction

cvref-qualifiers conversion rules: basics

`struct type {}; using T =`	`type`	`type&`	`type&&`	`const type`	`const type&`	`const type&&`
`T`	`type`	`type&`	`type&&`	`const type`	`const type&`	`const type&&`
`T&`	`type&`	`type&`	`type&`	`const type&`	`const type&`	`const type&`
`T&&`	`type&&`	`type&`	`type&&`	`const type&&`	`const type&`	`const type&&`
`const T`	`const type`	`type&`	`type&&`	`const type`	`const type&`	`const type&&`
`const T&`	`const type&`	`type&`	`type&`	`const type&`	`const type&`	`const type&`
`const T&&`	`const type&&`	`type&`	`type&&`	`const type&&`	`const type&`	`const type&&`

cvref-qualifiers conversion rules: `declval`, `forward` and `move`

`struct type {}; using T =`	`type`	`type&`	`type&&`	`const type`	`const type&`	`const type&&`
`decltype(std::declval<T>())`	`type&&`	`type&`	`type&&`	`const type&&`	`const type&`	`const type&&`
`decltype(std::forward<T>(std::declval<type>()))`	`type&&`	`type&`	`type&&`	`const type&&`	`const type&`	`const type&&`
`decltype(std::forward<T>(std::declval<type&>()))`	`type&&`	`type&`	`type&&`	`const type&&`	`const type&`	`const type&&`
`decltype(std::forward<T>(std::declval<type&&>()))`	`type&&`	`type&`	`type&&`	`const type&&`	`const type&`	`const type&&`
`decltype(std::forward<T>(std::declval<const type>()))`	•	•	•	`const type&&`	`const type&`	`const type&&`
`decltype(std::forward<T>(std::declval<const type&>()))`	•	•	•	`const type&&`	`const type&`	`const type&&`
`decltype(std::forward<T>(std::declval<const type&&>()))`	•	•	•	`const type&&`	`const type&`	`const type&&`
`decltype(std::move(std::declval<T>()))`	`type&&`	`type&&`	`type&&`	`const type&&`	`const type&&`	`const type&&`

Qualifiers manipulation rules: in functions and lambdas

cvref-qualifiers manipulation rules: functions

struct type {}; using T =	type	type&	type&&	const type	const type&	const type&&
template <class X> X f(X x); decltype(f(std::declval<T>()))	type	type	type	type	type	type
template <class X> X f(X& x); decltype(f(std::declval<T>()))	•	type	•	const type	const type	const type
template <class X> X f(X&& x); decltype(f(std::declval<T>()))	type	type&	type	const type	const type&	const type
template <class X> X f(const X x); decltype(f(std::declval<T>()))	type	type	type	type	type	type
template <class X> X f(const X& x); decltype(f(std::declval<T>()))	type	type	type	type	type	type
template <class X> X f(const X&& x); decltype(f(std::declval<T>()))	type	•	type	type	•	type

cvref-qualifiers manipulation rules: trailing return type

With Δ<T> = decltype(f(std::declval<T>()))

struct type {}; using T =	type	type&	type&&	const type	const type&	const type&&
template <class X> auto f(X x) -> decltype(x); Δ<T>	type	type	type	type	type	type
template <class X> auto f(X& x) -> decltype(x); Δ<T>	•	type&	•	const type&	const type&	const type&
template <class X> auto f(X&& x) -> decltype(x); Δ<T>	type&&	type&	type&&	const type&&	const type&	const type&&
template <class X> auto f(const X x) -> decltype(x); Δ<T>	const type	const type	const type	const type	const type	const type
template <class X> auto f(const X& x) -> decltype(x); Δ<T>	const type&	const type&	const type&	const type&	const type&	const type&
template <class X> auto f(const X&& x) -> decltype(x); Δ<T>	const type&&	•	const type&&	const type&&	•	const type&&
auto f = [](auto x) -> decltype(x){}; Δ<T>	type	type	type	type	type	type
auto f = [](auto& x) -> decltype(x){}; Δ<T>	•	type&	•	const type&	const type&	const type&
auto f = [](auto&& x) -> decltype(x){}; Δ<T>	type&&	type&	type&&	const type&&	const type&	const type&&
auto f = [](const auto x) -> decltype(x){}; Δ<T>	const type	const type	const type	const type	const type	const type
auto f = [](const auto& x) -> decltype(x){}; Δ<T>	const type&	const type&	const type&	const type&	const type&	const type&
auto f = [](const auto&& x) -> decltype(x){}; Δ<T>	const type&&	•	const type&&	const type&&	•	const type&&

Qualifiers manipulation rules: deduced return type and lambda capture

cvref-qualifiers manipulation rules: deduced return type

With `Δ<T> = decltype(f(std::declval<T>()))`

`struct type {}; using T =`	type	type&	type&&	const type	const type&	const type&&
`template <class X> auto f(){return X{};} Δ<T>`	type	•	type	type	type	type
`template <class X> auto f(X&& x){return x;} Δ<T>`	type	type	type	type	type	type
`template <class X> auto f(X&& x){return std::forward<X>(x);} Δ<T>`	type	type	type	type	type	type
`template <class X> decltype(auto) f(){return X{};} Δ<T>`	type	•	type&&	const type	const type&	const type&&
`template <class X> decltype(auto) f(X&& x){return x;} Δ<T>`	type&	type&	type&	const type&	const type&	const type&
`template <class X> decltype(auto) f(X&& x){return std::forward<X>(x);} Δ<T>`	type&&	type&	type&&	const type&&	const type&	const type&&

cvref-qualifiers manipulation rules: lambda capture

With `Δ<T> = decltype(f(std::declval<T>()))`

With `Λ(λ) = template <class T> decltype(auto) f(T&& x) {auto lambda = λ; return lambda();}`

`struct type {}; using T =`	type	type&	type&&	const type	const type&	const type&&
`Λ([=]() -> decltype(auto) {return x;}); Δ<T>`	const type	const type	const type	const type	const type	const type
`Λ([=]() mutable -> decltype(auto) {return x;}); Δ<T>`	type	type	type	const type	const type	const type
`Λ([&]() -> decltype(auto) {return x;}); Δ<T>`	type&	type&	type&	const type&	const type&	const type&
`Λ([&]() mutable -> decltype(auto) {return x;}); Δ<T>`	type&	type&	type&	const type&	const type&	const type&
`Λ([&]() -> decltype(auto) {return std::forward<T>(x);}); Δ<T>`	type&&	type&	type&&	const type&&	const type&	const type&&
`Λ([&]() mutable -> decltype(auto) {return std::forward<T>(x);}); Δ<T>`	type&&	type&	type&&	const type&&	const type&	const type&&

Qualifiers manipulation rules: class template argument deduction

cvref-qualifiers manipulation rules: class template argument deduction

With Ω<T> = `typename decltype(s(std::declval<T>()))::type`
With Γ(γ) = `template <class T> struct s {s(γ); using type = T;};`

`struct type {}; using T =`	type	type&	type&&	const type	const type&	const type&&
Γ(T); Ω<T>	type	type	type	type	type	type
Γ(T&); Ω<T>	•	type	•	const type	const type	const type
Γ(T&&); Ω<T>	type	•	type	const type	•	const type
Γ(const T); Ω<T>	type	type	type	type	type	type
Γ(const T&); Ω<T>	type	type	type	type	type	type
Γ(const T&&); Ω<T>	type	•	type	type	•	type

cvref-qualifiers manipulation rules: class template argument deduction with user-defined deduction guides

With Ω<T> = `typename decltype(s(std::declval<T>()))::type`
With Φ(ϕ) = `template <class T> struct s {template <class U> s(ϕ); using type = T;}; template <class U> s(ϕ) -> s<U>;`

`struct type {}; using T =`	type	type&	type&&	const type	const type&	const type&&
Φ(U); Ω<T>	type	type	type	type	type	type
Φ(U&); Ω<T>	•	type	•	const type	const type	const type
Φ(U&&); Ω<T>	type	type&	type	const type	const type&	const type
Φ(const U); Ω<T>	type	type	type	type	type	type
Φ(const U&); Ω<T>	type	type	type	type	type	type
Φ(const U&&); Ω<T>	type	•	type	type	•	type

After all that, some equivalences

The unqualified type

```
1  template <class X> X f0(X x);                                  using T0 = decltype(f0(std::declval<T>()));
2  template <class X> X f1(const X x);                            using T1 = decltype(f1(std::declval<T>()));
3  template <class X> X f2(const X& x);                           using T2 = decltype(f2(std::declval<T>()));
4  template <class X> auto f3(X x) -> decltype(x);                using T3 = decltype(f3(std::declval<T>()));
5  auto f4 = [](auto x) -> decltype(x){};                         using T4 = decltype(f4(std::declval<T>()));
6  template <class X> auto f5(X&& x){return x;}                   using T5 = decltype(f5(std::declval<T>()));
7  template <class X> auto f6(X&& x){return std::forward<X>(x);}using T6 = decltype(f6(std::declval<T>()));
8  template <class T> struct s7 {s7(T); using type = T;}          using T7 = typename decltype(s7(std::declval<T>()))::type;
9  template <class T> struct s8 {s8(const T); using type = T;}    using T8 = typename decltype(s8(std::declval<T>()))::type;
10 template <class T> struct s9 {s9(const T&); using type = T;} using T9 = typename decltype(s9(std::declval<T>()))::type;
```

struct type {}; using T =	type	type&	type&&	const type	const type&	const type&&
T0; T1; T2; T3; T4; T5; T6; T7; T8; T9;	type	type	type	type	type	type

Template argument deduction and forwarding references

```
1  template <class X> X f(X&& x);
2  using T0 = decltype(f(std::declval<T>()));
3  template <class T> struct s {template <class U> s(U&&); using type = T;};  template <class U> s(U&&) -> s<U>;
4  using T1 = typename decltype(s(std::declval<T>()))::type;
```

struct type {}; using T =	type	type&	type&&	const type	const type&	const type&&
T0; T1;	type	type&	type	const type	const type&	const type

Equivalences for forwarding references

Forwarding references

```cpp
// Preamble
struct type {};
using T = /*cvref-qualified type*/;
// Functions
template <class X> auto f4(X&& x) -> decltype(x);
auto f5 = [](auto&& x) -> decltype(x){};
template <class X> decltype(auto) f6(X&& x){return std::forward<X>(x);}
template <class T> decltype(auto) f7(T&& x) {
    auto lambda = [&]() -> decltype(auto) {return std::forward<T>(x);};
    return lambda();
}
template <class T> decltype(auto) f8(T&& x) {
    auto lambda = [&]() mutable -> decltype(auto) {return std::forward<T>(x);};
    return lambda();
}
// Types
using T0 = decltype(std::declval<T>());
using T1 = decltype(std::forward<T>(std::declval<type>()));
using T2 = decltype(std::forward<T>(std::declval<type&>()));
using T3 = decltype(std::forward<T>(std::declval<type&&>()));
using T4 = decltype(f4(std::declval<T>()));
using T5 = decltype(f5(std::declval<T>()));
using T6 = decltype(f6(std::declval<T>()));
using T7 = decltype(f7(std::declval<T>()));
using T8 = decltype(f8(std::declval<T>()));
```

struct type {}; using T =	type	type&	type&&	const type	const type&	const type&&
T0; T1; T2; T3; T4; T5; T6; T7; T8;	type&&	type&	type&&	const type&&	const type&	const type&&

Coming back to the question about qualifiers

Mixing forwarding references and template deduction guides

```cpp
// Structure definition
template <class T>
struct structure {
    // Type
    using type = T;
    // Data member
    type value;
    // Default constructor
    constexpr structure() = default;
    // Constructor
    template <class U, class = std::enable_if_t<std::is_constructible_v<T, U&&>>>
    constexpr structure(U&& x): value(std::forward<U>(x)) {}
    // Get value
    decltype(auto) operator()() {return value;}
    decltype(auto) operator()(int) {return std::forward<T>(value);}
};
```

The type guessing game: what are A_1 and A_2, as well as B_1 and B_2? $\Rightarrow$ Now we know...

```cpp
template <class U> structure(U&&) -> structure<U>;
using A1 = decltype(structure{42}());  // -> int
using A2 = decltype(structure{42}(0)); // -> int&&
```

```cpp
template <class U> structure(U&&) -> structure<U&&>;
using B1 = decltype(structure{42}());  // -> int&
using B2 = decltype(structure{42}(0)); // -> int&&
```

Coming back to forwarding classes

Mixing forwarding references and template deduction guides

```cpp
template <class T> struct structure {
    // Type
    using type = T;
    // Data member
    type value;
    // Default constructor
    constexpr structure() = default;
    // Constructor
    template <class U, class = std::enable_if_t<std::is_constructible_v<T, U&&>>>
    constexpr structure(U&& x): value(std::forward<U>(x)) {}
    // Get value
    decltype(auto) operator()() {return std::forward<T>(value);}
};
```

Option 1

```cpp
template <class U> structure1(U&&) -> structure1<U>;
```

Option 2

```cpp
template <class U> structure2(U&&) -> structure2<U&&>;
```

Remark

$\forall T$, `decltype(structure1{std::declval<T>()}())` $\iff$ `decltype(structure2{std::declval<T>()}())`

A problem of lifetime? Let's use a reporter

Introducing the reporter class: https://github.com/vreverdy/reporter

```cpp
// Reporter class definition
template <class T = std::size_t> class reporter {
    // Assertions
    public: static_assert(std::is_constructible_v<T, std::size_t>, "");
    // Types
    using value_type = T;
    using size_type = std::size_t;
    using pointer = value_type*;
    // Lifecycle
    reporter();
    explicit reporter(std::ostream& stream);
    reporter(const reporter& other);
    reporter(reporter&& other);
    ~reporter();
    // Assignment
    reporter& operator=(const reporter& other);
    reporter& operator=(reporter&& other);
    // Function call operators
    template <class... Args> void operator()(Args&&...);
    template <class... Args> void operator()(Args&&...) const;
    template <class... Args> void operator()(Args&&...) volatile;
    template <class... Args> void operator()(Args&&...) const volatile;
    void operator()() &;
    void operator()() const&;
    void operator()() volatile&;
    void operator()() const volatile&;
    void operator()() &&;
    void operator()() const&&;
    void operator()() volatile&&;
    void operator()() const volatile&&;
};
```

About the reporter class

Example of use

```
1  reporter r1;
2  reporter r2(r1);
3  reporter r3(r2);
4  reporter r4(std::move(r1));
5  reporter r5;
6  r1();
7  r2();
8  r3();
9  r4();
10 r5();
```

```
[type_id, thread_id, id, memory_address, value]: operation
[39895, 64801, 00001, 0x00F5DC20, 00001]: reporter::reporter()
[39895, 64801, 00002, 0x00F5DD20, 00001]: reporter::reporter(const reporter&)
[39895, 64801, 00003, 0x00F5DE20, 00001]: reporter::reporter(const reporter&)
[39895, 64801, 00004, 0x00F5DC20, 00001]: reporter::reporter(reporter&&)
[39895, 64801, 00005, 0x00F5DF20, 00005]: reporter::reporter()
[39895, 64801, 00001, 0x00000000, 00000]: void reporter::operator()() &
[39895, 64801, 00002, 0x00F5DD20, 00001]: void reporter::operator()() &
[39895, 64801, 00003, 0x00F5DE20, 00001]: void reporter::operator()() &
[39895, 64801, 00004, 0x00F5DC20, 00001]: void reporter::operator()() &
[39895, 64801, 00005, 0x00F5DF20, 00005]: void reporter::operator()() &
[39895, 64801, 00005, 0x00F5DF20, 00005]: reporter::~reporter()
[39895, 64801, 00004, 0x00F5DC20, 00001]: reporter::~reporter()
[39895, 64801, 00003, 0x00F5DE20, 00001]: reporter::~reporter()
[39895, 64801, 00002, 0x00F5DD20, 00001]: reporter::~reporter()
[39895, 64801, 00001, 0x00000000, 00000]: reporter::~reporter()
```

And finally, the conclusion on forwarding classes

Option 1

```
1 template <class U> structure1(U&&) -> structure1<U>;
2 structure1 s1{reporter<>{}}; s1()();
3 structure1{reporter<>{}}();
```

```
[39895, 36535, 00001, 0x02374C20, 00001]: reporter::reporter()
[39895, 36535, 00002, 0x02374C20, 00001]: reporter::reporter(reporter&&)
[39895, 36535, 00001, 0x00000000, 00000]: reporter::~reporter()
[39895, 36535, 00002, 0x02374C20, 00001]: void reporter::operator()() &&
[39895, 36535, 00003, 0x02374EE0, 00003]: reporter::reporter()
[39895, 36535, 00004, 0x02374EE0, 00003]: reporter::reporter(reporter&&)
[39895, 36535, 00004, 0x02374EE0, 00003]: void reporter::operator()() &&
[39895, 36535, 00004, 0x02374EE0, 00003]: reporter::~reporter()
[39895, 36535, 00003, 0x00000000, 00000]: reporter::~reporter()
[39895, 36535, 00002, 0x02374C20, 00001]: reporter::~reporter()
```

Option 2

```
1 template <class U> structure2(U&&) -> structure2<U&&>;
2 structure2 s2{reporter<>{}}; s2()();
3 structure2{reporter<>{}}();
```

```
[39895, 36535, 00001, 0x00DFFC20, 00001]: reporter::reporter()
[39895, 36535, 00001, 0x00DFFC20, 00001]: reporter::~reporter()
[39895, 36535, 00001, 0x00000000, 00000]: void reporter::operator()() &&
[39895, 36535, 00002, 0x00DFFEC0, 00002]: reporter::reporter()
[39895, 36535, 00002, 0x00DFFEC0, 00002]: void reporter::operator()() &&
[39895, 36535, 00002, 0x00DFFEC0, 00002]: reporter::~reporter()
```

Conclusion

The way to go with forwarding classes is to follow the option 1

```
1 template <class U> structure(U&&) -> structure<U>;
```

What callables are

What is a callable?

The thing with `std::invoke`

```
1 template <class F, class... Args>
2 std::invoke_result_t<F, Args...> invoke(F&& f, Args&&... args) noexcept(/*...*/);
```

Invoke the `Callable` object `f` with the parameters `args`.

Definition of a `Callable` in the C++ standard and on cppreference

23.14.2 Definitions [func.def]

1 The following definitions apply to this Clause:

2 A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

3 A *callable type* is a function object type (23.14) or a pointer to member.

4 A *callable object* is an object of a callable type.

5 A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.

6 A *call wrapper* is an object of a call wrapper type.

7 A *target object* is the callable object held by a call wrapper.

C++ named requirements: *Callable*

A ***Callable*** type is a type for which the INVOKE operation (used by, e.g., `std::function`, `std::bind`, and `std::thread::thread`) is applicable. This operation may be performed explicitly using the library function `std::invoke`. (since C++17)

Requirements

The type T satisfies *Callable* if

Given

- f, an object of type T
- ArgTypes, suitable list of argument types
- R, suitable return type

The following expressions must be valid:

Expression	Requirements
`INVOKE<R>(f, std::declval<ArgTypes>()...)`	the expression is well-formed in unevaluated context

Question

But in practice, what is a callable? On what kind of types one can call `invoke`? How many categories of callable types exist?

What is a callable?

Functions

```
1 R(Args...);                    // Example: int f(int x, int y) {return x + y;}
1 R(Args..., ...);               // Variadic arguments, can also be noted R(Args.....)
1 R(Args...) noexcept;           // Since C++17: noexcept as part of the function signature
2 R(Args..., ...) noexcept;      // Since C++17: noexcept as part of the function signature
```

Qualified functions?

```
1 int f(int x, int y) const;
```

```
#> g++ const_function_test.cpp -o const_function_test
error: non-member function 'int f(int, int)' cannot have cv-qualifier
```

Qualified functions

```
1 // Default version
2 template <class T>
3 struct member_type;
4
5 // Specialization
6 template <class T, class C>
7 struct member_type<T C::*> {
8     using type = T;
9 };
```

```
1 struct structure {int f(int x, int y) const;};
2 using const_function_type = typename member_type<decltype(&structure::f)>::type;
```

An overview of qualified versions

Functions (48 types)

```
1  // Basic versions (4 types)
2  R(Args...);
3  R(Args..., ...);
4  R(Args...) noexcept;
5  R(Args..., ...) noexcept;
6
7  // Qualified versions (11 types)
8  R(Args...) &;
9  R(Args...) &&;
10 R(Args...) const;
11 R(Args...) const &;
12 R(Args...) const &&;
13 R(Args...) volatile;
14 R(Args...) volatile &;
15 R(Args...) volatile &&;
16 R(Args...) const volatile;
17 R(Args...) const volatile &;
18 R(Args...) const volatile &&;
19
20 // Variadic versions (11 types)
21 R(Args..., ...) &;
22 R(Args..., ...) &&;
23 R(Args..., ...) const;
24 R(Args..., ...) const &;
25 R(Args..., ...) const &&;
26 R(Args..., ...) volatile;
27 R(Args..., ...) volatile &;
28 R(Args..., ...) volatile &&;
29 R(Args..., ...) const volatile;
30 R(Args..., ...) const volatile &;
31 R(Args..., ...) const volatile &&;
32
33 // Noexcept versions (11 types)
34 R(Args...) & noexcept;
35 R(Args...) && noexcept;
36 R(Args...) const noexcept;
37 R(Args...) const & noexcept;
38 R(Args...) && noexcept;
39 R(Args...) volatile noexcept;
40 R(Args...) volatile & noexcept;
41 R(Args...) volatile && noexcept;
42 R(Args...) const volatile noexcept;
43 R(Args...) const volatile & noexcept;
44 R(Args...) const volatile && noexcept;
45
46 // Variadic noexcept versions (11 types)
47 R(Args..., ...) & noexcept;
48 R(Args..., ...) && noexcept;
49 R(Args..., ...) const noexcept;
50 R(Args..., ...) const & noexcept;
51 R(Args..., ...) && noexcept;
52 R(Args..., ...) volatile noexcept;
53 R(Args..., ...) volatile & noexcept;
54 R(Args..., ...) volatile && noexcept;
55 R(Args..., ...) const volatile noexcept;
56 R(Args..., ...) const volatile & noexcept;
57 R(Args..., ...) const volatile && noexcept;
```

Function lvalue-references (48 types)

```
1  // Basic versions (4 types)
2  R(&)(Args...);
3  R(&)(Args..., ...);
4  R(&)(Args...) noexcept;
5  R(&)(Args..., ...) noexcept;
6
7  // Qualified versions (11 types)
8  R(&)(Args...) &;
9  R(&)(Args...) &&;
10 R(&)(Args...) const;
11 R(&)(Args...) const &;
12 R(&)(Args...) const &&;
13 R(&)(Args...) volatile;
14 R(&)(Args...) volatile &;
15 R(&)(Args...) volatile &&;
16 R(&)(Args...) const volatile;
17 R(&)(Args...) const volatile &;
18 R(&)(Args...) const volatile &&;
19
20 // Variadic versions (11 types)
21 R(&)(Args..., ...) &;
22 R(&)(Args..., ...) &&;
23 R(&)(Args..., ...) const;
24 R(&)(Args..., ...) const &;
25 R(&)(Args..., ...) const &&;
26 R(&)(Args..., ...) volatile;
27 R(&)(Args..., ...) volatile &;
28 R(&)(Args..., ...) volatile &&;
29 R(&)(Args..., ...) const volatile;
30 R(&)(Args..., ...) const volatile &;
31 R(&)(Args..., ...) const volatile &&;
32
33 // Noexcept versions (11 types)
34 R(&)(Args...) & noexcept;
35 R(&)(Args...) && noexcept;
36 R(&)(Args...) const noexcept;
37 R(&)(Args...) const & noexcept;
38 R(&)(Args...) && noexcept;
39 R(&)(Args...) volatile noexcept;
40 R(&)(Args...) volatile & noexcept;
41 R(&)(Args...) volatile && noexcept;
42 R(&)(Args...) const volatile noexcept;
43 R(&)(Args...) const volatile & noexcept;
44 R(&)(Args...) const volatile && noexcept;
45
46 // Variadic noexcept versions (11 types)
47 R(&)(Args..., ...) & noexcept;
48 R(&)(Args..., ...) && noexcept;
49 R(&)(Args..., ...) const noexcept;
50 R(&)(Args..., ...) const & noexcept;
51 R(&)(Args..., ...) && noexcept;
52 R(&)(Args..., ...) volatile noexcept;
53 R(&)(Args..., ...) volatile & noexcept;
54 R(&)(Args..., ...) volatile && noexcept;
55 R(&)(Args..., ...) const volatile noexcept;
56 R(&)(Args..., ...) const volatile & noexcept;
57 R(&)(Args..., ...) const volatile && noexcept;
```

Function rvalue-references (48 types)

```
1  // Basic versions (4 types)
2  R(&&)(Args...);
3  R(&&)(Args..., ...);
4  R(&&)(Args...) noexcept;
5  R(&&)(Args..., ...) noexcept;
6
7  // Qualified versions (11 types)
8  R(&&)(Args...) &;
9  R(&&)(Args...) &&;
10 R(&&)(Args...) const;
11 R(&&)(Args...) const &;
12 R(&&)(Args...) const &&;
13 R(&&)(Args...) volatile;
14 R(&&)(Args...) volatile &;
15 R(&&)(Args...) volatile &&;
16 R(&&)(Args...) const volatile;
17 R(&&)(Args...) const volatile &;
18 R(&&)(Args...) const volatile &&;
19
20 // Variadic versions (11 types)
21 R(&&)(Args..., ...) &;
22 R(&&)(Args..., ...) &&;
23 R(&&)(Args..., ...) const;
24 R(&&)(Args..., ...) const &;
25 R(&&)(Args..., ...) const &&;
26 R(&&)(Args..., ...) volatile;
27 R(&&)(Args..., ...) volatile &;
28 R(&&)(Args..., ...) volatile &&;
29 R(&&)(Args..., ...) const volatile;
30 R(&&)(Args..., ...) const volatile &;
31 R(&&)(Args..., ...) const volatile &&;
32
33 // Noexcept versions (11 types)
34 R(&&)(Args...) & noexcept;
35 R(&&)(Args...) && noexcept;
36 R(&&)(Args...) const noexcept;
37 R(&&)(Args...) const & noexcept;
38 R(&&)(Args...) && noexcept;
39 R(&&)(Args...) volatile noexcept;
40 R(&&)(Args...) volatile & noexcept;
41 R(&&)(Args...) volatile && noexcept;
42 R(&&)(Args...) const volatile noexcept;
43 R(&&)(Args...) const volatile & noexcept;
44 R(&&)(Args...) const volatile && noexcept;
45
46 // Variadic noexcept versions (11 types)
47 R(&&)(Args..., ...) & noexcept;
48 R(&&)(Args..., ...) && noexcept;
49 R(&&)(Args..., ...) const noexcept;
50 R(&&)(Args..., ...) const & noexcept;
51 R(&&)(Args..., ...) && noexcept;
52 R(&&)(Args..., ...) volatile noexcept;
53 R(&&)(Args..., ...) volatile & noexcept;
54 R(&&)(Args..., ...) volatile && noexcept;
55 R(&&)(Args..., ...) const volatile noexcept;
56 R(&&)(Args..., ...) const volatile & noexcept;
57 R(&&)(Args..., ...) const volatile && noexcept;
```

Function pointers (48 types)

```
1  // Basic versions (4 types)
2  R(*)(Args...);
3  R(*)(Args..., ...);
4  R(*)(Args...) noexcept;
5  R(*)(Args..., ...) noexcept;
6
7  // Qualified versions (11 types)
8  R(*)(Args...) &;
9  R(*)(Args...) &&;
10 R(*)(Args...) const;
11 R(*)(Args...) const &;
12 R(*)(Args...) const &&;
13 R(*)(Args...) volatile;
14 R(*)(Args...) volatile &;
15 R(*)(Args...) volatile &&;
16 R(*)(Args...) const volatile;
17 R(*)(Args...) const volatile &;
18 R(*)(Args...) const volatile &&;
19
20 // Variadic versions (11 types)
21 R(*)(Args..., ...) &;
22 R(*)(Args..., ...) &&;
23 R(*)(Args..., ...) const;
24 R(*)(Args..., ...) const &;
25 R(*)(Args..., ...) const &&;
26 R(*)(Args..., ...) volatile;
27 R(*)(Args..., ...) volatile &;
28 R(*)(Args..., ...) volatile &&;
29 R(*)(Args..., ...) const volatile;
30 R(*)(Args..., ...) const volatile &;
31 R(*)(Args..., ...) const volatile &&;
32
33 // Noexcept versions (11 types)
34 R(*)(Args...) & noexcept;
35 R(*)(Args...) && noexcept;
36 R(*)(Args...) const noexcept;
37 R(*)(Args...) const & noexcept;
38 R(*)(Args...) && noexcept;
39 R(*)(Args...) volatile noexcept;
40 R(*)(Args...) volatile & noexcept;
41 R(*)(Args...) volatile && noexcept;
42 R(*)(Args...) const volatile noexcept;
43 R(*)(Args...) const volatile & noexcept;
44 R(*)(Args...) const volatile && noexcept;
45
46 // Variadic noexcept versions (11 types)
47 R(*)(Args..., ...) & noexcept;
48 R(*)(Args..., ...) && noexcept;
49 R(*)(Args..., ...) const noexcept;
50 R(*)(Args..., ...) const & noexcept;
51 R(*)(Args..., ...) && noexcept;
52 R(*)(Args..., ...) volatile noexcept;
53 R(*)(Args..., ...) volatile & noexcept;
54 R(*)(Args..., ...) volatile && noexcept;
55 R(*)(Args..., ...) const volatile noexcept;
56 R(*)(Args..., ...) const volatile & noexcept;
57 R(*)(Args..., ...) const volatile && noexcept;
```

An overview of callable types: functions

Functions

- functions: `R(Args...)` [48 types]
- function lvalue-references: `R(&)(Args...)` [48 types]
- function rvalue-references: `R(&&)(Args...)` [48 types]
- function pointers: `R(*)(Args...)` [48 types]
- function const pointers: `R(* const)(Args...)` [48 types]
- function volatile pointers: `R(* volatile)(Args...)` [48 types]
- function const volatile pointers: `R(* const volatile)(Args...)` [48 types]
- function pointer lvalue-references: `R(*&)(Args...)` [48 types]
- function const pointer lvalue-references: `R(* const &)(Args...)` [48 types]
- function volatile pointer lvalue-references: `R(* volatile &)(Args...)` [48 types]
- function const volatile pointer lvalue-references: `R(* const volatile &)(Args...)` [48 types]
- function pointer rvalue-references: `R(*&&)(Args...)` [48 types]
- function const pointer rvalue-references: `R(* const &&)(Args...)` [48 types]
- function volatile pointer rvalue-references: `R(* volatile &&)(Args...)` [48 types]
- function const volatile pointer rvalue-references: `R(* const volatile &&)(Args...)` [48 types]

An overview of callable types: member function pointers

Member function pointers

- member function pointers: `R(C::*)(Args...)` [48 types]
- member function const pointers: `R(C::* const)(Args...)` [48 types]
- member function volatile pointers: `R(C::* volatile)(Args...)` [48 types]
- member function const volatile pointers: `R(C::* const volatile)(Args...)` [48 types]
- member function pointer lvalue-references: `R(C::*&)(Args...)` [48 types]
- member function const pointer lvalue-references: `R(C::* const &)(Args...)` [48 types]
- member function volatile pointer lvalue-references: `R(C::* volatile &)(Args...)` [48 types]
- member function const volatile pointer lvalue-references: `R(C::* const volatile &)(Args...)` [48 types]
- member function pointer rvalue-references: `R(C::*&&)(Args...)` [48 types]
- member function const pointer rvalue-references: `R(C::* const &&)(Args...)` [48 types]
- member function volatile pointer rvalue-references: `R(C::* volatile &&)(Args...)` [48 types]
- member function const volatile pointer rvalue-references: `R(C::* const volatile &&)(Args...)` [48]

How many types so far?

So far $15 \times 48 = 720$ function types and $12 \times 48 = 576$ member function pointer types.

An overview of callable types: member object pointers

Member object pointers

- member object pointers: `T C::*` [12 types] (`T C::*`, `T& C::*`, `T&& C::*`, `const T C::*`, ...)
- member object const pointers: `T C::* const` [12 types]
- member object volatile pointers: `T C::* volatile` [12 types]
- member object const volatile pointers: `T C::* const volatile` [12 types]
- member object pointer lvalue-references: `T C::*&` [12 types
- member object const pointer lvalue-references: `T C::* const &` [12 types]
- member object volatile pointer lvalue-references: `T C::* volatile &` [12 types]
- member object const volatile pointer lvalue-references: `T C::* const volatile &` [12 types]
- member object pointer rvalue-references: `T C::*&&` [12 types]
- member object const pointer rvalue-references: `T C::* const &&` [12 types]
- member object volatile pointer rvalue-references: `T C::* volatile &&` [12 types]
- member object const volatile pointer rvalue-references: `T C::* const volatile &&` [12 types]

Subtotal

$12 \times 12 = 144$ member object pointer types.

An overview of callable types: classes

Class types with a function call operator

- closure types: lambdas, qualified lambdas, and reference to possibly qualified lambdas
- `class` with at least one public, possibly inherited, `operator()` member, including cv-ref qualified versions
- `struct` with at least one public, possibly inherited, `operator()` member, including cv-ref qualified versions
- `union` with at least one public, `operator()` member, including cv-ref qualified versions

Subtotal

12 or $4 \times 12 = 48$ types depending on the counting method.

Total

~ 1500 types ($720 + 576 + 144 + 48 = 1488$)

Examples of invocation

Examples of invocation: function

```
1 double add(double x, double y) {return x + y;}
2 std::invoke(add, 4, 2);
```

```
1 using add_lref_t = double(&)(double, double);
2 add_lref_t add_lref = add;
3 std::invoke(add_lref, 4, 2);
```

```
1 using add_rref_t = double(&&)(double, double);
2 add_rref_t add_rref = add;
3 std::invoke(add_rref, 4, 2);
```

```
1 using add_ptr_t = double(*)(double, double);
2 add_ptr_t add_ptr = add;
3 std::invoke(add_ptr, 4, 2);
```

```
1 using add_misc_t = double(* const volatile &&)(double, double);
2 add_full_t add_misc = add;
3 std::invoke(add_misc, 4, 2);
```

With auto?

```
1 auto x = add;
2 // -> double(*)(double, double)
3
4 decltype(auto) y = add;
5 // -> ?
```

```
#> g++ -std=c++17 auto_function.cpp
error: initializer for
'decltype(auto) y' has function
type (did you forget the '()' ?)
```

Examples of invocation: class

```
1 struct adder_t {double operator()(double x, double y) const {return x + y;}};
2 adder_t adder;
3 std::invoke(adder, 4, 2);
```

```
1 using adder_ptr_t = double(adder::*)(double, double) const;
2 adder_ptr_t adder_ptr = &adder::operator();
3 std::invoke(adder_ptr, adder, 4, 2);
```

```
1 using adder_misc_t = double(adder::* const volatile &&)(double, double) const;
2 adder_full_t adder_misc = &adder::operator();
3 std::invoke(adder_misc, adder, 4, 2);
```

Would C++ be that simple?

Conversion rules

```
double add(double x, double y) {return x + y;}
struct adder {double operator()(double x, double y) const {return x + y;}};
```

```
template <class T>
struct wrapper {
    // Type and value
    using type = T;
    type value;
    // Constructor
    template <class U, class = std::enable_if_t<std::is_constructible_v<T, U&&>>>
    constexpr wrapper(U&& f): value(std::forward<U>(f)) {}
    // Converter
    constexpr operator T() {return value;}
    // Call
    template <class... Args>
    constexpr decltype(auto) call(Args&&... args) {
        return std::invoke(value, std::forward<Args>(args)...);
    }
};
```

```
// A wrapper with a conversion operator to a function pointer, reference etc...
std::is_invocable_v<wrapper<double(*)(double, double)>, double, double>;                                  // true
std::is_invocable_v<wrapper<double(&)(double, double)>, double, double>;                                  // true
std::is_invocable_v<wrapper<double(* const volatile &&)(double, double)>, double, double>;               // true
// A wrapper with a conversion operator to a class with an operator()
std::is_invocable_v<wrapper<adder>, double, double>;                                                      // false
// A wrapper over a class that has an operator()
std::is_invocable_v<decltype(&wrapper<adder>::value), wrapper<adder>>;                                    // true
std::is_invocable_v<decltype(&wrapper<adder>::value), wrapper<adder>, double, double>;                    // false
std::is_invocable_v<decltype(&wrapper<adder>::call<double, double>), wrapper<adder>, double, double>;     // true
```

An overview of callable types: synopsis

Summary

A callable type can be:

- a possibly cvref-qualified function, reference to it, possibly cvref-qualified pointer or pointer reference to it
- a possibly cvref-qualified pointer to a possibly cvref-qualified member function, or reference to it
- a possibly cvref-qualified pointer to a possibly cvref-qualified member object, or reference to it
- a possibly cvref-qualified object of closure, `class`, `struct`, `union` type with at least 1 public `operator()`, or reference to it
- a possibly cvref-qualified object of `class`, `struct`, `union` type with at least one public non-explicit conversion operator to a possibly cvref-qualified pointer, pointer reference or reference to a possibly cvref-qualified function, or reference to it

for a total of more than 1500 forms of types

The standard formally defines

- closure types, see `[expr.prim.lambda.closure]`
- classes with at least one public `operator()` (the standard does not give them a name)
- function pointers
- function objects, which include all the preceding categories, including function pointers, see `[function.objects]`
- member pointers
- callables, as any type for which there exist arguments so that `invoke` can be called

Custom overload sets strike back

Custom overload sets strike back (reminder)

What about

- non-lambda classes with an `operator()`: should work without problem
- functions: we cannot inherit from functions, however `overload_set` should wrap them in a class
- pointer to members function?
- references to qualified pointers to member objects?
- references to classes with an `operator()`?
- final classes with an `operator()`?
- unions with an `operator()`?

When taking all that into account...

`overload_set` becomes a total nightmare

To give a glimpse of why, we need to clarify two things

- How to make forwarding classes in C++17?
- What is a callable?

Considering the details on forwarding classes and callables...

It is possible to build a set of general-purpose type traits to help making `overload_set`

- Details in P1016R0: A few additional type manipulation utilities
- Implementation available at https://github.com/vreverdy/type-utilities

Pointers removal

```
remove_all_pointers
```

Qualifiers copy

```
copy_const
copy_volatile
copy_cv
copy_reference
copy_signedness
copy_extent
copy_all_extents
copy_pointer
copy_all_pointers
copy_cvref
```

Qualifiers cloning

```
clone_const
clone_volatile
clone_cv
clone_reference
clone_extent
clone_all_extents
clone_pointer
clone_all_pointers
clone_cvref
```

Inheritance

```
empty_base
is_instantiable
is_inheritable
inherit_if
```

Callables categorization

```
is_closure
is_functor
is_function_object
is_callable
```

Miscellaneous helpers

```
index_constant
type_t
false_v
true_v
```

Wrapping up on `overload_set`

With the help of the traits described in P1016R0: A few additional type manipulation utilities

It becomes possible to write an `overload_set` class that works for all cases...

...except...

For `final` classes and `union`s: they are not inheritable and cannot be wrapped to mimic the desired behaviour.

Exercise left to the developer

Provides an implementation of `overload_set` working with all the 1500 categories of callables except `final` classes and `union`s. Good luck. Have good nightmares. (it is doable we promise)

Related work

C++ generic overload functio, Vincente J. Botet Escribá.

A new hope

The upcoming C++ reflection facilities may help solving the remaining problems.

Playing with types

Playing with types

Goal

```
1 using type = decltype(some_tool(
2     some_other_tool<int>(/*...*/),
3     some_other_tool<std::vector<int>>(/*...*/),
4     some_other_tool<double>(/*...*/)
5 ));
```

A possible solution

Strategy

Using again the same kind of techniques as previously illustrated in `is_valid` and `overload_sequence`.

Suggestion of implementation

```
1  // Invalid option
2  struct invalid_option {};
3
4  // Type option
5  template <class T, class... F>
6  constexpr std::conditional_t<(std::is_invocable_v<F, T> && ...), T, invalid_option>
7  type_option(F&&...);
8
9  // Type selector
10 template <class... T>
11 constexpr std::tuple_element_t<
12     tuple_index_v<
13         std::true_type,
14         std::tuple<
15             std::conditional_t<
16                 std::is_same_v<T, invalid_option>,
17                 std::false_type,
18                 std::true_type
19             >...
20         >
21     >,
22     std::tuple<T...>
23 > type_selector(T&&...);
```

Usage

Example of use: returns the first type associated to a valid expression

```cpp
int main(int argc, char* argv[]) {
    auto lambda = [](auto x) -> decltype(x.size()){};
    using type = decltype(type_selector(
        type_option<int>(lambda),
        type_option<std::vector<int>>(lambda)
    ));
    print_type<type>();
    return 0;
}
```

```
#> g++ -std=c++17 type_selector.cpp -o type_selector && ./type_selector
void print_type() [T = std::vector<int, std::allocator<int> >]
```

Unevaluated contexts

It is not yet possible to use this technique with the syntax `type_option<T>([]() -> decltype(/*...*/) {})`. This would require the possibility of declaring lambdas in unevaluated contexts, and it currently forbidden.

So we have a way to select types without too much boilerplate code

Solving the original problem!

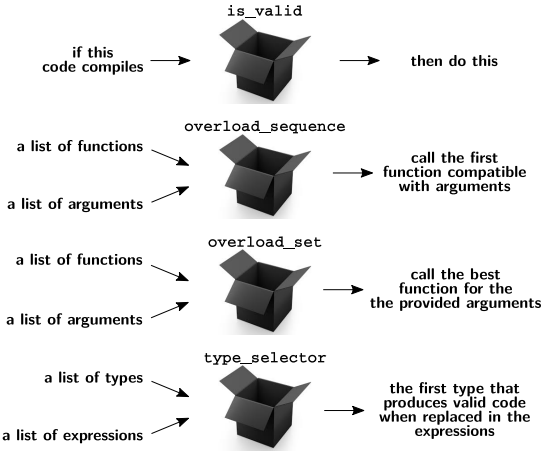
```
auto lambda = [](auto x) -> decltype(x.size()){};
using type = decltype(type_selector(
    type_option<int>(lambda),
    type_option<std::vector<int>>(lambda),
    type_option<double>(lambda)
)); // type is equal to std::vector<int>
```

Conclusion

Conclusion

Main question

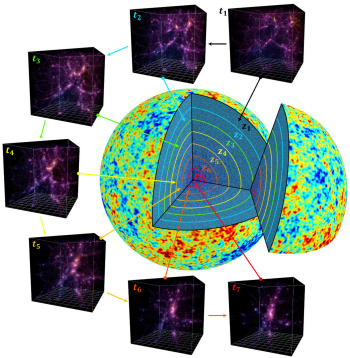
How to improve code genericity in C++17 without paying the price of code complexity and readability?

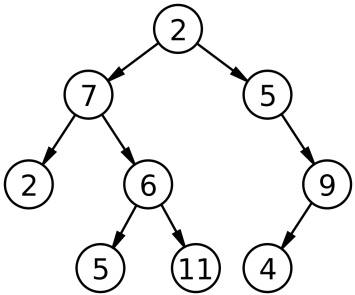


Conclusion

Coming back to the beginning (of time)

Great tools to write high performance generic code and branch on the right function or type depending on, for example, algorithmic complexity.





Summary of tools

Qualifiers copy

```
copy_const
copy_volatile
copy_cv
copy_reference
copy_signedness
copy_extent
copy_all_extents
copy_pointer
copy_all_pointers
copy_cvref
```

Qualifiers cloning

```
clone_const
clone_volatile
clone_cv
clone_reference
clone_extent
clone_all_extents
clone_pointer
clone_all_pointers
clone_cvref
```

Inline SFINAE

```
validator
validate
is_valid
```

Overload sequences

```
overload_sequence
overload_sequence_r
overload_sequence_nothrow
overload_sequence_nothrow_r
forwarding_overload_sequence
forwarding_overload_sequence_r
forwarding_overload_sequence_nothrow
forwarding_overload_sequence_nothrow_r
overload_sequence_selector
```

Overload sets

```
overload_set
forwarding_overload_set
overload_set_selector
```

Type selection

```
type_selector
type_option
invalid_option
```

Pointers removal

```
remove_all_pointers
```

Inheritance

```
empty_base
is_instantiable
is_inheritable
inherit_if
```

Callables categorization

```
is_closure
is_functor
is_function_object
is_callable
```

Miscellaneous helpers

```
index_constant
type_t
false_v
true_v
tuple_index
remove_cvref
print_type
reporter
```

Related work

Additional type traits to serve as a basis of this work (C++ proposals P1016R0 and P1081R0)

TYPE UTILITIES: https://github.com/vreverdy/type-utilities

A reporter class to help with debugging

REPORTER: https://github.com/vreverdy/reporter

The numerical code for relativistic raytracing

MAGRATHEA-PATHFINDER: https://github.com/vreverdy/magrathea-pathfinder

Bit manipulation using some generic techniques (C++ proposal P0237R9)

THE BIT LIBRARY: https://github.com/vreverdy/bit

Acknowledgments

This work has been made possible thanks to a number of people, organizations and institutions:

The Laboratory for Computation, Data, and Machine Learning at the University of Illinois (LCDM)

An in particular Robert Brunner as well as Collin Gress, Minas Charalambides and Maghav Kumar.

The National Science Foundation (NSF)

- Award NSF-SI2-SSE-1642411: Award Scalable Tree Algorithms for Machine Learning Applications
- Award NSF-CCF-1647432: EAGER: Next Generation Tree Algorithms

Institutions

- The Astronomy Department of the University of Illinois at Urbana-Champaign (UIUC)
- The Laboratory Universe and Theories at the Observatory of Paris (LUTH/OBSPM)
- The National Center for Supercomputing Applications (NCSA)

Thank you for your attention

. . . and if you want to help with proposals to standardize some of these tools, don't hesitate to reach out . . .